# Pokročilé funkce knihovny Swing 4

V předchozím semináři jsme si představili základní třídy knihovny Swing pro tvorbu uživatelského rozhraní. V tomto semináři na to návažeme a ukážeme si některé pokročilejší vlastnosti, zejména si ukážeme složitější komponenty, tvorbu dialogových oken a další layout managery.

## 1 Rozšiřující možnosti komponent a práce s HTML

### 1.1 Formátovaný textový vstup

V minulém semináři jsme si představili komponentu `JTextArea`, která zajišťuje víceřádkový textový vstup. Tento textový vstup však není možné naformátovat, např. určit tučné písmo atp. Pokud chceme mít formátovány vstup, můžeme k tomu použít třídu `JEditorPane`. Nejpřímočařejší je použití této komponenty společně se vstupem ve formátu HTML, jak ukazuje následující příklad, kde vytvoříme objekt třídy `JEditorPane`, nastavíme, že obsahuje data ve formátu text/html, a samozřejmě nastavíme obsah.

```
JEditorPane editor = new JEditorPane();
editor.setContentType("text/html");
editor.setText("<html>Formatovany <b>text</b> s pomoci " +
               "<span style=\"color:red\">html a CSS</span>.</html>");
```

Třída `JEditorPane` umožňuje definovat i vlastní formát dat, ale použití HTML je nejjednodušší volba pro formátovaný vstup. Sluší se dodat, že možnosti jazyka HTML ve třídě `JEditorPane` jsou omezené a ani v nejmenším neodpovídají dnešní podobě jazyka HTML, ale pro základní formátování jsou dostačující. Pokud bychom potřebovali plnohodnotné jádro prohlížeče, je nutné šáhnout po JavaFX.

### 1.2 HTML a další komponenty

Na knihovně Swing je zajímavé, že i jiné komponenty mají podporu pro formátování pomocí jazyka HTML. Můžeme tedy jednoduše naformátovat textové popisky (`JLabel`) nebo tlačítka (`JButton`) jen tím, že jejich obsah uzavřeme do tagů `<html>...</html>`. Vyzkoušejme si to pro `JLabel` nebo `JButton`:

```
JLabel lbFoo = new JLabel("<html>Formatovany <b>text</b> s pomoci " +
                          "<span style=\"color:red\">html a CSS</span>.</html>");
```

Jedná se sice o zajímavou vlastnost, ale protože může mít značně rušivý vliv, je nutné s ní zacházet velice opatrně. Praktické použití to může mít například v situaci, kdy potřebujeme ve formuláři napsat vysvětlující komentáře a chceme některé pasáže zvýraznit.

### 1.3 Posuvníky

Pokud si zkusíme zadat do textové oblasti (`JTextArea`) nebo editoru (`JEditorPane`) rozsáhlejší text, který bude překračovat zobrazovanou oblast, zjistíme, že nejsou k dispozici posuvníky, které by umožňovaly pohodlné procházení rozsáhlejším textem. V knihovně Swing jednotlivé komponenty tuto funkcionalitu postrádají a je vyčleněna do samostatné třídy `JScrollPane`, která posuvníky zajišťuje. Použití je jednoduché, stačí požadovanou komponentu obalit do třídy `JScrollPane` a vložit do panelu, jak ukazuje následující příklad.

```
panel.add(new JScrollPane(editor));
```

## 2 Seznam a architektura MVC

Komponenty, které jsme si doposud představili, měly vcelku jednoduchou a z pohledu programu snadno uchopitelnou logiku. Stisk tlačítka vyvolá akci, textový vstup (klidně i víceřádkový) lze reprezentovat řetězcem. Pokud ale komponenta potřebuje pracovat se složitějšími daty, jakými jsou například seznam nebo tabulka, celá problematika se komplikuje. Na příkladu třídy `JList`, která slouží k výběru hodnoty nebo hodnot ze seznamu, si ukážeme princip, jak tuto problematiku řeší knihovna Swing.

Začneme tím nejjednodušším možným příkladem:

