Java pro geoinformatiky

Tomáš Bayer, Michal Schneider

Obsah

1 Než začneme programovat... 7

1.1 Charakteristika Javy . . . 7
1.2 Použití Javy . . . 8
1.3 Verze Javy . . . 9
1.4 Vizuální vývojová prostředí . . . 9
1.5 Komponenty jazyka Java . . . 9
1.6 Java a její specifika . . . 10

2 První program v jazyce Java 14

2.1 Tvorba dokumentace . . . 16
2.1.1 Javadoc . . . 17
2.2 Balíčky v Javě . . . 17
2.2.1 Práce s balíčky . . . 18

3 Datové typy a proměnné v Javě 20

3.1 Základní datové typy . . . 20
3.1.1 Celočíselné datové typy . . . 20
3.1.2 Znakový typ . . . 21
3.1.3 Řetězcové konstanty . . . 22
3.1.4 Logický datový typ . . . 22
3.2 Reálné datové typy . . . 22
3.3 Deklarace proměnných . . . 23
3.4 Přiřazování a přetypování . . . 24
3.5 Operátory v Javě . . . 26
3.6 Matematické metody . . . 28
3.7 Práce se standardním formátovaným vstupem a výstupem . . . 28
3.7.1 Formátovaný výstup . . . 29

3.7.2 Standardní formátovaný vstup . . . 30
3.8 Příkazy v Javě . . . 31
3.9 Pole . . . 31
3.9.1 Kopírování polí . . . 33

4 Větvení programu a opakování **34**
4.1 Přehled příkazů pro větvení programu . . . 34
4.1.1 Konstrukce if-else . . . 34
4.1.2 Operátor ? . . . 35
4.1.3 Příkaz switch . . . 36
4.2 Přehled příkazů pro opakování . . . 37
4.2.1 Cyklus while . . . 37
4.2.2 Cyklus for . . . 38
4.2.3 Cyklus do while . . . 39
4.2.4 Vnořené cykly . . . 39
4.2.5 Příkaz break . . . 40
4.2.6 Příkaz continue . . . 40

5 Objekty, třídy, metody **42**
5.1 Metody . . . 42
5.1.1 Metody třídy . . . 43
5.1.2 Implicitní a explicitní konverze. . . . 46
5.1.3 Rekurzivní metody . . . 46
5.1.4 Přetěžování metod . . . 47
5.2 Třídy . . . 47
5.2.1 Deklarace třídy . . . 48
5.2.2 Tělo třídy a přístupová práva . . . 48
5.3 Operace s třídami . . . 49
5.3.1 Vytvoření objektu . . . 50
5.3.2 Přístup k datovým položkám . . . 50
5.3.3 Rušení objektů . . . 52
5.3.4 Konstruktor . . . 52
5.3.5 Odkaz this . . . 54
5.3.6 Inicializace objektu jiným objektem . . . 54
5.3.7 Spuštění metody v konstruktoru. . . . 55
5.3.8 Práce se statickými členy třídy . . . 55
5.3.9 Gettery a settery . . . 57
5.3.10 Pole objektů . . . 57
5.3.11 Předávání objektů jako parametrů . . . 58

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

6 Dědičnost **61**

6.1 Deklarace rodičovské a odvozené třídy . . . 61

6.1.1 Inicializace rodičovské třídy . . . 63

6.1.2 Inicializace odvozené třídy . . . 63

6.2 Předefinování a přetížení metody . . . 64

6.3 Finální metody a finální třídy . . . 65

6.4 Abstraktní metody a abstraktní třídy . . . 66

6.5 Konverze mezi instancemi rodičovské a odvozené třídy . . . 67

6.6 Polymorfismus . . . 67

7 Výjimky **69**

7.1 Druhy výjimek . . . 69

7.2 Ošetření výjimek . . . 70

7.2.1 Ošetření výjimky propagací . . . 71

7.2.2 Ošetření výjimky metodou chráněných bloků . . . 72

7.2.3 Ošetření výjimky metodou chráněných bloků a její předání výše . . . 74

7.3 Tvorba vlastních výjimek . . . 76

8 Rozhraní **78**

8.1 Implementace jednoho rozhraní třídou . . . 78

8.2 Implementace více rozhraní třídou . . . 81

8.3 Rozhraní a dědičnost . . . 82

8.4 Vnitřní třídy . . . 82

8.4.1 Vnitřní třídy a rozhraní . . . 84

8.4.2 Anonymní vnitřní třídy . . . 85

9 Dynamické datové struktury **87**

9.1 Kontejnery . . . 88

9.2 Iterátory . . . 89

9.3 Rozhraní List . . . 89

9.3.1 ArrayList . . . 90

9.4 Rozhraní Set . . . 91

9.4.1 TreeSet . . . 92

9.5 Rozhraní Map . . . 92

9.5.1 TreeMap . . . 93

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"

10 Java, grafické uživatelské rozhraní **95**

10.1 Základní komponenty a události . . . 96

10.1.1 Rozdělení komponent . . . 96

10.1.1.1 Top-level komponenty . . . 97

10.1.1.2 Kontejnerové komponenty . . . 97

10.1.1.3 Základní komponenty . . . 97

10.1.1.4 Needitovatelné informační komponenty . . . 98

10.1.1.5 Interaktivní komponenty . . . 98

10.1.2 Formuláře . . . 99

10.1.2.1 Nastavení vzhledu komponent . . . 101

10.1.2.2 Nastavování parametrů grafického rozhraní v konstruktoru jiné třídy . . . 102

10.1.2.3 Nastavování parametrů grafického rozhraní v konstruktoru téže třídy . . . 102

10.1.3 Přidávání dalších komponent na formulář . . . 104

10.1.4 Práce s událostmi . . . 107

10.1.4.1 Ošetření stejné události u více komponent . . . 110

10.1.4.2 Několik posluchačů téže události . . . 112

10.1.5 Barvy, fonty, grafický vzhled komponent . . . 112

10.1.5.1 Barvy v Javě . . . 112

10.1.5.2 Fonty v Javě . . . 114

10.1.5.3 Základní vlastnosti komponent . . . 114

10.1.6 Layout managery . . . 116

10.1.6.1 FlowLayout . . . 117

10.1.6.2 GridLayout . . . 118

10.1.6.3 BorderLayout . . . 118

10.1.7 Modely v Javě . . . 119

10.1.8 Přehled nejpoužívanějších komponent . . . 120

10.1.8.1 JLabel . . . 120

10.1.8.2 JButton . . . 121

10.1.8.3 JCheckBox . . . 121

10.1.8.4 JRadioButton . . . 123

10.1.8.5 JTextField . . . 125

10.1.8.6 JComboBox . . . 128

10.1.8.7 JList . . . 129

10.1.8.8 JTable . . . 134

10.1.8.9 Dialogová okna . . . 138

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"

CZECH SPACE OFFICE

10.1.8.10 JFileChooser 142
10.1.8.11 JPanel 144
10.1.9 Grafika 146
10.1.9.1 Základní entity a jejich kresba 147
10.1.9.2 Příklady 149
10.1.9.3 Práce s obrázky 155

11 Aplety **161**
11.1 Struktura appletu 161
11.1.1 Životní cyklus appletu 161
11.2 Tvorba appletu 162
11.3 Spuštění appletu 164
11.3.1 Java archivy 165
11.3.2 JAR soubory a aplety 166
11.3.3 Applet viewer 167
11.3.4 Spouštění appletu jako aplikace 168

12 Java a databáze **170**
12.1 Komunikace s databázovým systémem 171
12.2 Získávání metadat o databázi 172
12.3 Vkládání dat do databáze 173
12.4 Výběr dat z databáze 176
12.5 Vymazání dat z databáze 177
12.6 Aktualizace dat v databázi 178
12.7 Další příklady 179
12.8 Závěr 180

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

Úvod

Tento materiál je věnován základům programovacího jazyku Java, který je velmi často používaným a populárním nástrojem pro vývoj aplikací všeho druhu. Není jazykem nejjednodušším, jeho přeložený kód není nejrychlejší. Je to však velmi efektivní, univerzálně použitelný, dynamicky se rozvíjející jazyk vhodný pro výuku programování. Jedná se o jazyk silně typový, objektově orientovaný. Uživatel se při práci s ním seznámí s technikou hierarchického popisu problému prostřednictvím objektového modelu. Tento způsob myšlení mu poskytne zcela jiný pohled na řešení úloh a úkolů, se kterými se v geoinformatické praxi běžně setkává.

Jednou z výhod jazyka Java je její multiplatformita, tj. schopnost běžet na prakticky libovolném operačním systému. V současné době (jaro 2007) byl zdrojový kód jazyka Java firmou Sun otevřen, představuje tedy open source platformu pro vývoj aplikací.

Materiál není zaměřen na konkrétní operační systém ani konkrétní vývojové prostředí. Nejedná se o ucelené dílo, novou učebnici jazyka Java, takových je na současném trhu dostatek. Představuje soubor přednášek převedených do elektronické formy, který studentům geoinformatiky poskytne první seznámení se světem programování. Zájemcům o hlubší studium můžeme doporučit zejména [2] [3], které se staly standardem, a vychovaly celou řadu výborných programátorů. Z nich, a z mnoha jiných, čerpá i tato publikace.

Kapitoly 1-8, 10, 11 napsal Tomáš Bayer, kapitoly 9, 12 Michal Schneider. Většina příkladů byla odladěna ve vývojovém prostředí NetBeans firmy Sun.

Předem děkujeme za případné připomínky, které přispějí ke zlepšení tohoto textu zaslané na adresy:

`bayertom@natur.cuni.cz` nebo `schneide@natur.cuni.cz`.

Praha, říjen 2007.

Tomáš Bayer
Michal Schneider

Pro sazbu tohoto materiálu byl použit systém LaTeX.

Materiál neprošel recenzním řízením.

Kapitola 1

Než začneme programovat...

1.1 Charakteristika Javy

Java patří mezi poměrně nové programovací jazyky. Uveďme některá zajímavá fakta o tomto jazyku. Java vznikla v roce 1995 v laboratořích firmy Sun. Původně měla být použita především pro ovládání inteligentních domácích spotřebičů jako ledniček či mikrovlnných trub.

Název Java lze přeložit jako káva, resp. espreso. V logu nalezneme kouřící hrneček s kávou, jména řady komponent souvisí také s kávou, např. Java Beans (kávové boby). Jazyk se měl jmenovat Oak, údajně podle dubu, který rostl tvůrci, J. Goslingovi, pod okny jeho domu. V současné době používá Javu několik desítek miliónů programátorů po celém světě.

Javu lze ji charakterizovat několika následujícími body:

- *Objektově orientovaný programovací jazyk*

  Téměř vše v Javě je objektem. Stejně jako C++, ze kterého vychází, umožňuje programování bez nutnosti znalosti objektově orientovaného programování. Tento přístup však neumožňuje řešení složitějších problémů a není doporučován.

- *Nezávislost na platformě*

  Program napsaný v jazyce Java běží bez nutnosti provádět jakékoliv úpravy zdrojového kódu prakticky na všech operačních systémech.

- *Silná typovost jazyka*

- *Využití v prostředí Internetu*

  Java umožňuje snadné používání v prostředí sítě Internet, někdy bývá nazývána "jazykem Internetu".

- *Integrace grafického a síťového rozhraní, možnost práce s multimédii*

  Java obsahuje grafiku nezávislou na zařízení zabudovanou přímo do jazyka. Grafické operace jsou standardizovány a poskytují stejné výsledky bez ohledu na typ počítače či používaný operační systém.

- *Zdarma*

  Java je k dispozici včetně několika vývojových prostředí zdarma.

- *Jednodušší syntaxe*

  Z důvodů zjednodušení práce vývojáře došlo i ke zjednodušení syntaxe Javy vzhledem k C++.

- *Vysoký výkon*

  Snadná práce s objekty prostřednictvím tzv. garbage collectoru.

- *Bezpečnost*

  Odstranění řady potenciálně nebezpečných konstrukcí, které bývají zdrojem častých a těžko odhalitelných chyb.

1.2 Použití Javy

Java má všestranné použití. Díky své relativní jednoduchosti je také vhodným jazykem pro výuku objektově orientovaného programování. Java je využívána zejména pro tvorbu:

- aplikací ve formě samostatných programů, toto použití však není zcela typické.
- appletů, tj. krátkých programů běžících v okně webového prohlížeče.
- servletů, tj. programů běžících na serverech vytvářejících dynamické stránky.
- distribuovaných systémů, tj. programů skládajících se z více částí běžících současně na větším množství počítačů.
- aplikací představující mezivrstvu pro komunikaci s databázovými stroji či síťovou komunikaci.
- aplikací pro mobilní či telekomunikační zařízení.

Java má však i své nevýhody, které se dají shrnout do následujících bodů:

- Hardwarová náročnost. Dána vestavěnou kontrolou práce s pamětí, která je mnohem detailnější než v C++. Doporučená HW konfigurace představuje cca. 1GB RAM.
- Rychlý vývoj jazyka. Vývoj JDK je velmi rychlý, některé funkce jsou "zastaralé" pár měsíců po vydání JDK a v další verzi kompletně přepracovány. V současných verzích jsou již základní funkce standardizovány.
- Rozsáhlost Javy je současně i její nevýhodou, disponuje řádově tisíci funkcemi a nutnost prostudování značného množství literatury.
- Složitější práce se standardním vstupem/výstupem.
- Poněkud nižší rychlost běhu programů způsobená tím, že se nejedná o jazyk kompilovaný (cca. 2x nižší než u C++). První verze byly až 30 pomalejší, s každou další novou verzí se tento rozdíl stírá.
- Délka zápisu kódu pro realizaci standardních operací je u Javy větší než u jiných programovacích jazyků.

V současné době se poměrně dynamicky rozvíjí prostředí .NET firmy Microsoft, konkrétně jazyk C#. Využívá podobný přístup jako Java, pro běh programů je nutno, aby na cílovém zařízení byl přítomen virtuální stroj, tzv. .NET framework. Syntaxe jazyka C# je velmi podobná syntaxi Javy. Prostředí .NET bylo portováno na jiné operační systémy v rámci projektu Mono. Další informace lze nalézt na `http://www.mono-project.org`.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"

1.3 Verze Javy

Java je jazyk dynamicky se vyvíjející. V současné době (2007) máme k dispozici již jeho šestou verzi. Verze Javy jsou označovány buď desetinnými čísly JDK 1.2, 1.3, 1.4 nebo Java 2, 3, 4; aktuálně je dostupné JDK 6.0.

Pro vývoj aplikací v jazyce Java je nutno mít nainstalováno prostředí JDK, tzv. Java Development Kit, pro běh aplikací prostředí JRE (Java Runtime Environment). Tato prostředí lze stáhnout ze stránek `http://www.sun.com/java`.

Rozdíly mezi verzemi spočívají především v kvalitnější implementaci knihoven, přidávání nových funkcí a komponent, např. pro grafické či databázové rozhraní. Od Javy 1.2 bylo nahrazeno původní grafické rozhraní AWT (Abstract Windows Toolkit) novým grafickým rozhraním Swing.

1.4 Vizuální vývojová prostředí

Java umožňuje vytváření klasických "okenních" aplikací. Lze tak činit přímým zápisem zdrojového kódu nebo ve speciálních nástrojích nazývaných RAD (Rapid Application Development).

Tyto nástroje umožňují provádět vizuální návrh aplikací jejich sestavováním z předem připravených vizuálních komponent. Děje se tak zpravidla pouhým přetahováním jednotlivých grafických komponent (tlačítek, přepínačů, dialog boxů,...) myší s následným nastavením vzhledových a funkčních parametrů. Zdrojový kód k těmto akcím je generován vývojovým prostředím automaticky. Tyto nástroje bývají nazývány také GUI buildery.

Upozorněme, že takový kód často nebývá tak efektivní jako kód napsaný programátorem. Cílem RAD nástrojů je usnadnění vizuálního návrhu aplikace, programátor může ušetřený čas věnovat na zlepšení funkčnosti aplikace.

Vizuální vývojová prostředí jsou k dispozici jak v komerčních, tak i bezplatných verzích. Přehled RAD nástrojů:

- Eclipse (http://www.eclipse.com): Nejčastěji používané vizuální vývojové prostředí. K dispozici zdarma, některé nástroje a funkce však pouze v placené verzi.
- NetBeans (http://www.netbeans.org): Původně české vývojové prostředí vzniklé na MFF UK, koupila ho firma Sun. Disponuje velmi kvalitním layout managerem nazývaným Mattise. K dispozici i se všemi doplňky zdarma.
- JBuilder (http://www.borland.com): Vývojové prostředí od firmy Borland, k dispozici zdarma.
- IntelliJ IDEA (http://www.jetbrains.com): V současné době jedno z nejlepších vývojových prostředí pro Javu.
- Sun Java Studio (http://www.sun.com): Nástroj pro vývoj enterprise aplikací založených na jazyce Java.