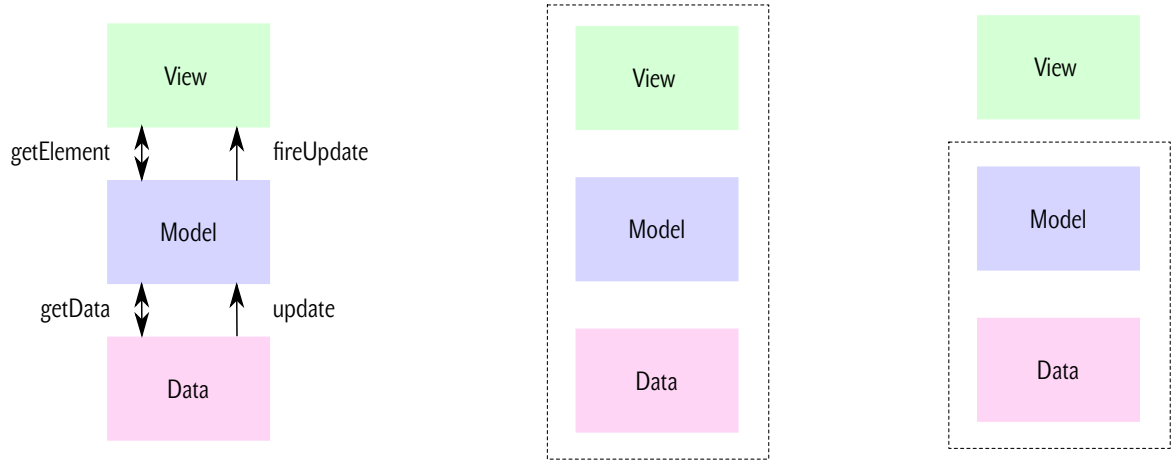
```
JList<String> fooList = new JList(new String [] {"Foo", "Bar", "Baz" });
```

Takto vytvořený seznam čítá právě tři položky, které nelze změnit. Pro některé situace to může být dostačující, ale velice často pracujeme se seznamem, jehož položky potřebujeme měnit. Než si ukážeme, jak toho dosáhnout, musíme si osvojit architekturu MVC (model-view-controller), kterou knihovna Swing používá.

V knihovně Swing jsou striktně oddělené jednotlivé vrstvy této architektury. To znamená, že jednotlivé grafické komponenty se starají pouze o prezentační stránku dat, tj. o vykreslení komponenty. Komponenta si žádná data neudržuje. Aby mohla komponenta vůbec nějaká data vykreslit, využívá vrstvu, která se označuje jako *model* a poskytuje jednotné rozhraní pro přístup k datům. V případě seznamu model poskytuje metody typu `getSize()`, která vrací počet položek, nebo `getElementAt(int index)`, která vrací položku s daným indexem. Tuto komunikaci ilustruje Obrázek 1 (vlevo). Díky těmto metodám je možné přistupovat k datům, která jsou uložena libovolným způsobem. Model funguje jako určitý adaptér, který převádí data z formy, jak s nimi pracuje program, do formy, ve které s nimi může pracovat grafická komponenta.

Vedle toho model plní ještě jednu důležitou úlohu, a to je informování grafické komponenty o tom, že došlo ke změně v datech. Ta pak na to může zereagovat svým překreslením. Model typicky poskytuje jednotné rozhraní, které udává, které položky byly přidány, změněny nebo odstraněny. Tyto metody jsou volány na základě změny dat nebo informace od *controller*u. Na Obrázku 1 jsou tyto dva zdroje změn sjednoceny pod položkou *Data*.

Ukažme si použití tohoto rozdělení na příkladu seznamu, resp. jeho modelu, který bude obsahovat celá sudá čísla.




Obrázek 1: Realizace architektury MVC v knihovně Swing (vlevo), vytvoření komponenty včetně dat a modelu (uprostřed), spojení modelu a dat (vpravo).

```
public class EvenIntsListModel extends AbstractListModel<Integer> {
    private int count;
    public EvenIntsListModel(int n) {
        this.count = n;
    }
    @Override public int getSize() {
        return count;
    }
    @Override public Integer getElementAt(int index) {
        return (index + 1) * 2;
    }
    public void add() {
        this.count++;
        // nutne, aby doslo k aktualizaci uzivatelskeho prvku
        fireIntervalAdded(this, 0, count);
    }
}
```

Model, který použijeme je třída `AbstractListModel<T>`, která má jeden typový parametr udávající typ hodnot v seznamu, v našem případě jsou to celá čísla. Dále zde máme atribut, který obsahuje počet čísel, které jsou v seznamu, a který je inicializován v konstruktoru. V námi vytvořené třídě musíme implementovat metody `getSize()` a `getElementAt(int)`, které definují obsah seznamu. Všimněme si, že v tomto případě nikde nevytváříme objekt obsahující seznam celých sudých čísel, jen odpovídáme na jednotlivé požadavky od třídy, která se stará o vykreslení dat. Prakticky můžeme prezentovaná data brát odkudkoliv, např. z databáze nebo síťové komponenty. Proto, abychom vytvořili validní model pro seznam typu `JList` jsou tyto dvě metody dostačující. V našem příkladu je ještě metoda `add`, která umožňuje přidat další číslo do seznamu. U této metody si všimněme volání `fireIntervalAdded`, které se postará o upozornění na změnu seznamu, konkrétně, že byla přidána nová hodnota.

Pokud chceme takto vytvořený model použít, předáme jej do konstrukturu třídy `JList`.

```
EvenIntsListModel listModel = new EvenIntsListModel(10);
JList<Integer> mainList = new JList<>(listModel);
```

Takto pojaté rozdělení funkcí při tvorbě uživatelského rozhraní sice nabízí zajímavé možnosti, avšak v drtivě většině případů je to příslovečný kanón na vrabce. Proto většina komponent umí navíc vytvořit vhodný model pro data, se kterými má pracovat, jako v příkladu ze začátku kapitoly, viz Obrázek 1 (uprosřed). Nebo jsou k dispozici přichystané třídy, které v sobě již spojují model a vhodné uložení dat, viz Obrázek 1 (vpravo). V případě seznamu máme k dispozici třídu `DefaultListModel<T>`, která jednak implementuje abstraktní metody `AbstractListModelu` a současně obstaráva uložení dat ve formě arraylistu.

Přístup, kdy se uplatňuje členění založené na architektuře MVC se používá i u dalších grafických komponent, např. tabulky `JTable` (viz přiložený kód) nebo stromů `JTree`.

# 3 Dialogy

U běžných aplikací obvykle nevystačíme s jedním oknem a potřebujeme od uživatelů získávat další vstupní hodnoty. K tomu slouží dialogy.

## 3.1 Vyvolání dialogu

V principu se práce s dialogy příliš neliší od práce s běžnými okny. Dialog dědí ze třídy `JDialog` a potomek by v konstruktoru měl přijímat odkaz na okno, které dialog vyvolalo. Vyvolání dialogu ukazuje následující příklad.

```
DialogExample dlg = new DialogExample(this);
dlg.setModal(true);
dlg.setVisible(true);
// můžeme zpracovat výsledek z dialogu
dlg.dispose();
```

Nejdříve vytvoříme objekt dialogu a předáme mu odkaz na hlavní okno. Následně nastavíme, že se jedná o modální dialog. To má dva důsledky. Jednak se z dialogu nebude možné přepnout do rodičovského okna [1], a tím pádem okno zůstane nad rodičovským, dokud jej nezavřeme. Dále, pokud máme okno jako modální, zavolání metody `setVisible(true)` zajistí, že volající metoda nebude dál pokračovat, dokud se okno nezavře. To umožňuje rozumně předávat výsledky mezi metodami, které dialogy otevírají a samotnými dialogy. Po skončení práce s dialogem je nutné uvolnit všechny zdroje, které byly s dialogem spojené, tj. zavolat metodu `dispose`.

## 3.2 Ukončení dialogu

V předchozí podkapitole jsme na práci s dialogem nahlíželi z pohledu rodičovského okna, které otevírá dialog. Zbývá ujasnit, jak by měla vypadat komunikace z druhé strany, z pohledu samotného dialogového

[1] Proto jsme ten odkaz předávaly do konstruktoru

okna.

Co se týče obsahu, vytvoření komponent pro dialog je shodné s tím, co jsme viděli u běžných oken. Avšak je dobrým zvykem, že dialogové okno má minimálně dvě tlačítka s významem *Ok* (potvrzení dialogu a jeho zavření) a *Cancel* (zavření dialogu). S těmito tlačítky přirozeně souvisí i předávání hodnot mezi dialogem a volajícím oknem, kdy v prvním případě (*Ok*) potřebujeme předat informace z dialogu a v druhém případě (*Cancel*) potřebujeme informovat volající metodu, že uživatel dialog zavřel a nechce danou operaci provést.

Řešení tohoto problému je ve své podstatě velmi banální. Stačí si uvědomit, že dialogové okno je objekt jako každý jiný, který může mít své atributy. Zavedeme si například atribut `result`, do kterého při zavření okna uložíme výsledek, případně příznak `null`, pokud uživatel stisknul tlačítko *Cancel*. Následující kód ukazuje, jak by mohly vypadat listenery pro tlačítka *Ok* a *Cancel*.

```
btnOk = new JButton("Ok");
btnOk.addActionListener((ActionEvent e) -> {
    result = /* nastav vracene hodnoty */;
    setVisible(false);
});
btnCancel = new JButton("Cancel");
btnCancel.addActionListener((ActionEvent e) -> {
    result = null;
    setVisible(false);
});
```

Zpracování výsledku na straně volajícího okna je přímočaré.