1.5 Komponenty jazyka Java

Java je tvořena třemi základními komponentami:

- Programovací jazyk Java

- Javovský virtuální stroj (JVM)
- Aplikační rozhraní (API)

Java se liší od ostatních programovacích jazyků přístupem, jakým je program překládán a spouštěn. Stručné vysvětlení nalezneme v následujících odstavcích. Java je jazyk ležící na rozhraní mezi kompilovanými a interpretovanými jazyky. U jazyků kompilovaných je zdrojový kód přeložen do strojových instrukcí procesoru. Takový kód může být následně spuštěn. U jazyků interpretovaných je překlad programu opakovaně prováděn při jeho spuštění, aby program mohl vykonávat svoji funkci, je nutná přítomnost tzv. interpretru, který provádí překlad programu do strojového kódu.

Bajtový kód. Zdrojový kód programu představovaný textovým souborem s koncovkou *.JAVA je přeložen do jazyka javovského virtuálního stroje ve formě tzv. bajtového kódu. Bajtový kód tvořený souborem *.CLASS může být spuštěn pouze na virtuálním počítači představovaným JVM, nelze ho spustit přímo na cílovém počítači. Bajtový kód je tak nezávislý na konkrétním hardwarovém a softwarovém vybavení počítače.

Javovský virtuální stroj. JVM následně provádí převod bajtového kódu do strojového kódu příslušného procesoru, na kterém je právě spuštěn. Tomuto procesu říkáme interpretace. Překlad do bajtového kódu je proveden pouze jednou, interpretace při každém spuštění javovského programu. Upozorňujeme, že bajtový kód lze zpětně velmi snadno dekompilovat a získat výpis zdrojového kódu původního programu. Z tohoto důvodu by zdrojový kód neměl obsahovat žádné citlivé informace jako např. hesla, která jsou z něj snadno extrahovatelná.

Aplikační rozhraní. Aplikační rozhraní Javy (tzv. Java API) představuje seznam knihoven a softwarových komponent, obsahující řadu nástrojů pro práci se soubory, databázemi, řetězci, grafikou, třídící algoritmy,... Jsou sdružovány do balíčků zvaných packages. Jeden balíček obsahuje komponenty, které spolu souvisí logicky či tématicky. Tyto nástroje ušetří programátorovi práci, protože využívá již ověřené komponenty.

1.6 Java a její specifika

Java je založena na jazyce C++, je však více objektově orientovaným programovacím jazykem než C++. Lze prohlásit, že se jedná o čistokrevný objektově orientovaný programovací jazyk. V Javě je (prakticky) vše objektem, tento jednotný přístup umožňuje jednodušší osvojení základů jazyka a jeho používání. Programátor nemusí přemýšlet, zda pracuje či nepracuje s objektem, využívá tedy jednotnou syntaxi.

Java obsahuje pokročilé nástroje pro práci s objekty. Pokud nejsou některé z níže uvedených pojmů čtenáři jasné, v dalším textu se s nimi seznámí podrobněji. Poměrně často budeme používat srovnání Javy s jazykem C++, který je považován za "standard" v oblasti programování. Čtenář, který tento jazyk nezná, může poznámky přeskočit. Čtenáři znalému jazyka C++ pomůže ulehčit přechod na jazyk Java a vyvarovat se některých základních chyb plynoucích z podobnosti implementace, ale rozdílné funkčnosti kódu. V dalším textu budeme pro zápis zdrojového kódu používat neproporcionální font.

Identifikátor objektu. Identifikátor objektu představuje odkaz na něj, jedná se o obdobu ukazatelů v používaných např. v jazyce C++. Zápis

```
String o
```

znamená vytvoření odkazu `o` na objekt typu `String`. S většinou datových typů v Javě zacházíme "objektově", hovoříme o tzv. objektových datových typech.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

Dynamická tvorba objektů. Java zjednodušuje práci s objekty, na rozdíl od C++ lze pracovat pouze s dynamicky vytvořenými objekty. Jsou vytvářeny prostřednictvím operátoru `new`, nemusíme se však starat o jejich rušení.

Garbage collector. Java disponuje automatickou správou paměti, tento nástroj nazýváme garbage collector. Stará se o automatické odstraňování nepoužívaných objektů (tj. objektů, na které již neexistuje odkaz). Odstraňuje tak spoustu programátorských chyb souvisejících s životností objektů. Okamžik smazání objektu není možné ovlivnit, provede se, až to systém uzná sám za vhodné. Garbage collector lze vyvolat ručně pomocí `System.gc`.

Inicializace odkazu na objekt. Výše uvedený zápis nevede k vytvoření samostatného objektu. Pokud bychom se pokoušeli s takovým odkazem pracovat, dojde k výjimce (tj. systémové chybě). Odkaz zatím není propojen na žádný objekt. Konstrukce

```
o=new String("Ahoj");
```

vede k vytvoření inicializovaného odkazu. Inicializace je provedena prostřednictvím řetězce typu `String`. Oba kroky lze spojit do jednoho.

```
String o=new String("Ahoj");
```

Platí zásada: pokud vytvoříme odkaz, zpravidla k němu vytváříme i objekt. V Javě lze výše uvedenou konstrukci zapsat jednodušší formou.

```
String objekt="Ahoj";
```

Výsledek této operace je stejný, neilustruje však tak názorně práci s objekty.

Primitivní datové typy. V Javě existují tzv. primitivní datové typy. Představují speciální skupinu datových typů, se kterými se zachází "neobjektově". Používají se v případě, kdy by byl objektový přístup neefektivní, např. při práci s běžnými proměnnými; Java v tomto případě kopíruje přístup jazyka C++. Takové proměnné vytváříme jako tzv. *statické proměnné*, mají některé speciální vlastnosti, se kterými se seznámíme v dalším textu. Ke každému primitivnímu datovému typu existuje jeho objektová varianta. Kromě vlastností se liší i zápisem, název primitivní varianty začíná malým písmenem, název objektové velkým písmenem, viz. tab. 1.1.

Primitivní typ	Objektový typ	Velikost
`boolean`	`Boolean`	-
`char`	`Char`	8b
`byte`	`Byte`	8b
`short`	`Short`	16b
`int`	`Integer`	32b
`long`	`Long`	64b
`float`	`Float`	32b
`double`	`Double`	64b

Tabulka 1.1: Primitivní a objektové datové typy v Javě.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

Velikost datových typů. Každý datový typ má v Javě na různých počítačích s různými operačními systémy vždy stejnou velikost, Java je tedy na rozdíl od ostatních programovacích jazyků přenositelná mezi různými platformami.

Pole v Javě. Používání polí v Javě je mnohem bezpečnější než v C++, nejedná se totiž o pouhé paměťové bloky. Nemůže dojít k situaci, kdy při práci s polem přesáhneme index pole. Kontrola indexu pole je začleněna přímo do jazyka Java za cenu drobného zatížení paměti. Při vytvoření pole je zaručena inicializace jeho jednotlivých prvků. Lze vytvářet pole primitivních i objektových datových typů.

Obor platnosti primitivních datových typů. U Javy jsou drobné rozdíly při určování platnosti objektů. V Javě neexistuje možnost definice stejné proměnné ve vnořeném bloku jako v nadřazeném bloku. Podívejme se na následující příklad.

```
{
  int a=10;
  int b=20; //k dispozici promenne a, b
   {
    double c=30.0; //k dispozici promenne a, b, c
     {
      double d=40.0; //k dispozi promenne a, b, c, d
      int a=50; //nelze
     }
    // k dispozici promenne a, b, c
   }
  //k dispozici promenne a, b
}
```

Obor platnosti dynamických datových typů. Životnost dynamických datových typů je jiná než v případě primitivních datových typů. Objekt zůstává v paměti i po ukončení rozsahu platnosti.

```
{
 String objekt=new String("Ahoj");
}//konec oboru platnosti
```

K tomuto objektu se však již není možno žádným způsobem dostat, protože odkaz je po ukončení bloku mimo rozsah platnosti. Takové objekty jsou rušeny prostřednictvím garbage collectoru. Ten hledá, zda na objekt vytvořený prostřednictvím operátoru `new` existuje ještě nějaká vazba prostřednictvím odkazu. Pokud neexistuje, je objekt zrušen.

Datový typ	Hodnota
`boolean`	`false`
`char`	`null`
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0`
`double`	`0.0`

Tabulka 1.2: Přehled datových typů a jejich inicializace

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"

Implicitní hodnoty primitivních datových typů. Při vytvoření proměnné jako datové položky třídy dochází k přiřazení výchozí hodnoty, i když proměnná zatím nebyla inicializována. S tímto přístupem se v jazyce C++ nesetkáme. Každá datová položka třídy je při svém vytvoření automaticky inicializována. Inicializace se netýká proměnných, které nejsou datovými položkami třídy. Pokud neprovedeme inicializaci takové proměnné, Java nás na rozdíl od jazyka C++ na tuto chybu upozorní a takový krok označí jako chybu.

```
int x;
x++;//chyba, promenna nebyla inicializovana
```

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

Kapitola 2

První program v jazyce Java

V této kapitole vytvoříme jednoduchý program v jazyce Java, přeložíme ho do spustitelného tvaru a vygenerujeme k němu nápovědu ve formátu HTML. Seznámíme se se syntaxí jazyka Java, základy práce s JVM a jeho konfigurací.

Volba editoru. Otázkou je, jaký však zvolit editor pro zápis zdrojového kódu? Lze použít libovolný editor pracující s ASCII kódem. Pro tyto účely se pod operačním systémem Windows hodí např. Poznámkový blok, pod operačním systémem Linux např. gEdit. Nelze použít klasický textový editor jako Word či Writer, do textu jsou vkládány dodatečné informace o formátování.

Zápis zdrojového kódu. Název souboru se zdrojovým kódem v Javě musí být totožný s jménem třídy a to včetně velkých i malých písmen. Vytvoříme nový dokument s názvem `Pokus.java`, do kterého zapíšeme následující text.

```
public class Pokus
  {
   public static void main(String [] args)
    {
     System.out.println("Tohle je muj");
     System.out.println("prvni program v Jave");
    }
  }
```

A máme hotový první program v jazyce Java.

Překlad programu. Program následně přeložíme do bajtového kódu. Použijeme k tomu vestavěný překladač javac, který se pro verzi JDK 6.0 nachází pod operačním systémem Windows v následující složce: `C:\Progra~1\Java\jdk1.6.0\bin\javac.exe`. Pro operační systém Linux ho nalezneme zpravidla v adresáři `/etc/lib/jvm/java-6-sun-1.6.0.00/bin`.

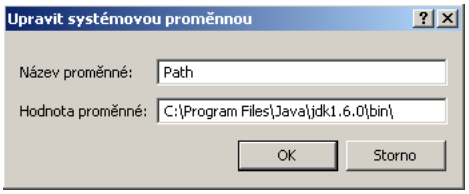
Předpokládáme, že se soubor Pokus.java nachází v adresáři totožném s pracovním adresářem, např. ve složce C:\Dokumenty. Do příkazového řádku zadejme následující příkaz:

```
javac Pokus.java
```

Příponu `.java` není možné vynechat. Pokud se objevilo hlášení

```
C:\>javac
javac není názvem vnitřního ani vnějšího příkazu,
spustitelného programu nebo dávkového souboru.
```

znamená to, že systém nezná cestu k souboru javac. Lze ji nastavit pomocí proměnné PATH. Postup nastavení se liší pro jednotlivé OS. Pro operační systém WindoswXP tak můžeme učinit v menu Start/Nastavení/Ovládací Panely/Systém/Upřesnit/Proměnné prostředí. Do proměnné PATH přidáme cestu k příslušnému adresáři. Pokud máme jinou verzi JDK popř. je JDK nainstalováno v jiné složce, upravíme cestu dle potřeby.



Obrázek 2.1: Konfigurace proměnné PATH.

Nechceme -li konfigurovat proměnnou PATH, můžeme cestu k souboru javac specifikovat v příkazové řádce.

```
C:\Dokumenty> C:\Progra~1\Java\jdk1.6.0\bin\javac Pokus.java
```

Pokud je JDK správně nainstalováno, vznikne ve stejné složce soubor s názvem `Pokus.class` obsahující bajtový kód. Jestliže se objeví výpis:

```
javac: Invalid argument: Pokus
```

pak jsme zřejmě zapomněli zadat koncovku java.

Změňte název třídy z `Pokus` na `pokus` a pokuste se program přeložit. Podaří se Vám to?

Spuštění programu. Spuštění programu provedeme příkazem:

```
javac Pokus
```

popř. se specifikací cesty:

```
C:\Progra~1\Java\jdk1.6.0\bin\java Pokus
```

Tímto způsobem voláme interpretr jazyku Java a předáváme mu jako parametr soubor `Pokus.class`. V příkazové řádce však *nesmíme* uvést příponu `.class`, program si jí doplní sám. Opět musíme dodržovat velká a malá písmena. Jestliže nemáme konfigurovanou proměnnou PATH, pracovní adresář předpokládáme totožný s adresářem, ve kterém jsou uloženy zdrojový i přeložený soubor. Pokud jsme byli úspěšní, v textové konzoli se objeví výpis:

```
Tohle je muj
prvni program v Jave
```

Jestliže se objeví podobný výpis:

```
c:\Program Files\Java\jdk1.6.0\bin>javac Pokus.java
Pokus.java:5: cannot find symbol
symbol  : class Strings
location: class Pokus
   public static void main(Strings [] args)
                           ^
Pokus.java:9: cannot find symbol
symbol  : method pritln(java.lang.String)
location: class java.io.PrintStream
     System.out.pritln("Tohle je muj");
                ^
2 errors
```

pak jsme zřejmě zdrojový kód špatně opsali a došlo k syntaktické chybě. V tomto případě bylo místo správného String použito Strings a místo println použito pritln.

Poznámka: Zadávání úplné cesty k souborům se zdrojovým kódem nemusí být vždy pohodlné. Je proto možné nastavit systémovou proměnnou `CLASSPATH` tak, aby obsahovala cesty k příslušným adresářům. Postup nastavení se liší pro jednotlivé OS. Pro operační systém Windosw XP tak lze učinit v menu Start/Nastavení/ Ovládací Panely/Systém/Upřesnit/Proměnné prostředí, je analogický nastavení proměnné PATH.

Program rozebereme podrobně na jiném místě, všimněme si analogie s programovacím jazykem C++. Rozdílem je fakt, že každý "výkonný kód" musí být v Javě umístěn *uvnitř* třídy, která jej obaluje.

2.1 Tvorba dokumentace

Java obsahuje elegantní nástroj pro vytváření vlastní dokumentace. S jeho pomocí můžeme automaticky generovat dokumentaci k jednotlivým *třídám*, *proměnným* či *metodám*. Výsledná dokumentace je ve formátu HTML souborů.

Komentáře. Podívejme se, jakým způsobem jsou v Javě implementovány komentáře. Jedná se o analogii s jazykem C++.

```
// Toto je jednoradkovy komentar
/* Toto je
viceradkovy
komentar */
```

Kromě těchto komentářů existují i tzv. *dokumentační komentáře*. Ty jsou uvozeny znaky `/**` a zakončeny znaky `*/`. Na počátku každého řádku se nachází hvězdičky, které jsou však ignorovány.

```
/** Toto je
* dokumentacni viceradkovy
* komentar */
```

Z těchto dokumentačních komentářů mohou být vygenerovány automaticky soubory s dokumentací. Komentáře však musí být umístěny bezprostředně před definicí třídy, proměnné či metody, v opačném případě budou ignorovány.

```
/** Vzorova trida pokus */
public class Pokus
  {
    /** Promenna a */
    private int a;

    /** Metoda main, ktera napise dve radky */
    public static void main(String []args)
    {
     System.out.println("Tohle je muj");
     System.out.println("prvni program v Jave");
    }
  }
```

2.1.1 Javadoc

Pro vytváření dokumentace slouží nástroj `javadoc`, který se nachází v JDK 1.6 v následující složce `C:\Progra~1\Java\jdk1.6.0\bin\`. resp `/etc/lib/ jvm/java-6-sun-1.6.0.00/bin.` Tvorbu dokumentace spustíme, za předpokladu, že jsme provedli nastavení proměnné ClASS, příkazem

```
javadoc Pokus.java
```

V opačném případě je nutno zadat celou cestu. Stejně jako při překladu musíme uvést jméno souboru včetně koncovky. Ve složce vznikne ještě spousta dalších souborů, výsledkem je soubor s názvem Pokus.html, který je umístěn ve stejné složce jako soubor se zdrojovým kódem. Do dokumentačních komentářů lze vkládat některé speciální znaky. Uveďme stručně některé nejčastěji používané.

HTML tagy umožňují do dokumentačních komentářů vkládat kód ve formátu HTML. Cílem je zlepšené formátování kódu pomocí kurzívy či tučného řezu písma. Tag pro tučný řez písma se vkládá do komentáře posloupností znaků `<B>`, mezi nimi následuje zvýrazněný text, ukončuje se posloupnost znaků `</B>`. Podobným způsobem je možno vytvořit i skloněný text, použijeme k tomu tagy `<I> a </I>`.Podívejme se na následující kód, kdy text "**ktera napise**" bude vysázen tučně.

```
/** Metoda main, <B>ktera napise</B> dve radky */
```

Speciální příkazy jdou do dokumentačních komentářů vkládat zadáním znaku @. Příkaz `@see` slouží k vytváření odkazů na dokumentaci jiných tříd, příkaz `@version` informaci o verzi programu, `@author` informace o autorovi.

```
/** Dokumentace k programu
* @see Pokus //odkaz na nazev tridy
* @version 2.0 //informace o verzi
* @author Novak //informace o autorovi.
*/
```

2.2 Balíčky v Javě

Vzhledem k tomu, že jména souborů představují názvy tříd, není možné mít v jedné složce stejně pojmenované třídy provádějící např. různou činnost. Pokud chceme pojmenovat dvě různé třídy stejně, Java umožňuje

využít tzv. balíčky (packages). Balíčky představují obdobu jmenných prostorů (namespace) v C++. Balíček je fyzicky uložen v adresáři. Může obsahovat i podbalíčky, které se budou nacházet v podadresářích adresáře.

Jméno balíčku je tvořeno cestou k souborům se třídami v něm obsaženým. Nezapisuje se však pomocí zpětných lomítek, adresáře jsou odděleny tečkou. Máme-li v adresáři `Pokus` balíček s názvem `Balik`, ve kterém se nachází soubor `Pokus.class`, lze cestu k tomuto souboru zapsat ve tvaru

```
Balik.Pokus
```

Při návrhu Javy se vytvořil systém pojmenování balíčků takový, aby byl v rámci celého světa jednoznačný. Název balíčku by měl vycházet z názvu internetových domén "v opačném" pořadí. Balík `Balik` nacházející se v doméně `moje.cz` bude mít dle této konvence název `cz.moje.balik`. V praxi se však tento přístup ne vždy dodržuje.

2.2.1 Práce s balíčky

Pokud chceme nějakou třídu přidat do balíčku, uvedeme tuto informaci do prvního řádku zdrojového kódu příkazem

```
package nazev_baliku;
```

Pokud bychom chtěli námi vytvořenou třídu `Pokus` přidat do balíčku `balik`, vypadal by zápis zdrojového kódu takto:

```
package balik;
public class Pokus
  {
   public static void main(String [] args)
    {
     System.out.println("Tohle je muj");
     System.out.println("prvni program v Jave");
    }
  }
```

Kompilaci provedeme stejným způsobem jako v předchozím případě. Soubor `Pokus.class` umístíme do adresáře `balik`. Pokud tento soubor chceme spustit, učiníme tak jménem balíku a jménem třídy: `balik.Pokus`.

```
C:\Progra~1\Java\jdk1.6.0\bin\java balik.Pokus
```

Uvedený postup nazýváme **kvalifikací třídy** jménem balíku. Pokud bychom chtěli tento balíček použít v jiném programu, museli bychom příslušnou třídu zapsat s uvedením kvalifikace balíku. Podívejme se na následující příklad. Vytvořme program `Pokus2`, ve kterém budeme chtít použít instanci třídy `Pokus`. Na třídu `Pokus` se budeme muset odkázat za použití kvalifikace.

```
public class Pokus2
  {
   public static void main(String [] args)
    {
     balik.Pokus objekt=new balik.Pokus();
    }
  }
```

Tento způsob práce je nepohodlný, název třídy s kvalifikací může být někdy značně dlouhý, zvláště u složitějších struktur balíčků. Pomocí příkazu `import` s uvedením názvu balíčku ho můžeme "připojit" a při práci s ním vynechat kvalifikaci.

```
import balik;
public class Pokus2
  {
   public static void main(String [] args)
    {
     Pokus objekt=new Pokus();
    }
  }
```

Obsahuje-li balíček nějaké podbalíčky, lze pro jejich zpřístupnění použít zástupný symbol `*`. V následujícím příkladu má balíček s názvem `balik` dva podbalíčky: `balicek1` a `balicek2`. Lze psát

```
import balik.balicek1;
import balik.balicek2;
```

Nebo využít zástupný znak `*` a psát

```
import balik.*;
```

Což znamená v přeneseném smyslu: připoj všechny podbalíčky balíčku `balik`. Práce s balíčky je typická u větších projektů, pomáhá vytvořit přehlednou hierearchickou strukturu zdrojového kódu.

V této kapitole jsme se stručně seznámili s vlastnostmi jazyku Java, ukázali si způsob, jakým lze psát jednoduché programy, vytvářet dokumentaci k nim a vysvětlili si základní operace s balíčky. V následujících kapitolách se s jazykem Java seznámíme podrobněji.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

Kapitola 3

Datové typy a proměnné v Javě

Java je jazyk relativně jednoduchý, jeho jádro obsahuje cca. 40 základních příkazů. Velmi rozsáhlé je však API jazyka Java, které skrývá stovky příkazů. I přes výše zmíněný fakt je struktura jazyka logická a umožní začátečníkovi poměrně rychlou orientaci v problematice a snadné zvládnutí základů programování. Java pracuje se základními (tj. primitivními) datovými typy a s objektovými datovými typy. Připomeňme, že označení identifikátorů základních datových typů začíná malým písmenem, označení identifikátorů objektových datových typů velkým písmenem.

3.1 Základní datové typy

V programovacím jazyce Java existuje několik druhů základních datových typů: celočíselné, logické, reálné a znakové. Toto členění odpovídá ostatním programovacím jazykům, nenalezneme zde tedy žádné "překvapení". Základní datové typy nejsou v Javě považovány za objekty, není proto s nimi možno jako s objekty zacházet. U každého jednoduchého datového typu je definována jeho velikost v bajtech (tj. není určována operačním systémem), tím pádem se program stává univerzálně přenositelný napříč různými operačními systémy.

3.1.1 Celočíselné datové typy

V Javě existují čtyři základní celočíselné datové typy. Všechny jsou se znaménky, neexistuje tedy u nich varianta bezznaménka, tj. unsigned. Jejich přehled je uveden v tab. 4.1.

Datový typ	Velikost	Rozsah
`byte`	8b	-128;127
`short`	16b	-32768;32767
`int`	32b	-2E09;2E09
`long`	64b	-9E18;9E18

Tabulka 3.1: Přehled celočíselných datových typů v Javě.

Chceme -li zapisovat hodnoty proměnných a konstant, můžeme tak učinit třemi způsoby:

- V desítkové soustavě (znaky 0-9): 9234
- V osmičkové soustavě (znaky 0-7): 0156

Escape sekvence	UNICODE	Význam
\n	\u000A	Nová řádka
\r	\u000D	Návrat na začátek řádku
\t	\u0009	Tabulátor
\\	\u005C	Zpětné lomítko
\'	\u002C	Apostrof
\"	\u0022	Uvozovky

Tabulka 3.2: Přehled escape sekvencí a UNICODE kódů některých důležitých znaků.

- V šestnáctkové soustavě (znaky 0-9, a-f): Začíná znaky 0x a pokračuje povolenými znaky, 0x26AF.

Nutno vzít v úvahu fakt, že konstanty jsou standardně datového typu `int`. Příkaz

```
a=123456;
```

přiřadí proměnné a hodnotu `int`. Chceme -li přiřadit této proměnné hodnotu typu `long`, musíme uvést za číselnou hodnotou symbol L.

```
a=12345678910L;
```

Pokud nevíme, jaký datový typ použít, a nemáme nějaké speciální požadavky, zpravidla používáme typ `int`.

3.1.2 Znakový typ

Java disponuje datovým typem `char`, který se používá pro práci se znaky. Využívá kódování a sady znaků UNICODE, velikost je 16 bitů. Podmnožinou UNICODE je znaková sada ASCII, obsahuje prvních 127 znaků. Znakové konstanty se zapisují několika způsoby:

- jedním znakem uzavřeným do apostrofů: `'A'`
- prostřednictvím escape sekvencí (speciální znaky pro skok na nový řádek, tabulátor, atd.): `'\n'`
- posloupností znaků \'uXXXX', kde XXXX představuje kód znaku v kódování UNICODE: `'\u000A'`

Stručný přehled důležitých znaků a jejich UNICODE kódů nalezneme v tabulce 3.2. Úplný přehled znakové sady UNICODE lze nalézt na stránce http://unicode.org.

Používání escape sekvencí a UNICODE kódů se odlišuje. UNICODE znaky se mohou vyskytovat kdekoliv v programu, např. jako součást identifikátorů, escape sekvence mohou být pouze součástí znakových a řetězcových konstant (viz dále).

```
double ko'\u010D'ka;//kočka
double ko'\99'ka//kocka, nelze
```

Typ char má vlastnosti celočíselného datového typu bez znaménka, jeho hodnota odpovídá kódu příslušného znaku. Na jednotlivé znaky se můžeme dívat jako na celá čísla, lze na ně aplikovat stejné operátory jako na datový typ `int`.

Datový typ	Velikost	Rozsah
`float`	32b	-1.4E45;3.4E48
`double`	64b	-4.9E-324,1.7E308;

Tabulka 3.3: Přehled reálných datových typů.

3.1.3 Řetězcové konstanty

Pro tvorbu řetězcových konstant platí stejné zákonitosti jako pro tvorbu znakových konstant. Řetězcové konstanty mohou být na rozdíl od znakové konstanty tvořeny více než jedním znakem. Nejsou zapisovány do apostrofů, ale do uvozovek.

```
System.out.println("Ahoj");
System.out.println("Prvn\u00ED program\n v jazyce Java");
```

Java, na rozdíl od C++, umožňuje sčítat řetězcové konstanty prostřednictvím operátoru +.

```
"prvni slovo"+"druhe slovo";
```

3.1.4 Logický datový typ

Logický datový typ `boolean` nabývá dvou hodnot: pravda resp. nepravda, tj. `true` resp. `false`. Obě hodnoty lze vyjádřit prostřednictvím celočíselných hodnot: 1 resp. 0. Uplatňuje se zejména při konstrukci logických podmínek.

```
boolean vysledek=false;
```

3.2 Reálné datové typy

Reálná čísla jsou představována desetinnými čísly. Java disponuje dvěma datovými typy pro práci s reálnými čísly: `float` a `double`, viz tab. 3.3. Všimněme si, že horní a dolní hranice intervalů jsou nesymetrické. Reálné konstanty lze zapsat různými způsoby:

- Zápisem s pevnou řádovou čárkou. Jedná se o stejný způsob zápisu, jakým zapisujeme čísla na papír. Jako oddělovač používáme desetinnou tečku.

  ```
  123.456
  10.0
  ```

- Zápisem v semilogaritmickém tvaru. Tímto způsobem běžně zapisujeme čísla "příliš malá" nebo "příliš velká": $0.1 * 10^{10}$ nebo $0.1 * 10^{-10}$.

  ```
  0.1e10
  0.1e-10
  ```

Reálná konstanta je standardně datového typu `double`. Chceme-li ji změnit na `float`, musíme za poslední číslici umístit znak `F`.

```
a=0.1e10F;
```

Maximální a minimální hodnota. Maximální a minimální hodnoty pro celočíselné i reálné datové typy lze získat prostřednictvím konstant `MIN_VALUE` a `MAX_VALUE`. Před konstantu umístíme označení datového typu v objektové variantě. Syntaxe vypadá takto:

```
Integer.MAX_VALUE;
Double.MIN_VALUE;
```

Na závěr uveďme dvě speciální hodnoty, které mohou vzniknout při "nevhodném" dělení nějakého čísla jiným číslem. Dělíme -li kladné resp. záporné číslo 0, výsledkem operace je nekonečno, tj. hodnota `POSITIVE_INFINITY` resp. `NEGATIVE_INFINITY`. Dělíme -li 0 nulou, tj 0/0, výsledkem operace je hodnota `NaN`, tj. not a number.

3.3 Deklarace proměnných

Deklarace proměnné představuje krok, při kterém vytvoříme proměnnou zvoleného jména některého z výše uvedených datových typů a přiřadíme ji (tj. inicializujeme ji) na nějakou počáteční hodnotu. Tento typ inicializace označujeme jako *explicitní*.

Jestliže neuvedeme žádnou inicializační hodnotu, je proměnná inicializována *implicitně*, tj. na výchozí hodnotu pro zvolený datový typ. Připomeňme, že implicitní inicializace proběhne pouze v případě, že proměnná představuje datovou položku třídy.

Pokud by se jednalo o proměnnou, která nepředstavuje datovou položku třídy, a nedošlo by k její explicitní inicializaci před prvním použitím, upozorní nás Java na tento nedostatek. Odstraňuje tak jednou z velkých nevýhod některých jazyků, tj. možnost práce s neinicializovanými proměnnými.

```
int a;
double b, c, d;
```

Uveďme stručně některé zásady či vlastnosti související s deklaracemi proměnných:

- V Javě neexistují jako v C++ globální proměnné. Není možné vytvořit proměnnou, která by byla dostupná v celém programu. Deklarace totiž nesmí být umístěna vně třídy, jak je to v C++ běžné např. ve spojitosti s klíčovým slovem `extern`.
- V Javě nerozlišujeme pojmy definice proměnné a deklarace proměnné. nelze totiž vytvořit a použít proměnnou bez její implicitní či explicitní inicializace.
- Je vhodné dodržovat zásadu: deklarace proměnných by z důvodu přehlednosti měly být umístěny na začátku bloku.

Práce s konstantami. Java používá jiný přístup při deklaraci konstant. Konstanty jsou deklarovány s klíčovým slovem `final`. Hodnotu konstanty již nelze dále měnit.

```
final int CISLO=10;
```

Příkaz

```
CISLO=15; //Chyba
```

způsobí vyvolání chyby, program nebude možno přeložit. Platí zásada, že jména konstant by měla být tvořena pouze velkými písmeny. Pokud chceme použít konstantu vně třídy, ve které "vznikly", je nutno jejich deklaraci provést vně všech metod, nastavit ji jako veřejnou, tj. za použití klíčového slova `public`. Jelikož se bude jednat o proměnnou třídy, nezapomeňme, že musí být společná pro všechny instance, je nutno ji deklarovat jako statickou za použití klíčového slova `static`.

```
public class Neco
{
 public static final int CISLO=10; //staticka promenna neco
}
```

3.4 Přiřazování a přetypování

Přiřazovací příkaz je používán při práci s proměnnými i konstantami. Operátorem přiřazení je znak `=`, operátor `==` představuje relační operátor "`je rovno`". Přiřazovací příkaz umožňuje přiřadit hodnotu nějakého výrazu jiné proměnné nacházející se vlevo od operátoru `=`. Výraz, který do proměnné přiřazujeme, musí mít nějakou hodnotu.

```
a=10.0;
b=c+15*f;
znak='A';
```

Přiřazovací příkaz může být používán i při tvorbě logických podmínek. Začátečníka by neměla zmást následující konstrukce

```
if (a==(b=10));
```

představující dotaz, zda hodnota proměnné `a` je rovna hodnotě proměnné `b`, která je inicializována na 10.

Přetypování. Při přiřazovacích operacích dochází často ke konverzím mezi různými datovými typy. Tyto operace nazýváme přetypováním. Výsledkem přiřazení je změna datového typu proměnné. Existují dva typy přetypování:

- *Implicitní konverze*

  Vzniká při přetypování proměnné s nižším rozsahem na proměnnou s vyšším rozsahem. Probíhá automaticky bez nutnosti zápisu jakéhokoliv programového kódu. Nedochází při ní ke ztrátě informací. Pro výše uvedené datové typy bývá prováděna v následujícím pořadí: `byte->short->int->long->float->double`.

  ```
  float a=50;
  double b=30;
  b=a;//implicitni konverze
  ```

- *Explicitní konverze*

  Vzniká při přetypování proměnné s vyšším rozsahem na proměnnou s nižším rozsahem. Konverze neproběhne automaticky (může při ní dojít k snížení přesnosti dat), je tedy nutno vynutit ji uvedením výsledného datového typu. Při explicitní konverzi dochází zpravidla ke ztrátě informací. Bývá prováděna v následujícím pořadí: `double->float->long->int->short->byte.`

```
a=b;//ztrata presnosti, nelze provest
a=(float)b;//explicitni konverze
```

Přetypování lze používat u OOP také při práci s rodičovskou a odvozenou třídou. Objekt rodičovské třídy označíme rodič, objekt odvozené třídy potomek. Potomek bývá zaměřen úžeji než rodič. Existují dvě možnosti přetypování:

- Přetypování z potomka na rodiče: implicitní konverze
- Přetypování z rodiče na potomka: explicitní konverze

Podrobněji se s nimi seznámíme v kapitole věnované objektově orientovanému programování.