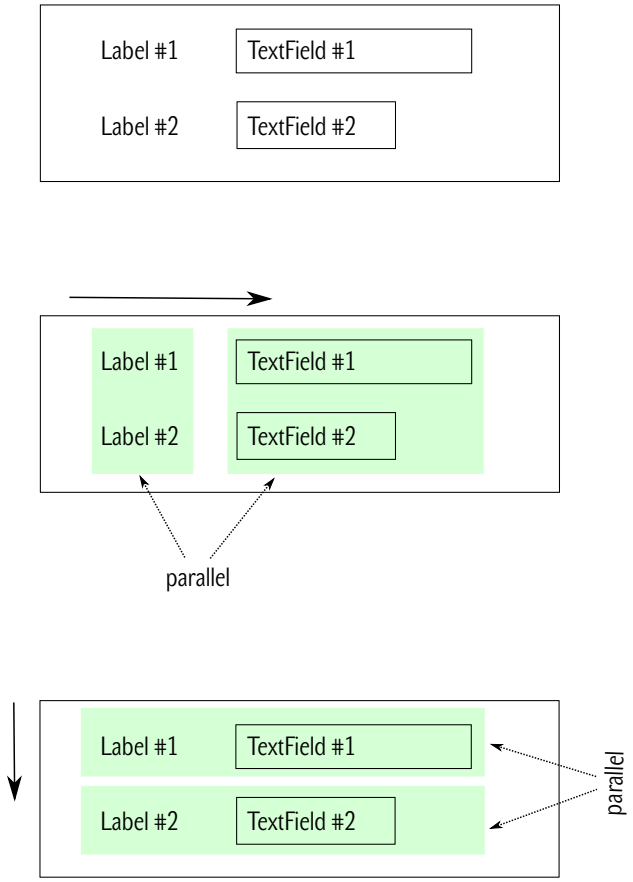
```
if (dlg.getResult() != null) {
    /* pouzij hodnotu */ dlg.getResult();
}
```

### 3.3 Další Layout Managery

V přiložené ukázkové aplikaci je práce s dialogy demonstrována na jednoduchém formuláři se dvěma popisky, dvěma textovými vstupy a dvěma tlačítky. U daného formuláře stojí za povšimnutí dva layout managery, které jsou v něm použity. Jednak se jedná o `GridLayout`, který rozmisťuje jednotlivé komponenty do mřížky. V ukázkovém formuláři je použit k rozmístění tlačítek.

Dále je v ukázkové aplikaci použit `GroupLayout`, což je docela sofistikovaný layout manager, ve kterém definujeme vzájemné vztahy v rozmístění komponent a layout manager se postará o jejich rozmístění. Princip `GroupLayout` je postaven na tom, že pro jednotlivé komponenty a případně jejich skupiny určíme, zda jdou za sebou nebo jsou vedle sebe. Toto uspořádání pak musíme přirozeně určit v horizontálním a vertikálním směru formuláře. Tento princip ilustruje Obrázek 2.

Použití tohoto layout manageru si ukážeme v následujícím kódu pro horizontální směr, pro vertikální směr je princip rozložení analogický a můžeme jej nalézt v přiložených zdrojových kódech.




Obrázek 2: Princip pozicování komponent s pomocí `GroupLayout`. Cílové rozložení (nahoře), horizontální rozložení (uprostřed), vertikální rozložení (dole). Zelená barva vyznačuje skupiny komponenty, které jsou vedle sebe.

```
layout.setHorizontalGroup(
   layout.createSequentialGroup()
            .addGroup(layout.createParallelGroup()
                              .addComponent(lbFirst)
                              .addComponent(lbSecond))
            .addGroup(layout.createParallelGroup()
                              .addComponent(txtFirst)
                              .addComponent(txtSecond)));
```

Nejdříve definujeme, že máme skupiny komponent, které jdou jsou v horizontálním směru umístěny za sebou (sekvenčně). Toho docíleme tak, že vytvoříme `SequentialGroup`. Tato skupina bude obsahovat další dvě skupiny komponent, jedna s popisky a druhá s textovými vstupy. V rámci každé z těchto skupin jsou komponenty v horizontálním směru umístěny vedle sebe (paralelně), proto je vložíme do `ParallelGroup` pomocí volání metody `createParallelGroup`. Toto rozložení ilustruje Obrázek 2 (uprostřed). Analogicky pak vertikální směr.

Výhoda tohoto layout manageru spočívá v tom, že umožňuje definovat poměrně komplikované rozložení, včetně upřesnění některých vlastnostní, ať už se to týká jednotlivých komponent, kdy můžeme definovat minimální, preferované a maximální rozměry při vložení kompomenty. Jak ukazuje následující kousek kódu.

```
.addComponent(txtSecond, GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE, 200)));
```

Případně můžeme nechat layout manager vytvořit vhodné mezery mezi komponentami a tím dosáhnout estetičtějšího vzhledu. Nastavení ukazuje následující kód.

```
layout.setAutoCreateContainerGaps(true);
layout.setAutoCreateGaps(true);
```

## Závěr

V tomto a předchozím semináři jsme si ukázali základní principy tvorby grafických aplikací s knihovnou Swing a představili jsme si základní komponenty, které lze použít. Tento výčet nebyl ani v nejmenším vyčerpávající a je na p.t. čtenářích, aby prozkoumali další komponenty a jejich možnosti, které knihovna Swing nabízí.