Smíšené konverze. Podívejme se na situace, kdy do aritmetických operací vstupují operandy smíšeného typu, hovoříme pak o tzv. smíšených výrazech. Platí následující zákonitosti:

1. Pokud je jeden z operandů typu double, bude výsledkem operace datový typ double
2. Pokud je jeden z operandů typu float, bude výsledkem operace datový typ float
3. Pokud je jeden z operandů typu long, bude výsledkem operace datový typ long

Jejich cílem je realizace aritmetických operací tak, aby při nich došlo k co nejmenší ztrátě přesnosti. Pozor musíme dát u výrazů obsahujících zlomky, pokud čitatel i jmenovatel představují celočíselné hodnoty. Výsledek operace bude opět datového typu `int`.

```
int a=5;//int
int b=10;//int
int c=a/b;//vysledek=0
```

Výše uvedený zápis je z formálního hlediska špatným, neměl by být nikdy použit. Správný je následující zápis, při kterém vznikne smíšený výraz:

```
int a=5.0;//double
int b=10;//int
int c=a/b;//vysledek=0.5;
```

Objektové datové typy a přiřazování. Uveďme pro úplnost, že i když vytvoříme datový typ jako objektový, tak nemá rysy skutečného objektu, např. operace přiřazení je prováděna nikoliv odkazem, jak bychom očekávali, ale stejně jako u primitivního datového typu hodnotou. Java tedy není stroprocentně objektový jazyk, jak je patrné.

Konverze základních datových typů, řetězců a znaků. Vzájemné konverze primitivních datových typů a řetězců jsou používány velmi často. Chceme -li konvertovat některý ze základních datových typů na datový typ `String`, použijeme statickou metodu třídy `String` s názvem `valueOf()`.

```
int a=10;
boolean b=true;
double c=15.0;
String retezec=String.valueOf(a);
String retezec=String.valueOf(b);
String retezec=String.valueOf(c);
```

Konverzi základního datového typu na řetězec lze provést i "fintou" se sčítáním prázdného řetězce a proměnné:

```
String retezec=""+a;
```

Opačný postup, při kterém konvertujeme řetězec na některý ze základních datových typů, lze provést dvěma způsoby:

- Starší přístup, který je poněkud těžkopádný. Je daní za ne 100% "objektovost" jazyka Java. V prvním kroku provedeme konverzi řetězce na objekt příslušné třídy za použití metody `valueof()`. Tento objekt je nutno v následujícím kroku převést na primitivní datový typ prostřednictvím některé z trojice metod: `intValue()`, `doubleValue()`, `booleanValue()`;

  ```
  String retezec="1234";
  int a=String.valueOf(retezec).intValue();
  double b=String.valueOf(retezec).doubleValue();
  bool c=String.valueOf(retezec).booleanValue();
  ```

- Novější přístup vycházející z JDK prostřednictvím metod `parseLong(), parseDouble(), parseFloat()`.

  ```
  String retezec="1234";
  String retezec2="true";
  int a=(int)Long.parseLong(retezec);
  ```

Chceme -li přetypovat znakovou konstantu na některý z primitivních datových typů, použijeme metodu `digit()`.

```
char znak='A';
int cislo=Character.digit(znak);
```

Výše uvedené konverzní postupy použijeme při práci se standardním vstupem a výstupem.

3.5 Operátory v Javě

Kromě operátoru přiřazení, se kterým jsme se již seznámili, je v Javě používána řada dalších typů operátorů.

Aritmetické operátory. Aritmetické operátory jejich stručný popis jsou zobrazeny v tabulce 3.4. Slouží k realizaci základních aritmetických operacích. Při jejich vyhodnocování se postupuje ve směru zleva do prava dle priority: nejprve operace násobení/dělení/celočíselné dělení, poté se vyhodnotí zbývající operace.

Operátor	Popis
+	Sčítání
-	Odečítání
*	Násobení
/	Dělení
%	Zbytek po celočíselném dělení.

Tabulka 3.4: Přehled aritmetických operátorů.

Operátor	Úplný zápis
a+=5	a=a+5
a-=5	a=a-5
a*=5	a=a*5
a/=5	a=a/5

Tabulka 3.5: Přehled operátorů inkrementace a dekrementace.

Chceme -li změnu priority vyhodnocování provést uměle, můžeme tak použít závorky. Počet pravých a levých závorek by měl být stejný.

```
int a=3+3*3;//a=12
int a=(3+3)*3;//18
```

Operátory přiřazení. Operátory přiřazení představují kombinaci přiřazení a aritmetických operátorů. Tyto operátory umožňují zkrácený zápis běžných aritmetických operací.

Operátory inkrementace a dekrementace. Operátory inkrementace a dekrementace umožňují zvyšování či snižování hodnoty proměnné o 1. Jedná se o "zdvojené" operátory ++ resp. −−. Někteří zlí jazykové tvrdí, že Java=C+ + −−, tj. že Java je to samé to samé C++ :-).

Existují dva typy inkrementace resp. dekrementace:

- *předcházející inkrementace/dekrementace.*

  Při ní jsou operátory inkrementace/dekrementace umístěny před proměnnou. V tomto případě se nejprve provede přiřazovací příkaz a až následně inkrementace/dekrementace proměnných.

  ```
  int a=5, b=5;
  int c=++a;//a=6, c=5;
  int d=--b;//b=4, d=5;
  ```

- *následná inkrementace/dekrementace.*

  Při ní jsou operátory inkrementace/dekrementace umístěny za proměnnou. V tomto případě se nejprve provede inkrementace/dekrementace a až následně přiřazovací příkaz.

  ```
  int a=5, b=5;
  int c=a++;//a=6, c=6;
  int d=b--;//b=4, d=4;
  ```

Relační operátory. Relační operátory se používají při konstruování logických podmínek. Přehled relačních operátorů je uveden v tab. 3.5.

Operátor	Popis
==	Rovná se
!=	Nerovná se
&&	Logický součin
||	Logický součet
!	Negace
<	Menší
>	Větší
<=	Menší nebo rovno
>=	Větší nebo rovno

Tabulka 3.6: Přehled relačních operátorů.

3.6 Matematické metody

Java disponuje jak základními, tak pokročilými matematickými funkcemi. Ty jsou umístěny ve třídě `Math`, která se nachází v balíku `java.lang`. Představují jednotlivé metody třídy Math, proto k nim můžeme přistupovat za použití syntaxe `Math.Metoda(parametr)`. Metoda může být volána s formálními parametry, které se zapisují do závorek. Výsledkem výpočtu je vždy pouze jedna hodnota.

```
double Ro=180/Math.PI;
double vysledek=Math.sin(45/Ro);
```

Pro úplnost uveďme, že stejně jako v ostatních jazycích, jsou vstupní hodnoty trigonometrických funkcí uváděny v *radiánech*. Chceme -li v našem programu využívat matematické metody a nechceme -li je kvalifikovat názvem balíčku, musíme importovat balíček `Math`.

```
import java.math.*;
```

Některé metody jsou přetížené, existuje více metod se stejným názvem, jejich vstupní parametry jsou však různé: float, double, int, long. Pokud není vstupní hodnota požadovaného datového typu, je provedena její konverze (vynucená konverze), pokud to je možné. Většina z matematických metod poskytuje výsledek typu double. Přehled nejčastěji používaných metod nalezneme v tabulce 3.7.

3.7 Práce se standardním formátovaným vstupem a výstupem

Při běhu programu musí existovat možnost zadávání vstupních dat (vstupní parametry pro výpočet) i možnost práce s výstupními daty (výsledky výpočtů). Hovoříme o terminálovém vstupu a výstupu. Upozorněme, že standardní formátovaný vstup a výstup nejsou v Javě na rozdíl od jazyka C++ určeny pro vážnou práci, ale pouze pro výukové účely. Java totiž disponuje grafickým rozhraním umožňujícím provádět vstup/výstup údajů s využitím vizuálních komponent, např. textfieldů či editboxů, seznamů. Práce s terminálovým vstupem je v Javě poněkud komplikovanější než v jazyce C++.

Nástroje pro práci se standardním vstupem a výstupem se nacházejí v balíku `java.io`. Chceme -li pracovat se standardním vstupem a výstupem, použijeme

```
import java.io.*;
```

Pro práci se standardním formátovaným vstupem v Javě existují objekty `System.in` a `System.out`. Objekt `System.in` realizuje standardní vstup, nejčastěji čtením hodnot z klávesnice. `System.out` je objekt, který se používá pro zobrazování výstupních údajů na monitoru.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

Funkce	Mat. zápis	Vstupní hodnoty	Výstupní hodnoty
`Math.abs(x)`	$\|x\|$	int, long, float, double	int, long, float, double
`Math.acos(x)`	arccos(x)	double	double
`Math.asin(x)`	arcsin(x)	double	double
`Math.atan(x)`	arctg(x)	double	double
`Math.atan2(y,x)`	arctg(y/x)	double	double
`Math.ceil(x)`	$int(x + 0.5)$	double	double
`Math.cos(x)`	cos(x)	double	double
`Math.exp(x)`	e^x	double	double
`Math.floor(x)`	int(x-0.5)	double	double
`Math.log(x)`	ln(x)	double	double
`Math.max(x,y)`	max(x,y)	int, float, double	int, float, double
`Math.min(x,y)`	min(x,y)	int, float, double	int, float, double
`Math.random(x)`		-	double
`Math.pow(x,y)`	x^y	double	double
`Math.round(x,y)`	int(x+0.5)	double, float	double, float
`Math.sin(x)`	sin(x)	double	double
`Math.sqrt(x)`	$\sqrt{x}$	double	double
`Math.tan(x)`	tg(x)	double	double

Tabulka 3.7: Přehled vybraných matematických funkcí.

3.7.1 Formátovaný výstup

Pro formátovaný výstup je používán objekt `System.out`. S použitím metody `print()` můžeme provést vytisknutí řetězce na aktuální pozici v řádku. Pokud se na výstup dostane hodnota jiného datového typu než řetězec, je provedena vynucená konverze na řetězec.

```
int a=10;
int b=20;
System.out.print(a);
System.out.print(b);
System.out.print("Ahoj");
```

Na výstupu se objeví: `1020Ahoj`. Na rozdíl od jazyka C++ není možno hodnoty na výstupu nijak formátovat, tj. např. stanovit počet desetinných míst. Chceme -li zřetězit hodnoty na výstupu, tj. vytisknout v jednom kroku více údajů, použijeme operátor +. Můžeme vzájemně sčítat "hrušky s jablky", tj. jak řetězce, tak i proměnné primitivních datových typů. Použijeme

```
System.out.print ("a="+a+" b="+b+" Ahoj");
```

obdržíme na výstupu:`a=10 b=20 Ahoj`. V řetězci se mohou vyskytnout i escape sekvence, uveďme dvě nejčastěji používané:

- znak `'\t'` (znaková konstanta) resp. "`\t`" (část řetězce): tabulátor.
- znak `'\n'` (znaková konstanta) resp. "`\n`" (část řetězce): přechod na novou řádku.

Příkazy

```
System.out.print ("a="+a+"\n"+"b="+b+"\n Ahoj");
System.out.print ("jak"+"\t"+"se"+"\tmate");
```

vytisknou na obrazovku

```
a=10
b=20
Ahoj
jak   se   mate
```

Stejného efektu dosáhneme za použití metody println().

```
System.out.println("a="+a);
System.out.println("b"+b);
System.out.println("jak"+"\t"+"se"+"\tmate");
```

3.7.2 Standardní formátovaný vstup

Práce s formátovaným vstupem v Javě je poněkud složitější než v jazyce C++. Je používán objekt `System.in`. Při zadávání znaků z klávesnice jsou jednotlivé znaky interpretovány jako bajty zakončené ukončovacím znakem `'\n'`. Je vhodné je proto načíst do pole bajtů. V následujícím příkladu si ukážeme načtení dat z formátovaného vstupu a jejich konverze na typ `int`.

```
byte [] pole=new byte [30];
System.in.read(pole);
```

Bajty následně převedeme na řetězec typu String

```
String retezec=new String(pole);
```

a ořízneme ukončovací znaky

```
retezec=retezec.trim();
```

Následně provedeme konverzi `String` na typ `int`. Nezapomeňme na explicitní přetypování.

```
int cislo=(int)Long.parseLong(retezec);
```

Obdobným způsobem bychom postupovali i při konverzi na jiné datové typy:

```
double cislo=Double.parseDouble(retezec);
```

Tento kód je vhodné obalit do konstrukce try/catch pro odchycení případné výjimky vzniklé např, tím, že uživatel na vstupu zadá textový řetězec obsahující jeden nebo více znaků, které nejsou číslicemi. S tímto přístupem se setkáme až v části věnované odchycování výjimek.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

3.8 Příkazy v Javě

Příkazy představují základní stavební jednotky programů. S jejich využitím přepisujeme algoritmy do prostředí příslušného formálního jazyka (tj. programovacího jazyka). Příkazy jsou zpravidla prováděny v pořadí, v jakém byly programátorem zapsány. Příkazy v Javě jsou velmi podobné příkazům v jazyce C++. V Javě rozeznáváme dva typy příkazů: jednoduchý příkaz a složený příkaz.

Jednoduchý příkaz je ukončen středníkem

```
System.out.println ("Ahoj"); //jednoduchý příkaz
```

Platí zásada, že na každém řádku by měl být umístěn pouze jeden příkaz, program by se v opačném případě stal nepřehledným.

Složený příkaz je uzavřen do složených závorek a vytváří blok. Představuje skupinu příkazů, která se z hlediska syntaxe Javy chová jako jediný příkaz. Za blokem nepíšeme středník.

```
if (a<b)
  {
   a++;
   System.out.println(a);
  }
```

Blok může být prázdný, nemusí se v něm vyskytovat žádný příkaz. Taková konstrukce neprovádí žádnou činnost, není však syntaktickou chybou. Označujeme ji jako tzv. *prázdný příkaz.* Pro úplnost uveďme, že prázdný příkaz vznikne i použitím samotného středníku.

```
;
{}
```

Uvnitř bloku může být definován další blok, hovoříme o tzv. *vnořeném bloku.* Připomeňme tvrzení z úvodní kapitoly, které říká: pokud definujeme ve vnořeném bloku proměnnou, nelze ji používat mimo vnořený blok.

```
if (a<b)
  {
   a++;
   if (b==0)
     {
       b--;
       System.out.println(a);
     }
  }
```

V následujících kapitolách se budeme seznamovat s jednotlivými příkazy jazyku Java.

3.9 Pole

Pole představuje skupinu proměnných stejného datového typu, se kterými může zacházet jako s celkem. Pole je jeden z dvou neprimitivních datových typů, druhý je objekt. Jednotlivé proměnné nazýváme prvky pole. Lze k nim přistupovat za použití jména pole a indexu (ten je uváděn v hranatých závorkách).

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

Index představuje pořadové číslo prvku pole od jeho počátku. Označíme -li celkový počet prvků pole n, úvodní prvek pole má index 0, poslední prvek pole má index n-1. Index musí vždy představovat kladné celé číslo.

Práce s polem je v Javě bezpečnější než v C++, kde pole představují pouze bloky paměti. V Javě je automaticky prováděna kontrola překročení indexu pole, k řadě chyb těžko odhalitelných v C/C++ zde nedochází. Dynamicky vytvořené pole nemusíme rušit manuálně, pokud na něj neexistuje platný odkaz, je zrušeno prostřednictvím garbage collectoru.

Práce s polem. S polem se v Javě pracuje prostřednictvím odkazu, tj. podobně jako s objekty. Nejprve je nutno deklarovat odkaz na pole, následně se vytvoří pole samotné. Odkaz na obecné pole lze vytvořit následující konstrukcí: `datovy_typ [] identifikator`.

Odkaz na pole typu double

```
double [] pole;
```

V deklaraci se neuvádí počet prvků pole. Při deklaraci nedochází k přidělení paměti pro pole, jakákoliv operace s takovým polem skončí vyvoláním výjimky. Druhý krok představuje vytvoření vlastního pole znamenající přidělení potřebné paměti a nastavení odkazu tak, aby ukazoval na vytvořené pole. Budeme vytvářet pole o 10 prvcích

```
pole=new double[10];
```

Počet prvků pole je při jeho vytváření uváděn v hranatých závorkách. Oba kroky lze spojit do jednoho

```
double [] pole=new double[10];
```

Po vytvoření kompilátor zaručí, že prvky pole budou inicializovány na hodotu 0, u polí typu `boolean` na hodnotu `false`, u polí odkazů (tj. referencí) na hodnotu `null`. Z výše uvedeného postupu vidíme, že s polem se pracuje podobně jako s objektem.

Délka pole. Důležitou vlastností pole je jeho délka, tj. počet prvků. Lze ji určit prostřednictvím proměnné `length`. Výsledkem je kladná celočíselná hodnota.

```
pole.length;
```

Tuto hodnotu je vhodné použít při procházení prvků pole pomocí cyklu jako horní mez.

Inicializované pole. Existuje možnost vytvořit pole inicializované předem danými hodnotami, v takovém případě se nepoužívá konstrukce s `new`.

```
double [] pole={2.0, 4.4,17.85};
```

Počet prvků nemusíme uvádět, kompilátor si je spočítá "sám". S takto vytvořeným polem se pracuje stejně jako s jakýmkoliv jiným vytvořeným polem. Přístup k jednotlivým prvkům pole lze provádět přes index. Následující příkaz přiřadí proměnné cislo hodnotu druhého prvku pole.

```
double cislo=pole[1];
```

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

Vícerozměrné pole. Při složitějších matematických výpočtech používáme často dvojrozměrná či trojrozměrná pole, typickým příkladem jsou matice. Práce s vícerozměrným polem je analogií práce s jednorozměrným polem, pro přístup ke konkrétnímu prvku použijeme dvojici indexů.

```
double [][] pole2d;
pole2d=new double[10][10];
```

Počet řádků pole lze získat jako

```
pole2d.lenght;
```

počet sloupců pole jako

```
pole2d[i].length;
```

kde i představuje index libovolného řádku pole.

Vícerozměrné inicializované pole. Lze ho vytvořit stejným způsobem jako jednorozměrné inicializované pole, tj. bez použití konstrukce `new`. Následující konstrukce vytvoří dvourozměrné pole o třech řádcích a dvou sloupcích.

```
double [][] pole2d={{2.0, 4.4},{17.85, 0.3}, {10.9, 6.4}};
```

S poli zpravidla pracujeme za použití cyklů (viz dále), jednotlivé prvky lze efektivněji procházet.

3.9.1 Kopírování polí

Při kopírování dvou polí nelze použít přiřazovací příkaz, prováděli bychom pouze kopírování odkazů, nikoliv kopírování skutečného obsahu polí. Následující konstrukce je chybná.

```
int [] pole1={1,2,3};
int [] pole2=new int [3];
pole2=pole1;//zkopirovani odkazu, nikoliv pole!
```

Došlo ke zkopírování odkazu na `pole1` do `pole2`. Odkazy `pole1` a `pole2` ukazují fyzicky na stejné pole. Chceme -li provádět kopírování obsahu polí, přepíšeme konstrukci do tvaru

```
int [] pole1={1,2,3};
int [] pole2=new int [3];
for (int i=0;i<pole1.length;i++) pole2[i]=pole1[i];
```

V tomto případě provádíme kopírování prvek po prvku za použití cyklu. Existuje i příkaz `arraycopy`, který provádí stejnou činnost pohodlněji.

```
int [] pole1={1,2,3};
int [] pole2=new int [3];
System.arraycopy(pole1,0,pole2,0,pole.length);
```

Má celkem 5 parametrů. První představuje pole, jehož obsah chceme kopírovat, druhý pozici, od které kopírujeme, třetí pole, do kterého kopírujeme, čtvrtý pozici, od které chceme zkopírované prvky ukládat, pátý celkový počet zkopírovaných prvků.

Kapitola 4

Větvení programu a opakování

Příkazy pro větvení bývají často označovány jako tzv. řídící struktury. Umožňují provádět větvení programu a ovlivnit, které příkazy budou provedeny v závislosti na hodnotách logických výrazů (tj. podmínek). Příkazy pro opakování nazýváme iteračními příkazy, umožňují opakovaně provádět jeden nebo více příkazů. Patří k nejčastěji používaným konstrukcím.

4.1 Přehled příkazů pro větvení programu

4.1.1 Konstrukce if-else

Tento příkaz nalezneme prakticky ve všech programovacích jazycích. V Javě se vyskytuje ve dvou variantách, a to jako podmínka úplná nebo podmínka neúplná.

Neúplná podmínka. Neúplnou podmínku lze zapsat jedním z následujících způsobů:

```
if (vyraz)
  prikaz;     //jednoduchy prikaz
if (vyraz)
  {
    prikazy;  //blok
  }
```

První zápis představuje variantu s jednoduchým příkazem, druhý zápis variantu s blokem. Neúplnou podmínku lze interpretovat takto: Pokud výraz (booleovský) nabývá pravdivé hodnoty, provede se příkaz resp. blok příkazů. Zopakujme znovu, že blok nesmí být ukončen středníkem. V praxi platí zásada, že do bloku dáváme i jednoduchý příkaz, program se stává čitelnějším. Pokud je to možné, podmínky neuvádíme v negaci, ale v "kladné" formě.

Úplná podmínka. Úplná podmínka vznikne rozšířením neúplné podmínky o konstrukci `else`, která bude provedena, pokud podmínka nebude splněna. Úplnou podmínku lze zapsat jedním z následujících způsobů popř. je kombinovat:

```
if (vyraz)
  prikaz;     //jednoduchy prikaz
else prikaz;
if (vyraz)
  {
    prikazy;  //blok
  }           //nelze pouzit strednik
else
  {
    prikazy;
  }
```

V případě užití jednoduchého příkazu nesmíme před `else` zapomenout na středník, u bloku před `else` středník použít naopak nemůžeme. Lze vytvářet i složitější konstrukce, při kterých mohou být uvnitř podmínky vnořené další bloky obsahující podmínky. V takovém případě else odpovídá nejbližšímu nespárovanému if. Konstrukci

```
if (vyraz1) prikaz1
    if (vyraz2)prikaz3
        else prikaz4;
else prikaz2;
```

lze chápat jako

```
if (vyraz1)
  {
    prikaz1;
    if (vyraz2)
      {
       prikaz3;
      }
    else  //odpovida nejblizsimu if
      {
       prikaz4;
      }
  }
else
  {
   prikaz2;
  }
```

4.1.2 Operátor ?

Tento operátor nazýváme ternárním, má tři argumenty. Umožňuje zapsat úplnou podmínku stručnějším, avšak méně přehledným, způsobem. Jeho syntaxe je

```
Vyraz? Prikaz1: Prikaz2;
```

Je -li výraz vyhodnocen jako pravdivý, je proveden Prikaz1, v opačném případě Prikaz2. Podmínku

```
if (a<b) a++
  else b++;
```

lze tedy zapsat jako

```
a<b? a++: b++;
```

4.1.3 Příkaz switch

Příkaz `switch` představuje přepínač, který umožní větvení programu do více větví. Počet větví není omezen, může být libovolný. V každé větvi se může vyskytovat více příkazů, nemusí být uvedeny v bloku. Strukturu příkazu `switch` lze zjednodušeně zapsat takto:

```
switch(vyraz)
 {
  case konstanta1: Prikaz1;
  break;
  case konstanta2: Prikaz2;
  break;
  ...
  default PrikazX;
 }
```

Syntaxe příkazu je poměrně složitá, uveďme její stručné vysvětlení. Za příkazem `switch` je uveden výraz, jehož vyhodnocením musí vzniknout celočíselná hodnota. Tělo příkazu tvoří tzv. návěští se syntaxí `case konstanta`, kde konstanta představuje celočíselnou nebo znakovou konstantu. Za návěštím jsou uvedeny příkazy, které mohou být provedeny.

Nejprve je vyhodnocena hodnota výrazu. Poté je vybráno takové návěští, jehož konstanta má stejnou hodnotu jako výraz. Následně se vykoná příkaz umístěný za tímto návěštím. Pokud není nalezeno odpovídající návěští, je vykonán kód nacházející se za návěštím `default`. Návěští `default` je však nepovinné, nemusí být uvedeno.

Příkaz `break` je přítomen proto, aby umožnil předčasné ukončení vykonávání těla příkazu `switch`. Nebyl -li by uveden, v opačném případě by došlo k provedení všech následujících větví bez ohledu na hodnoty návěští za příkazy case. Uveďme pro úplnost, že žádná z konstant se v příkazu nesmí opakovat. Podívejme se na následující příklad ilustrující použití příkazu `switch`.

```
switch(znak)
  {
   case '1':
     a++;
     a*=;
     break;
   case '2':
     a--;
     a/=;
     break;
   default:
     a*=;
     break;
  }
```

Na závěr uveďme příklad ilustrující používání podmínek při řešení kvadratické rovnice.

```
import java.math.*;
public class rovnice
  {
   public static void main (String [] args)
    {
     double x1, x2, a=1, b=2, c=3;
     double diskr=b*b-4*a*c;
     if (diskr>0)
       {
        x1=-b+Math.sqrt(diskr)/(2*a);
        x2=-b/Math.sqrt(diskr)/(2*a);
        System.out.printnln("x1:"+x1);
        System.out.printnln("x2:"+x2);
       }
     else if (diskr==0)
       {
        x1=-b/(2*a);
        System.out.printnln("x1=x2:"+x1);
       }
     else System.out.printnln("Nema reseni");
    }
  }
```

4.2 Přehled příkazů pro opakování

Tyto příkazy nazývané cykly umožňují provádět opakování jednoho nebo více příkazů tvořících tzv. tělo cyklu. Opakování (tzv. iterace) je prováděno na základě vyhodnocování logického výrazu nazvaného podmínka opakování. V Javě existují tři základní typy cyklů: cyklus `for`, cyklus `while`, cyklus `do while`. Při konstrukci podmínky opakování musíme dávat pozor, aby nevznikl nekonečný cyklus, který by probíhal neustále bez možnosti ukončení těla cyklu.

4.2.1 Cyklus while

Cyklus while obsahuje vyhodnocuje podmínku opakování před průchodem cyklu. Pokud je pravdivá, je provedeno tělo cyklu. Používáme ho v případech, kdy nevíme, kolikrát opakování proběhne. Jeho syntaxi lze zapsat takto

```
while (vyraz)
   telo prikazu;
```

Tělo příkazu nemusí proběhnout ani jednou, pokud je výraz napoprvé vyhodnocen jako `false`. V těle cyklu musí být modifikována proměnná ovlivňující hodnotu výrazu, jinak by vznikl nekonečný cyklus. Následující příklad ukazuje výpočet faktoriálu za použití cyklu `while`, hodnota je přečtena ze standardního vstupu.

```
public class faktorial
 {
```

```
  public static void main (String [] args)
    {
     int cislo, f;
     f=1;
     byte [] pole=new byte [30];
     System.out.println ("Zadejte cislo");
     System.in.read(pole);
     String retezec=new String(pole);
     retezec=retezec.trim();
     int cislo=(int)Long.parseLong(retezec);
     while (cislo>0)
      {
       f=f*cislo;
       cislo--;
      }
     System.out.println(cislo+"!="f);
    }
}
```

4.2.2 Cyklus for

Cyklus for je používán v případech, kdy známe údaje o výchozí a koncové hodnotě testovacího výrazu. Syntaxi cyklu for lze zapsat

```
for (inicializacni_vyraz; testovaci_vyraz; zmenovy_vyraz)
      telo_prikazu
```

Inicializační výraz je vykonán pouze jednou, a to ještě před vyhodnocením testovacího výrazu. Testovací výraz určuje, zda se má provést tělo cyklu. Tělo cyklu se provádí tak dlouho, dokud je testovací výraz vyhodnocen jako pravdivý. Změnový výraz se vyhodnotí na konci cyklu po vykonání těla cyklu, používá se pro změnu hodnoty testovacího výrazu. Inicializační, testovací i změnový výraz jsou nepovinné, nemusí být uváděny. Podívejme se na některé příklady. Nejčastěji je prováděna deklarace s inicializací přímo v hlavičce cyklu `for`.

```
for (int i=0;i<10;i++) System.out.println(i);
```

Deklarace může být provedena mimo cyklus, inicializace v cyklu.

```
int i;
for (i=0;i<10;i++) System.out.println(i);
```

Mimo cyklus mohou být umístěny deklarace i inicializace. V tom případě je nutno provádět změnu hodnoty testovacího výrazu v těle cyklu, jinak by vznikl nekonečný cyklus.

```
int i=0;
for (;i<10;i++) System.out.println(i++);
```

Lze vytvořit cyklus s prázdným tělem, kdy součástí změnového výrazu bude i výkonný kód.

```
int i=0;
for (;i<10;System.out.println(i++);
```

Může vytvořit i minimalistickou variantu cyklu využívající stejně jako příkaz `switch` k předčasnému ukončení cyklu příkaz `break`.

```
int i=0;
for (; ; ;)
  {
    if (i<10) System.out.println(i++);
    else break;
  }
```

Poslední tři konstrukce spíše ilustrují možnosti používání cyklu `for`, v praxi nejsou příliš často používány, zápis není přehledný. V cyklu for můžeme používat více inicializačních, testovacích a změnových výrazů, které jsou odděleny operátorem čárka.

```
int i, j;
for (i=0, j=0;i<10;i++,j++)
  {
   System.out.println(i);
   System.out.println(j);
  }
```

Deklarace obou proměnných je nutno v takovém případě provést vně cyklu. Není vhodné, aby byly hodnoty proměnných změnového výrazu na sobě vzájemně závislé.

```
int i, j;
for (i=0, j=0;i<10, j<20;i++,j=3*i)
  {
   System.out.println(i);
   System.out.println(j);
  }
```

V takovém případě je poměrně obtížné na první pohled stanovit hodnoty změnových výrazů.

4.2.3 Cyklus do while

Tento cyklus se od výše uvedených cyklů liší tím, že je podmínka testována až po vykonání těla cyklu. Cyklus proto proběhne vždy nejméně jedenkrát. Používá se opět nejčastěji v případě, kdy přesně nevíme, kolikrát má cyklus proběhnout (ale měl by proběhnout alespoň jednou). Jeho syntaxi lze zapsat takto

```
do telo_cyklu
    while testovaci_vyraz
```

4.2.4 Vnořené cykly

Při složitějších výpočtech jsou často používány vnořené cykly, typickým příkladem je práce např. s maticemi. Vnořený cyklus si můžeme představit jako cyklus nacházející se v jiném cyklu.

```
for (int i=0;i<10;i++)
   for (int j=0;j<10;j++) System.out.println(i*j);
```

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"

4.2.5 Příkaz break

Příkaz break, se kterým jsme se již setkali, umožňuje okamžitě ukončit provádění těla cyklu. Lze se setkat se dvěma variantami zápisu

```
break;
break navesti;
```

První možnost umožňuje "vyskočit" z cyklu, v jehož těle je tento kód zapsán. Druhá varianta provede ukončení provádění cyklu označeného návěštím. Nachází -li se příkay v nějakém vnořeném cyklu, můžeme vyskočit i o několik úrovní výše. Podívejme se na následující příklad.

```
Vyskoc:    //nazev navesti
   for (int i=0;i<10;i++)
     {
       for (int j=0;j<10;j++)
         {
           if (i==j+10)
            {
             System.out.println(i*j);
             break Vyskoc; //Ukonceni cyklu v navesti Vyskoc
             k=i+j;
            }
         }
       }
```

4.2.6 Příkaz continue

Příkaz `continue` slouží k přeskočení zbývající části příkazů nacházejících se v těle cyklu a k přechodu na další iteraci. Stejně jako příkaz `break` existuje ve dvou variantách.

```
continue;
continue navesti;
```

První varianta způsobí vynechání všech příkazů a skok na další iteraci.

```
   for (int i=0;i<10;i++)
     {
       for (int j=0;j<10;j++)
         {
           if (i==j+10)
            {
             System.out.println(i*j);
             continue;
             k++;
            }
         }
       }
```

Druhá varianta umožňuje přeskočení části cyklu, která je označena tímto návěštím. Tato konstrukce se však v praxi příliš často nepoužívá.

Podívejme se na příklad ilustrující používání cyklů ve spolupráci s poli provádějící hledání maximální hodnoty ze zadaných čísel. Vytvoříme inicializované pole, které budeme procházet za použití cyklu for.

```
public class Maximum
 {
   public static void main (String [] args)
     {
      int [] pole={3,67,32,6,33,0,5,76,54,4};
      int max=pole[0];
      for(int i=0;i<pole.length;i++)
       {
        if(pole[i]>max) max=pole[i];
       }
     System.out.println("Max.="+max);
    }
 }
```

Kapitola 5

Objekty, třídy, metody

Objektově orientované programování (OOP) představuje metodický postup řešení problému. Na rozdíl od procedurálního programování vázaného na algoritmus umožňuje (OOP) popis problému realizovat komplexněji a efektivněji. S objektově orientovaným přístupem k řešení problému souvisejí dva pojmy: objekt a třída.

V této kapitole se seznámíme se základy objektově orientovaného programování v jazyce Java. Naučíme se deklarovat třídy, vytvářet objekty, používat metody.

Objekty v programovacím jazyce představují nehmotnou analogií objektů, které nás obklopují v reálném světě. Každý z objektů má určité vlastnosti a chování. Vlastnosti objektu jsou ovlivněny daty, chování objektu pak metodami, které objekty používají pro vzájemnou komunikaci. Objekty spolu mohou komunikovat přes rozhraní.

5.1 Metody

Metoda patří mezi nejčastěji používané konstrukce programovacího jazyka. Představuje samostatnou část programu konající nějakou specializovanou funkci. Metody jsou umístěny mimo hlavní program, mohou být spouštěny z metody `main()` nebo z jakékoliv jiné metody. Metodu zpravidla voláme se seznamem parametrů, kterým předáváme hodnoty potřebné pro výpočet, metoda většinou vrací nějaký výsledek. Existují i metody, kterým nepředáváme žádné údaje, nebo naopak žádné výsledky nevrací.

Výhodou jejich používání je zjednodušení struktury zdrojového kódu či možnost opakovaného provádění výpočtů (tj. nemusíme psát znovu celý kód, pouze zavoláme příslušnou metodu). Z hlediska OOP lze metody chápat jako zprávy zaslané instancím nebo třídám. Metody bývají někdy označovány jako funkce či podprogramy, ale toto označení není přesné.

Dělení metod: Metody dělíme do dvou skupin:

- metody třídy (statické metody).
- metody instance.

Nejprve se seznámíme s první skupinou metod. Většinu získaných poznatků lze následně aplikovat i na metody instance.

5.1.1 Metody třídy

Metody třídy představují zprávy zaslané třídě jako celku, nelze je volat pro konkrétní instanci. Mohou být používány v případech, kdy ještě neexistuje žádná instance třídy. Metody třídy bývají označovány jako *statické metody*.

Ve statických metodách lze používat pouze statické proměnné nebo lokální proměnné, nemohou v nich být používány proměnné instance. Statické metody jsou v Javě používány při matematických výpočtech: např. `Math.sin(x)` představuje statickou metodu třídy Math.

Deklarace a volání metody

Deklarace metody je tvořena hlavičkou metody a tělem metody. Hlavička obsahuje informaci o jménu metody, typu návratové hodnoty, seznam formálních parametrů. Jméno metody je zpravidla psáno malými písmeny, mělo by vyjadřovat činnost metody.

```
//Deklarace metody tridy
static navratovy_typ jmeno_metody(typ form_param1, typ form_param2,...) //Hla-
vicka
{
 //Telo metody
}
```

Tělo metody tvoří výkonný kód umístěný uvnitř složených závorek. Metody třídy jsou deklarovány s klíčovým slovem `static`. Uveďme, že na rozdíl od jazyka C++ nemusíme vytvářet prototyp funkce. V Javě není podstatné, zda volání metody předchází její deklaraci či naopak.

```
jmeno_metody(skut_param1, skut_param2,...) //Volani metody
```

Podívejme se poněkud podrobněji na problematiku parametrů metod. Vysvětleme pojmy formální parametry metod a skutečné parametry metod.

Formální parametry. Deklaraci formálních parametrů nalezneme v hlavičce metody. Stejně, jako jakékoliv jiná proměnná, musí být i formální parametr deklarován s uvedením datového typu a názvu.

Skutečné parametry. Skutečné parametry používáme při volání metody, s hodnotami těchto parametrů metodu voláme. Skutečné parametry by měly být stejného datového typu jako parametry formální. Nejsou -li, je provedena implicitní konverze (pokud je to možné).

Předávání parametrů hodnotou. V Javě jsou předávány parametry primitivních datových typů hodnotou. Formální parametr představuje kopii skutečného parametru. Jakákoliv změna hodnoty formálního parametru neovlivní hodnotu skutečného parametru.

Předávání hodnot skutečných parametrů parametrům formálním je prováděno při volání metody. Hodnoty jsou předávány postupně, tj. hodnotě prvního formálního parametru je předána hodnota prvního skutečného parametru, hodnotě druhého formálního parametru hodnota druhého skutečného parametru, atd...

```
form_param1=skut_param1;
form_param2=skut_param2;
...
```

Návratový typ. Pokud metoda nevrací žádnou hodnotu, je její návratový typ `void`. Má -li metoda návratovou hodnotu (tj. její návratová hodnota je jiného typu než void), musí metoda obsahovat klíčové slovo `return` s uvedením hodnoty či výrazu, který bude metoda navracet jako výsledek. Dále již nesmí následovat žádný kód, byl by nedostupný, tzv. *unreachable kód*.

Metoda v Javě je schopna vrátit nejvýše jednu hodnotu. Nelze tedy vytvořit metodu, která by přímo vracela dvě hodnoty. Toto omezení lze "obejít" předáváním parametrů odkazem (viz dále) či předat metodě jako parametr složitější strukturu, do které budou výsledky uloženy.

Podívejme se na následující příklad, metoda `obvod()` slouží k výpočtu obvodu kruhu.

```
static double obvod (double r)// formalni parametr r
  {
   return 2*Math.PI*r;
  }
```

Pokud je tělo metody "krátké", lze metodu zapsat do jednoho řádku

```
static double obvod (double r){return 2*Math.PI*r;}
```

Metoda má jeden formální parametr typu `double`, vrací také hodnotu typu `double`: za klíčovým slovem `return` následuje výpočet $2\pi r$. Metodu voláme z "hlavního" programu uvedením jejího názvu a seznamu skutečných parametrů v kulatých závorkách.

```
public class kruh
  {
   //deklarace metody obvod
   static double obvod (double r)
    {
     return 2*Math.PI*r;
    }
   //deklarace metody main
   public static void main (String [] args)
    {
     double pol=50;
     //volani metody obvod se skutecnym parametrem pol
     double obv=obvod(pol);
     System.out.println(obv);
    }
 }
```

Předávání parametrů odkazem. V Javě existuje možnost předávání parametrů odkazem. Týká se pouze neprimitivních datových typů, tj. polí a objektů. V takovém případě změna hodnoty formálního parametru způsobí změnu hodnoty skutečného parametru. Při předávání nevzniká kopie pole či objektu, ale kopie odkazu na pole či objekt.

```
static navratovy_typ jmeno_metody(obj_typ form_param1, obj_typ form_param2,...)
{
 form_param1=... //Zmena form. parametru->zmenu skut. parametru
}
```

Jednoduchým příkladem ilustrujícím předávání parametrů odkazem je třídění. Metodě předáváme nesetříděné pole, které je v těle metody setříděno. Na následují ukázce je příklad třídícího algoritmu Bubble Sort.

```
void bubbleSort(double x [])  //formalnim parametrem odkaz na pole
  {
    for (int i=0;i<x.length;i++)
      {
        for (int j=x.length-1;j>i;j--)
          {
            if (x[j]<x[j-1])
              {
                double pom=x[j] //Lokalni promenna
                x[j]=x[j-1]     //Modifikace pole
                x[j-1]=pom      //Modifikace pole
               }
          }
       }
    }
```

Lokální proměnné. V metodách mohou být deklarovány lokální proměnné. Princip deklarace je stejný, jako by se jednalo o "běžnou proměnnou". Lokální proměnné vznikají při volání metody a zanikají po jejím ukončení. Na rozdíl od datových položek třídy není prováděna jejich automatická inicializace, kompilátor však kontroluje, zda lokální proměnná byla inicializována. Hodnoty lokálních proměnných z předchozích volání metod se neuchovávají.

Metoda bez návratové hodnoty

V praxi jsou tyto metody poměrně často používány, bývají nazývány procedurami. Jejich návratový typ je `void`. Takové metody slouží např. pro realizaci vstupních či výstupních operací (tj. tiskové výstupy).

```
static void tisk ()
  {
   System.out.println("Obvod je:"+obv+" m");
  }
```

Metody s více parametry

Metodě lze předat více než jeden parametr, parametry navíc mohou být různého datového typu. Jejich počet by však neměl být příliš velký (cca 5), taková metoda se stává nepřehlednou. Na první pohled není jasné, která data do výpočtu vstupují.

Podívejme se na příklad provádějící výpočet odvěsny v pravoúhlém trojúhelníku za použití Pythagorovy věty.

```
static double pythagoras (double a , double b)
 {
  return Math.sqrt(a*a+b*b);
 }
```

Metodu `pythagoras()` lze volat např. takto

```
double prepona=pythagoras(odvesna1, odvesna2);
```

5.1.2 Implicitní a explicitní konverze.

Zaměřme se podrobněji na problematiku implicitní a explicitní konverze mezi skutečnými a formálními parametry. Pokud je návratový typ metody odlišný od typu návratové hodnoty, je prováděna buď implicitní konverze nebo explicitní konverze. Implicitní konverze je prováděna kompilátorem automaticky, uživatel si ji v podstatě ani nevšimne, viz následující příklad.

```
static double pythagoras (int a , int b)
 {
  double vysledek;
  vysledek=Math.sqrt(a*a+b*b);//rozsirujici konverze
  return vysledek;
 }
```

Jak víme, při implicitní konverzi dochází ke konverzi datového typu s nižším rozsahem na datový typ s vyšším rozsahem,nemůže při ní dojít ke ztrátě přesnosti. Explicitní konverze musí být provedena uživatelem, důvodem je možnost ztráty přesnosti dat při konverzi datového typu s vyšším rozsahem na datový typ s nižším rozsahem, v opačném případě nebude možno zdrojový kód zkompilovat.

```
static int pythagoras (double a , double b)
 {
  double vysledek;
  vysledek=Math.sqrt(a*a+b*b);
  return (int) vysledek; //zuzujici konverze
 }
```

5.1.3 Rekurzivní metody

Rekurze je schopnost metody, datové struktury či objektu volat sebe sama nebo volat metodu, datovou strukturu či objekt podobného typu.

Tento přístup se často uplatňuje při řešení některých úloh. Takovým typům problémů se říká *dekomponovatelné*, mohou být rozloženy na podproblémy stejného nebo podobného typu nad menší množinou dat.

Rekurzivně lze volat každou metodu s výjimkou metody `main()`. Rekurzivní algoritmus musí obsahovat podmínku, při které dojde k jejímu ukončení, aby nepokračovala do nekonečna. V opačném případě bychom hovořili o tzv. *nekonečné rekurzi*. V určitém místě tedy musí dojít k takovému dořešení problému, které nebude rekurzivní. Při každém novém volání dochází k vytvoření nových lokálních proměnných (tato vlastnost je současně nevýhodou rekurze). Poté jsou zpětně dopočítávány hodnoty z předchozích volání metod.

Typický příklad použití rekurze představuje faktoriál.

```
static int fakt(int n)
{
 if (n>1) return n*fakt(n-1);
 else return 1;
}
```

Metodu lze volat např. takto:

```
int cislo=10;
int faktorial=fakt(cislo);
```

Rekurzivní postupy nejsou vždy výhodné, typickým příkladem je výpočet Fibonacciho posloupnosti.

5.1.4 Přetěžování metod

Přetěžování metod je jedním ze základních rysů OOP. Umožňuje deklarovat více metod stejného názvu, které se mohou lišit různým počtem, typem argumentů popř. jejich pořadím. Postup se používá nejčastěji u metod provádějících stejné činnosti, ale s různými typy dat. Pokud zavoláme přetíženou metodu, kompilátor na základě typu parametrů, jejich pořadí či počtu vybere "správnou" metodu.

Podívejme se na přetíženou metodu `vzdalenost()`, provádějící výpočet vzdálenosti mezi body zadané pravoúhlými souřadnicemi, souřadnicovými rozdíly popř. kombinací obou parametrů.

```
static double vzdalenost(double x1, double y1, double x2, double y2)
 {
  return Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
 }
static double vzdalenost(double dx, double dy)
 {
  return Math.sqrt(dx*dy+dy*dy);
 }
static int vzdalenost(int x1, int y1, int x2, int y2)
 {
  return Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
 }
```

Pokud budeme volat metodu `vzdalenost()` s následujícími parametry

```
double vzd=vzdalenost(1.0,0.0,6.0,0.0);
```

obdržíme výsledek 5.0, bude provedena první metoda. Pokud budeme volat metodu `vzdalenost()` s následujícími parametry

```
double vzd=vzdalenost(1,0,6,0);
```

obdržíme výsledek 5, bude provedena třetí metoda. Pokud budeme volat metodu `vzdalenost()` s následujícími parametry

```
double vzd=vzdalenost(5.0,0.0);
```

bude provedena druhá metoda, obdržíme výsledek 5.0.

5.2 Třídy

Třída představuje vzor, prostřednictvím kterého lze vytvářet objekty. Objekty někdy také nazýváme *instancemi třídy*. Tyto pojmy se často vzájemně zaměňují. V programovacím jazyce Java je kód umístěn uvnitř třídy, tj. v bloku s její deklarací. Každý program musí mít minimálně jednu třídu.

Modifikátor	Popis
`public`	*Veřejná třída.* Přístupná i mimo balíček, ve kterém byla deklarována. Nejčastěji používaná varianta.
`abstract`	*Abstraktní třída.* Od třídy nelze vytvořit objekt.
`final`	*Finální třída.* Od třídy nelze odvodit potomky.

Tabulka 5.1: Přehled modifikátorů přístupu ke komponentám třídy.

5.2.1 Deklarace třídy

Deklarace třídy začíná klíčovým slovem `class`, před ním se mohou vyskytovat tzv. *modifikátory přístupu.* Tělo třídy je uzavřeno do složených závorek a obsahuje zpravidla nějaký výkonný kód (analogie těla metod). Prozatím vynecháme případy, kdy třída může být potomkem jiné třídy. S těmito situacemi se setkáme až v kapitole věnované jednoduché dědičnosti.

Následující deklarace třídy je oproti skutečnosti poněkud zjednodušená, avšak pro první seznámení s třídou dostačující a poměrně názorná.

```
modifikátor class identifikator
  {
   //kod
  }
```

Všimněme si, že na rozdíl od jazyka C++ není deklarace třídy ukončena středníkem.

Přístup ke komponentám třídy. Existují tři typy modifikátorů, které ovlivňují přístup ke komponentám třídy. Jejich popis uveďme zatím bez bližšího vysvětlování v tab. 5.1, další podrobnosti se dozvíme až v závěru kapitoly. Pokud před klíčovým slovem `class` není uveden žádný modifikátor, považujeme třídu za *soukromou.* Od takové třídy lze vytvořit objekt, může být rodičovskou třídou. Některé modifikátory lze vzájemně kombinovat, jsou povoleny kombinace 1+2 a 1+3.

5.2.2 Tělo třídy a přístupová práva

Tělo třídy je uzavřeno do složených závorek, nalezneme v něm deklarace členů třídy, tj. datových položek a metod. Ke každé z nich je možno nastavit přístupová práva. V Javě existují čtyři základní typy přístupových práv; jsou podobná přístupovým právům v jazyce C++, jejich přehled nalezneme v tab. 5.2.

Princip zapouzdření. S aplikacemi přístupových práv k jednotlivým se budeme zabývat v dalším výkladu. Zopakujme na tomto místě, že při návrhu třídy je nutno dodržovat princip zapouzdření dat, kdy s datovými položkami mohou manipulovat pouze metody třídy. Měly by být proto deklarovány jako privátní. Metody realizující rozhraní by měly být deklarovány jako veřejné, ostatní metody mohou být deklarovány jako privátní. Zajistíme tak autorizovaný přístup k datům.

Zapouzdření umožňuje před ostatními objekty zatajit vnitřní stav objektu. Při náhodných operacích nemohou jiné objekty měnit stav objektu přímo a zanést do něj nechtěné chyby. Proměnné instance nejsou z vnějšku přímo viditelné, nemůžeme s nimi přímo manipulovat. Metody instance umožňují objektu komunikovat se svým okolím, představují jeho rozhraní.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

Přist. právo	Popis
`public`	Veřejné členy. Mohou být použity kdekoliv: v jiné třídě, v jiném balíku, mohou je používat i případní potomci. Tyto členy představují rozhraní, jako public deklarujeme metody, nikoliv proměnné.
`private`	Soukromé členy. Mohou být používány jen v metodách třídy, do které patří. Nelze k nim přistoupit z vnějšku třídy. Jako private bývají deklarovány proměnné, tímto způsobem je realizován princip zapouzdření dat.
`protected`	Chráněné členy. Mohou být použity v metodách třídy, kde vznikly, v metodách jejich potomků a ve všech třídách téhož balíčku. Potomci nemusí být ve stejném balíčku jako rodičovská třída.
`neuvedeno`	Přátelské členy. Mohou být používány pouze v rámci balíčku.

Tabulka 5.2: Přehled přístupových práv k jednotlivým členům třídy.

Návrh třídy. Před návrhem třídy je nutné pečlivě zvážit, jaké metody a datové položky použijeme. Nevhodná architektura třídy je kontraproduktivní, může vést k omezeni funkčnosti kódu, porušení zásad objektově orientovaného programování či k obtížnému udržování kódu. Návrhu třídy je proto nutné věnovat velkou pozornost. Upozorněme, že třída nemusí obsahovat metodu `main()`, v takovém případě nebude možné kód uvnitř třídy vykonávat přímo.

Postup předvedeme na praktickém příkladu. Vytvoříme veřejnou třídu `Souradnice,` která umožňuje provádět převod polárních souřadnic na pravoúhlé.

```
public class Souradnice
 {
  private double delka, uhel;

  public double x ()
   {
    return delka*Math.cos(180*uhel/Math.PI);
   }

  public double y ()
   {
    return delka*Math.sin(180*uhel/Math.PI);
   }

  public static void main (String [] args)
   {
    //zatim zadny kod
   }
 }
```

Třída `souradnice` obsahuje dvě datové položky, tj. proměnné typu `double`, představující polární souřadnice deklarované jako `private` a dvě metody pro výpočet pravoúhlých souřadnic deklarované jako `public`. Metoda `main()` zatím neobsahuje žádný výkonný kód, její tělo je prázdné.

5.3 Operace s třídami

Využijme dosavadních znalostí o metodách a pokusíme se je dále rozšířit. Na rozdíl od metody třídy je nutno pro metody instance vytvořit nějaký objekt, se kterým je budeme používat.

5.3.1 Vytvoření objektu

Pro vytvoření objektu (resp. instance) je nutno provést dva kroky:

1. *Deklarace referenční proměnné*

   Tento krok představuje vytvoření odkazu na referenční proměnnou určitého datového typu.

2. *Inicializace referenční proměnné objektem*

   Se samotným odkazem není možno pracovat, došlo by k výjimce. V dalším kroku je nutno vytvořit v paměti nový objekt a inicializovat jím vytvořenou referenční proměnnou.

   Pokud objekt není stejného typu jako referenční proměnná, je provedena implicitní konverze. Nelze -li provést implicitní konverzi, musí být provedena konverze explicitní (s těmito postupy se setkáme v kapitole věnované dědičnosti).

Praktický postup budeme ilustrovat na třídě `Souradnice`. Deklarujeme referenční proměnnou typu `Souradnice`

```
Souradnice sour;
```

V dalším kroku vytvoříme nový objekt a inicializujeme jím referenční proměnnou.

```
sour=new Souradnice();
```

Oba řádky bychom mohli spojit do jednoho.

```
Souradnice sour=new Souradnice();
```

Odkaz `sour` od tohoto okamžiku ukazuje na vytvořený objekt. Takto vytvořený objekt má dvě proměnné instance: `delka`, `uhel` a dvě metody instance: `x()`, `y()`. Tento stav lze zobecnit, říkáme že každý vytvořený objekt má k dispozici vlastní *sadu komponent*. Tyto sady komponent jsou vzájemně nezávislé.

5.3.2 Přístup k datovým položkám

K datovým položkám objektu lze přistupovat prostřednictvím tzv. *tečkové notace*. Obecně ji lze zapsat ve tvaru

```
objekt.polozka
```

Všimněme si, že v Javě neexistuje jako v C++ operátor nepřímého přístupu ->. Přístup prostřednictvím tečkové notace je tedy *jediný možný*. V našem případě bychom k datovým položkám mohli přistupovat takto:

```
sour.delka;
sour.uhel;
```

S využitím této konstrukce je možno datové položky i inicializovat.

```
sour.delka=10;
sour.uhel=20;
```

Tímto způsobem jsme vlastně provedli explicitní inicializaci proměnných. Implicitní inicializace byla provedena při vytvoření objektu. Jelikož se jednalo o datové položky, byly inicializovány na hodnotu 0. Vzhledem k tomu, že datové položky byly deklarovány jako soukromé, můžeme k ním tímto způsobem přistupovat pouze z *vnitřku* třídy. Výše uvedený kód zapišme do metody `main()`, která bude vypadat např. takto:

```
public static void main (String [] args)
   {
    Souradnice sour=new Souradnice();
    sour.delka=10;
    sour.uhel=20;
   }
```

Tento kód můžeme zapsat i do jiné metody než `main()`, metodu je pak nutné volat při inicializaci datových položek třídy. Upozorňujeme, že tento postup inicializace není typický. Není navíc příliš pohodlný ani přehledný. Java pro tyto účely používá speciální metodu zvanou *konstruktor*.

Práce s metodami instance

S metodami instance se pracuje podobným způsobem jako s datovými položkami, opět využíváme tečkovou notaci. Syntaxe vypadá takto:

```
objekt.metoda
```

Metody deklarované ve třídě `Souradnice` jsou bezparametrické. Chceme -li určit hodnotu pravoúhlých souřadnic pro aktuální hodnoty polárních souřadnic, vytvoříme nejprve dvě lokální proměnné typu `double`

```
double xpr;
double ypr;
```

do kterých následně uložíme spočtené hodnoty obou souřadnic x,y.

```
xpr=sour.x();
ypr=sour.y();
```

Kód opět zapíšeme do metody `main()`, i když ani zde to není podmínkou. Výsledné hodnoty můžeme vytisknout.

```
public static void main (String [] args)
   {
    Souradnice sour=new Souradnice();//vytvoreni instance
    sour.delka=10; //icializace jejich datovych polozek
    sour.uhel=20;
    double xpr; //vytvoreni lokalnich promennych
    double ypr;
    xpr=sour.x();//vypocet hodnot pravouhlych souradnic
    ypr=sour.y();
    System.out.println("X="+xpr+" Y="+ypr);
   }
```

5.3.3 Rušení objektů

Připomeňme, že v jazyce Java není nutno objekty rušit manuálně. O objekty, na které již neexistuje platný odkaz, se postará garbage collector, který je smaže. Odkaz na objekt může zaniknout dvěma způsoby:

- *Zánikem platnosti odkazu*

  Odkaz se chová stejně jako jakákoliv jiná proměnná. Po ukončení bloku, ve kterém byl deklarován, zaniká.

```
public static void main (String [] args)
   {
     {
      Souradnice sour=new Souradnice();
     } //zanika odkaz na objekt sour
   }
```

- *Přiřazením hodnoty* `null`.

  Hodnota `null` představuje speciální hodnotu, jejíž přiřazením docílíme stavu, kdy odkaz nikam neukazuje. Tímto způsobem ho "vynulujeme" a dosáhneme stejného výsledku jako v předchozím případě.

```
public static void main (String [] args)
   {
     Souradnice sour=new Souradnice();
     sour=null; //zanika odkaz na objekt sour
   }
```

Druhá varianta se používá často v případech, kdy pracujeme s rozsáhlými daty a potřebujeme rychle uvolnit systémové prostředky. Garbage collector provede následně smazání takového objektu. Neprovede ho okamžitě, ale až v případě, kdy to sám uzná za vhodné. Zopakujme, že garbage collector je možné spustit na vyžádání a s vysokou prioritou metodou `System.gc()`.

5.3.4 Konstruktor

V předchozím kódu jsme použili poněkud nepohodlný způsob ve tvaru `objekt.polozka`, který jsme následně i zkritizovali. Tuto konstrukci by v některých případech nebylo možno použít, k datovým položkám deklarovaných jako privátní bychom se nedostali z vnějšku třídy. Popsaný způsob tedy nemůžeme využít pro inicializaci datových položek prováděnou z jiné třídy, což však představuje poměrně častý způsob práce s objekty. Řešení toho zdánlivě neřešitelného problému nabízí konstruktor.

Konstruktor představuje speciální metodu používanou při inicializaci datových položek. Konstruktor je volán v okamžiku vytvoření objektu.

```
Souradnice sour;
sour=new Souradnice(); //ted je volan konstruktor
```

Konstruktor má následující vlastnosti:

- Jméno konstruktoru je shodné se jménem třídy.
- Konstruktor nemá žádnou návratovou hodnotu, ani `void`. Nesmí obsahovat klíčové slovo `return`.
- Konstruktor může mít parametry (explicitní konstruktor) nebo nemusí (implicitní konstruktor).
- Konstruktor může být přetížen.

Implicitní konstruktor

Implicitní konstruktor je konstruktor bezparametrický. Pokud nevytvoříme konstruktor vlastní, je automaticky generován právě implicitní konstruktor. Ten umožňuje provést inicializaci datových položek na výchozí hodnoty. V našem případě by byly proměnné `delka`, `uhel` inicializovány na hodnoty 0.

Pokud bychom chtěli v rámci implicitního konstruktoru inicializovat proměnné na jiné než výchozí hodnoty, museli bychom si konstruktor napsat sami. Následující konstruktor provede inicializaci datových položek třídy na hodnoty 10 (délka) a 20 (úhel).

```
Souradnice()
{
 delka=10;
 uhel=20;
}
```

Pokud budeme vytvářet nové instance třídy souřadnice podle implicitního konstruktoru, všechny datové položky objektů budou inicializovány na stejné hodnoty. Podobného výsledku bychom mohli dosáhnout, pokud bychom upravili deklaraci třídy, a obě proměnné inicializovali zde.

```
public class Souradnice
 {
  private double delka=10;
  private double uhel=20;
  ...
 }
```

Explicitní konstruktor

Explicitní konstruktor je konstruktor parametrický, disponuje formálními parametry. Umožňuje datovým složkám při inicializaci přiřadit libovolnou hodnotu. Explicitní konstruktor je volán s hodnotami parametrů, které budou přiřazeny formálním parametrům. Při inicializaci za použití explicitního konstruktoru může být každý z objektů inicKonstruktor tedy umožňuje provádět inicializaci objektu při jeho vytvoření.ializován *jinak*, což při použití implicitního konstruktoru není možné.

Podívejme na inicializaxi s využitíme explicitního konstruktoru:

```
Souradnice (double d, double u)
 {
  delka=d;
  uhel=u;
 }
```

Chceme -li vytvořit nový objekt a inicializovat ho explicitním konstruktorem, můžeme postupovat takto:

```
Souradnice sour;
sour=new Souradnice(10, 20); //volan explicitni konstruktor
```

S využitím odkazu this lze explicitní konstruktor přepsat do tvaru, ve kterém mají proměnné instance stejné jméno jako formální parametry.

```
Souradnice (double delka, double uhel)
 {
  this.delka=delka;
  this.uhel=uhel;
 }
```

Odkaz `this` tedy umožní rozlišit obě proměnné tak, že nedojde ke konfliktu jejich identifikátorů. Na případech implicitního a explicitního konstruktoru je patrné, že konstruktory mohou být přetěžovány. Přetěžování explicitních konstruktorů se řídí stejnými zákonitostmi jako přetěžování ostatních metod, nezapomeňme, že i konstruktor je "pouze" metodou.

5.3.5 Odkaz this

Existuje speciální typ odkazu, který má každá z metod. Odkaz `this` ukazuje na objekt, se kterým byla metoda volána. Tento odkaz je "skrytý", není potřeba ho někde inicializovat. Zaručuje, že každá metoda pracuje s daty svého objektu. S odkazem `this` pracujeme také prostřednictvím tečkové notace. Pro bližší pochopení lze výše uvedený implicitní konstruktor zapsat prostřednictvím `this`.

```
Souradnice()
{
 this.delka=10;
 this.uhel=20;
}
```

5.3.6 Inicializace objektu jiným objektem

Odkaz `this` je možno použít i při inicializaci jednoho objektu objektem jiným za použití explicitního konstruktoru. Na rozdíl od C++ není nutno v takovém případě vytvářet kopírovací konstruktor. Podívejme se na následující příklad, ve kterém definujeme explicitní konstruktor ve tvaru

```
Souradnice (Souradnice s)
 {
  this.delka=s.delka;
  this.uhel=s.uhel;
 }
```

Takový explicitní konstruktor má jako formální parametr odkaz na jiný objekt třídy. Chceme -li vytvořit objekt tak, aby byl při svém vzniku inicializován jiným objektem, budeme odkaz na tento objekt předávat jako parametr explicitnímu konstruktoru.

```
Souradnice sour=new Souradnice(10, 20);
//explicitni inicializace objektu sour2 objektem sour
Souradnice sour2=new Souradnice(sour);
```

Tento postup je v OOP velmi *často* používán, s jeho pomocí můžeme vytvářet poměrně snadno kopie objektů. Čtenář by se s ním měl proto nejen seznámit, ale měl by se ho naučit i používat.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

5.3.7 Spuštění metody v konstruktoru.

V těle konstruktoru lze volat i metody. Tento přístup se používá v případě, kdy je nutno provést "složitější" inicializaci datových položek (např. tak, že hodnoty jsou jim nastaveny nějakým funkčním předpisem). Podívejme se na následující, byť poněkud umělý případ.

```
Souradnice (double d, double u)
{
 NastavParametry (d,u);
}
void NastavParametry(double d, double u)
{
 this.delka=d;
 this.uhel=u;
}
```

Objekt následně inicializujeme stejným způsobem

```
Souradnice sour=NEW Souradnice(10, 20);
```

5.3.8 Práce se statickými členy třídy

Mezi statické členy třídy patří statické proměnné a statické metody. Statické proměnné nazýváme proměnné třídy, budou společné všem instancím třídy. Statické metody jsou takové metody, se kterými můžeme pracovat, aniž je vytvořena nějaká instance třídy. Statické datové položky třídy jsou deklarovány s klíčovým slovem `static`.

Proměnné třídy

Nejprve se seznámíme se statickými proměnnými a jejich využitím. Statické proměnné se velmi často používají jako tzv. *počitadla instancí*. Pomocí nich můžeme zjistit, kolik objektů dané třídy již bylo vytvořeno. Pokud je součástí třídy statická proměnná, můžeme na vhodném místě, např. v konstruktoru, provádět její inkrementaci.

Ke statickým proměnným přistupujeme konstrukcí

```
trida.promenna;
```

Všimněme si toho rozdílu, k proměnným instance přistupujeme prostřednictvím odkazu `this`. Statické proměnné jsou inicializovány pouze 1x, a to při svém vytvoření.

Upravme deklaraci naší třídy do tvaru:

```
public class Souradnice
 {
  private double delka;
  private double uhel;
  private static int pocet;
  ...
 }
```

V deklaraci třídy přibyla statická proměnná `pocet`, která je při vytvoření první instance inicializována na výchozí hodnotu, tj. 0. Bude nutno upravit i konstruktor (resp. všechny konstruktory), ve kterém by měla být statická proměnná při vytvoření každé nové instance inkrementována o hodnotu 1.

```
Souradnice (double delka, double uhel)
 {
  this.delka=delka;
  this.uhel=uhel;
  Souradnice.pocet++;
 }
```

Při každém vytvoření nového objektu za použití explicitního konstruktoru

```
Souradnice sour1=NEW Souradnice(10, 20);
Souradnice sour2=NEW Souradnice(10, 20);
Souradnice sour3=NEW Souradnice(10, 20);
```

se zvýší hodnota statické proměnné pocet o 1.

Metody třídy

Metody třídy (tj. statické metody) mohou používat pouze lokální nebo statické proměnné. S tímto faktem jsme se již seznámili v úvodu kapitoly. Přesto se statické metody používají v OOP poměrně často. Jsou dva typické příklady použití.

- První představují metody třídy `Math`, které jsou deklarovány jako statické. S tímto přístupem jsme se v praxi již několikrát setkali, uvádíme ho tedy pouze pro úplnost.
- Druhou možností použití statických metod je spolupráce se statickými proměnnými. Ty jsou stejně jako jiné proměnné deklarovány zpravidla jako privátní. Pro přístup k nim je tedy nutno použít statickou metodu. Pomocí metod třídy tedy můžeme měnit nebo získávat hodnoty proměnných třídy.

Ke statickým metodám přistupujeme ve tvaru

```
trida.metoda;
```

Pokud ke statické třídě přistupujeme z třídy, ve které je deklarována, můžeme název třídy vynechat. Práci s metodami třídy budeme ilustrovat na následujícím příkladu: do třídy `Souradnice` pridejme statickou metodu, která bude vracet počet vytvořených objektů

```
public static int Pocet() {return pocet;}
```

Budeme -li pomocí této statické metody vrátit počet instancí, můžeme to provést např. takto.

```
public static void main (String [] args)
   {
    //Vytvoreni instanci
    Souradnice sour1=NEW Souradnice(10, 20);
    Souradnice sour2=NEW Souradnice(10, 20);
    Souradnice sour3=NEW Souradnice(10, 20);

    //Spocitani instanci
    int kolik=Souradnice.Pocet();
    System.out.println ("Celkem objektu:"+kolik);
   }
```

5.3.9 Gettery a settery

Dodržujeme -li princip zapouzdření, k datovým položkám třídy (deklarovaným jako `private`) můžeme přistupovat a manipulovat s nimi pouze prostřednictvím metod. V Javě jsou standardizovány názvy pro určité typy metod:

- Pro nastavení hodnot datových položek jsou používány tzv. *settery*, tj. metody mající ve svém názvu slovo "set". V případě vzorové třídy můžeme pro nastavení hodnot datových položek `delka/uhel` vytvořit metody:

```
public void setDelka(double d)
{
 this.delka=d;
}
public void setUhel(double u)
{
 this.uhel=u;
}
```

- Pro získávání hodnot datových položek bývají používány tzv. *gettery*, tj. metody mající ve svém názvu slovo "get". V případě vzorové třídy můžeme pro získání hodnot datových položek `delka/uhel` vytvořit metody:

```
public double getDelka() {return this.delka;}
public double getUhel() {return this.uhel;}
```

Nic nám nebrání, metody pro nastavení a získání hodnot pojmenovat jinak, nicméně výše uvedený způsob je v Javě velmi často používán. Řada vývojových prostředí umí automaticky generovat pro datové položky gettery a settery.

5.3.10 Pole objektů

Pracujeme -li s větším množstvím instancí jedné třídy, je vhodné ukládat je do pole. Práce s polem objektů je podobná jako práce s polem primitivních datových typů. Nestačí však vytvořit pouze pole, musíme provést i druhý krok, a tím je vytvoření instancí jednotlivých objektů prostřednictvím operátoru new. K jednotlivým objektům lze následně přistupovat prostřednictvím indexu.

Postup budeme ilustrovat na praktickém případě třídy `Souradnice`. Budeme vytvářet pole 10 objektů. V prvním kroku deklarujeme odkaz ukazující na pole objektů.

```
Souradnice [] pole;
```

Následně vytvoříme pole pole odkazů na objekty, kterým inicializujeme odkaz:

```
pole=new Souradnice[10];
```

V tomto okamžiku však jednotlivé prvky pole neukazují na žádné konkrétní objekty, jejich hodnota je `null`. Teprve nyní vytvoříme jednotlivé objekty, odkazy na ně inicializujeme položky pole. Inicializace proměnných instance může proběhnout jak prostřednictvím implicitního, tak prostřednictvím explicitního konstruktoru; v našem případě proběhne inicializace prostřednictvím explicitního konstruktoru.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

```
for (int i=0;i<pole.length;i++)
 {
  pole[i]=new Souradnice(10,20);
 }
```

S jednotlivými objekty můžeme pracovat prostřednictvím odkazů, Jednotlivé hodnoty souřadnic x, y můžeme např. vypsat za použití cyklu.

```
for (int i=0;i<pole.length;i++)
 {
  System.out.println("X="+pole[i].x()+"Y="+pole[i].y());
  pole[i]=new Souradnice(10,20);
 }
```

5.3.11 Předávání objektů jako parametrů

Objekty jsou na rozdíl od primitivních datových typů v Javě předávány odkazem. Pro připomenutí uveďme, že se tento přístup týká i práce s poli.

Princip předávání objektů. Formálním parametrem v metodě je odkaz na objekt třídy, tj. reference. Skutečný parametr představuje odkaz na již existující objekt. Oba odkazy by měly být stejného datového typu, v opačném případě je provedena implicitní konverze (pokud je to možné).

Prostřednictvím reference může metoda přistupovat k metodám volajícího objektu (popř. k datovým položkám, nejsou -li deklarovány jako privátní). Tímto způsobem můžeme prostřednictvím formálního parametru měnit hodnoty skutečného parametru, tj. "volající" data.

Tento přístup ilustrujeme na nové třídě `Prevod`, kterou vytvoříme. Bude v ní implementována i metoda `vzdalenost()`, která zpětně z pravoúhlých souřadnic spočte jednu z polárních souřadnic: vzdálenost.

```
public class Prevod
{
 public double vzdalenost (Souradnice objekt)
  {
   return Math.sqrt(objekt.x*objekt.x+objekt.y*objekt.y));
  }
}
```

Metoda má jako formální parametr odkaz na objekt třídy `Souradnice`. Prostřednictvím formálního parametru přistupuje k metodám `x()` a `y()` volajícího objektu. V metodě `main()` třídy `Souradnice` vytvoříme dva objekty.

```
public static void main (String [] args)
   {
    //Vytvoreni instanci
    Souradnice sour=NEW Souradnice(10, 20);
    Prevod prev=new Prevod();
    //Volani metody a predani odkazu na objekt
    System.out.println("Vzdalenost="+prev.vzdalenost(sour));
   }
```

Metodě `vzdalenost()` předáme odkaz na objekt `sour`, spočtenou vzdálenost následně vytiskneme.

Práce s maticemi. Práci s poli ilustrujme na jednoduchém příkladu maticových výpočtů. Implementujme třídu `Matice`, která bude matici ukládat do 2D pole. Pro základní operace s maticemi vytvořme třídu `Operace`, která bude sčítat dvě matice. Další výpočty (transpozice, součin) by bylo možno realizovat obdobným způsobem, z důvodu jednoduchosti je však nebudeme uvádět.

```
public class Matice
 {
  private double [][] pole;
  private int m, n;
  Matice (double [][] p)
   {
    this.m=p.length;
    this.n=p[0].length;
    pole=new double[m][n];
    for (int i=0; i<m;i++)
     {
      for (int j=0;i<n;j++)
       {
        this.pole[i][j]=p[i][j];
       }
     }
   }
 }
```

Třída `Matice` obsahuje následující datové položky: počet řádků a sloupců matice představovaný proměnnými `m`, `n`, vlastní matici tvořenou 2D polem `pole`. Konstruktor je explicitní, předáváme mu odkaz na již vytvořené 2D pole, jehož obsah zkopírujeme po prvku do vytvářeného objektu. Při vytvoření nového objektu současně provedeme jeho inicializaci. Šlo by použít i jednodušší variantu a inicializovat objekt pouze odkazem na pole.

```
public class Operace
 {
  public void Soucet(Matice a, Matice b, Matice c)
   {
    for (int i=0;i<a.lengt;i++)
     {
      for (int j=0; j<a[0].length;j++)
       {
        c[i]=a[i]+b[i];
       }
     }
   }
 }
```

Třída `Operace` má jednu metodu `soucet()` se 3 formálními parametry: první dva představují odkazy na obě vstupní matice, třetí parametr tvoří odkaz na "výslednou" matici. Nejprve vytvoříme dynamická pole představující matice, následně objekty představující matice. Tento přístup je sice poněkud těžkopádný, vyplývá však z toho, že každou z matic chceme inicializovat explicitně.

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

```
    double [][] m1={{1,2,3},{4,5,6}};
    double [][] m2={{1,0,1},{1,0,1}};
    double [][] m3={{0,0,0},{0,0,0}};
    //vytvoreni objektu
    Matice A=new Matice(m1);
    Matice B=new Matice(m2);
    Matice B=new Matice(m2);
    Operace o=new Operace();
    //Soucet matic
    o.Soucet(A,B,C);
  }
```

Poté vytvoříme objekt třídy `Operace`, jako formální parametry mu předáme odkazy na objekty A, B, C. Následně bychom výslednou matici mohli např. vytisknout, do třídy `Matice` bychom mohli implementovat novou metodu realizující tuto operaci. Výše uvedený přístup je pouze ilustrativní, v praxi by bylo vhodné zamyslet se nad efektivnějším návrhem tříd a metod.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"

Kapitola 6

Dědičnost

Dědičnost je jedním ze základních principů objektově orientovaného programování. Využívá poznatku, že některé objekty mají podobné vlastnosti. Pomocí dědičnosti můžeme při návrhu tříd použít hierarchický přístup a vytvořit strom podřízených a nadřízených tříd. Třídy na nižší úrovni mohou dědit vlastnosti a chování tříd nacházejících se na vyšší úrovni. Novou třídu lze odvodit z podobné existující třídy přidáním požadovaných funkcí a vlastností. Třídu, ze které dědíme, nazýváme *rodičovskou* nebo také *bázovou* třídou. Zděděné třídy označujeme jako třídy *odvozené* popř. je nazýváme *potomky*.

Java, na rozdíl od jiných jazyků, podporuje pouze jednoduchou dědičnost. Potomek tedy může mít nejvýše jednoho rodiče. Částečnou náhradu vícenásobné dědičnosti v Javě představuje rozhraní. Seznámíme se s ním v kapitole 9.

Všimněme si, že dědičnost je v jazyce Java hluboce zakořeněna. Pokud nově vytvářenou třídu explicitně nedědíme z nějaké jiné třídy, nová třída své vlastnosti implicitně dědí od bázové třídy `Object`. Důkazem budeme ilustrovat při vytvoření třídy `NovaTrida`, která nemá žádné proměnné ani metody, pouze implicitní konstruktor:

```
public class NovaTrida {
  public NovaTrida() {
  }
}
```

Pokud vytvoříme instanci třídy `NovaTrida`, budeme mít k dispozici metody `equals()`, `getClass()`, `hashCode()`, `notify()`, `notifyAll()`, `toString()` a `wait()`.

6.1 Deklarace rodičovské a odvozené třídy

Dědičnost naznačuje podobnost mezi odvozenou a rodičovskou třídou. Fakt, že je odvozená třída potomkem rodičovské třídy, uvádíme v deklaraci odvozené třídy příkazem `extends`, za kterým následuje název rodičovské třídy.

```
public class OdvozenaTride extends BazovaTrida
  {
   // nejaky kod
  }
```

Odvozenou třídu tímto postupem zpravidla rozšiřujeme o novou funkčnost nebo upravujeme funkčnost, která nám nevyhovuje.

Princip dědičnosti budeme ilustrovat na praktickém příkladu. Vytvoříme třídou `Auto`, která bude obsahovat datovou položky `rychlost`, `max_rychlost` a implementovat metody `zastav()` a `zrychli()`. Odvozená třída `Osobni` bude obsahovat datové položky `osob`, `max_osob` a implementovat metody `pristup()` a `vystup()`.

Deklarace rodičovské třídy. V souladu s výše uvedenými informacemi vytvoříme deklaraci rodičovské třídy. Obě datové položky budou typu `float`, rychlost auta se mění spojitě. Konstruktor třídy bude explicitní, každému novému vozidlu stanovíme aktuální rychlost a maximální rychlost, kterou nesmí překročit. Metoda `zastav()` nastaví okamžitou rychlost rovnu nule, metoda `pridej()` zvýší aktuální rychlost vozidla o zadanou hodnotu ale tak, aby nepřesáhla maximální rychlost.

```
public class Auto {
  private float rychlost, max_rychlost;
    public Auto(float max_rychlost, float rychlost)
    {
      this.max_rychlost=max_rychlost;
      this.rychlost=rychlost;
    }
    public void zastav()
    {
     this.rychlost=0;
    }

    public void zrychli(double v)
    {
     if(this.rychlost<max_rychlost-v)
      {
         this.rychlost+=v;
      }
     else this.rychlost=max_rychlost;
     }
    }
```

Deklarace odvozené třídy. Datové položky odvozené třídy budou typu int, použijeme explicitní konstruktor, kterému jako formální parametr předáme počet osob a maximální počet osob ve vozidle. Metoda `vystup()` nastaví počet osob ve vozidle nule, metoda `nastup()` zvýší počet lidí ve vozidle o zadanou hodnotu tak, aby nepřesáhl maximální počet přepravovaných osob.

```
public class Osobni extends Auto
 {
  private int lidi, max_lidi;
  public Osobni(int max_lidi, int lidi)
  {
    this.max_lidi=max_lidi;
    this.lidi=lidi;
  }

  public void vystup()
```

```
{
 this.lidi=0;
}

public void pristup(int l)
{
 if(this.lidi<max_lidi-l)
  {
   this.lidi+=l;
  }
 else this.lidi=max_lidi;
 }
}
```

Ačkoliv to zatím na první pohled není zřejmé, implementace odvozené třídy vykazuje jeden podstatný nedostatek. Pokusíme se vytvořit instance odvozené a rodičovské třídy.

```
public static void main(String[] args)
   {
     Auto a = new Auto(150,100);
     Osobni o=new Osobni (4,2);
   }
```

První příkaz vytvoří instanci rodičovské třídy a inicializuje její datové položky. Druhý příkaz vytvoří instanci odvozené třídy, inicializuje datové položky odvozené třídy, ale již neinicializuje položky rodičovské třídy. Na problematiku inicializace datových položek s využitím konstruktorů se tedy budeme muset podívat podrobněji v následující kapitole.

6.1.1 Inicializace rodičovské třídy

Vzhledem k tomu, že pracujeme se dvěma třídami, inicializace instancí rodičovské a odvozené třídy se na první pohled může zdát poněkud nepřehledná. Pro inicializaci datových položek rodičovské třídy používáme konstruktory. Mohou být přetížené. Použijeme -li konstruktor implicitní, budou mít všechny instance rodičovské třídy stejné vlastnosti. Použijeme -li konstruktor explicitní, může mít každá instance jiné vlastnosti.

V případě rodičovské třídy se žádné překvapení nekoná, její datové položky můžeme inicializovat stejným způsobem jako datové položky jakékoliv jiné třídy.

6.1.2 Inicializace odvozené třídy

V případě inicializace datových položek odvozené třídy je situace složitější. Inicializaci datových položek instance odvozené třídy musí předcházet inicializace datových položek rodičovské třídy. Vlastnosti instance odvození třídy mohou být závislé na vlastnostech instance rodičovské třídy.

Vytváříme –li instanci odvozené třídy, je nejprve vyvolán konstruktor základní třídy a teprve potom konstruktor odvozené třídy. Musí být vytvořen jako první, protože konstruktor odvozené třídy může vycházet z položek základní třídy. Konstruktor rodičovské třídy je proveden před tím, než program vstoupí do konstruktoru odvozené třídy. Instance odvozené třídy nemůže být vytvořena dříve, než je vytvořena instance základní třídy.

Konstruktor nadřazené třídy musíme volat pomocí klíčového slova `super()` se seznamem formálních parametrů. Pokud tak neučiníme, je volán automaticky konstruktor implicitní konstruktor rodičovské třídy.

Pokud rodičovská třída obsahuje pouze implicitní konstruktor, nemusíme v odvozené třídě tento konstruktor explicitně volat.

Teprve po vykonání konstruktoru rodičovské třídy smíme provést inicializaci datových položek rodičovské třídy. Tu provádíme standardním způsobem.

Upravený příklad konstruktoru odvozené třídy bude vypadat takto:

```
public Osobni(float max_rychlost, float rychlost, int max_lidi, int lidi)
    {
      super(max_rychlost, rychlost);
      this.max_lidi=max_lidi;
      this.lidi=lidi;
    }
```

Chceme -li vytvořit instanci odvozené třídy, postupujeme takto:

```
Osobni o=new Osobni (150,100, 4,2);
```

V tomto případě již inicializace datových položek rodičovské třídy při vytváření instance odvozené třídy proběhne korektně.

Platí tedy, že odvozená třída zdědí po rodičovské třídě všechny její datové položky a všechny metody s výjimkou konstruktoru.

Metody rodičovské a odvozené třídy pro jednotlivé instance voláme

```
a.zastav();
a.pridej(25);
o.vystup();
o.pristup(2);
```

6.2 Předefinování a přetížení metody

Ve většině případů, kdy doplňujeme metody odvozené třídy o novou funkčnost, provádíme jejich předefinovávání. Připomeňme rozdíl mezi předefinováním a přetížením metody.

Předefinovaná metoda musí mít stejnou hlavičku, tj. jméno i seznam formálních parametrů. Pokud by bylo použito stejné jméno, ale jiný počet formálních parametrů či jiný typ formálních parametrů, došlo by pouze k přetížení metody.

V odvozené třídě můžeme předefinovat metody `zastav()` a `zrychli()` tak, aby vypisovaly aktuální rychlost.

```
public void zastav()
    {
     this.rychlost=0;
     System.out.println(this.rychlost);
    }

    public void zrychli(double v)
    {
     if(this.rychlost<max_rychlost-v)
```

```
        {
           this.rychlost+=v;
        }
       else this.rychlost=max_rychlost;
       System.out.println(rychlost);
      }
```

Vytvoříme přetíženou metodu `zrychli()`, jejímž vykonáním se zvýší rychlost o 10 km/hod.

```
public void zrychli()
    {
     if(this.rychlost<max_rychlost-10)
      {
         this.rychlost+=10;
      }
     else this.rychlost=max_rychlost;
     System.out.println(rychlost);
    }
```

Přetíženou metodu voláme ve tvaru

```
a.zrychli();
```

6.3 Finální metody a finální třídy

Finální metody představují takové metody, které nemohou být v odvozené třídě předefinovány. Taková metoda nemůže být překryta, pouze přetížena. Finální metody jsou označeny v rodičovské třídě klíčovým slovem `final`. Používají se ve speciálních případech, např. z důvodu zaručení stejné funkčnosti metody i v odvozené třídě.

Typickým příkladem může být metoda `zastav()`, která jakémukoliv potomkovi třídy `Auto` nastaví nulovou rychlost, důsledek zastavení je u všech aut stejný, mají nulovou rychlost.

```
final void zastav()
    {
     this.rychlost=0;
     System.out.println(this.rychlost);
    }
```

Finální třídy představují takové třídy, které není možné zdědit. Bývají nazývány *koncovými* třídami. Třídy označené jako `final` mohou být při překladu optimalizovány vzhledem k rychlosti. Ukázka finální třídy.

```
final class Auto {
  private float rychlost, max_rychlost;
    public Auto(float max_rychlost, float rychlost)
    {
      this.max_rychlost=max_rychlost;
      this.rychlost=rychlost;
```

```
}
final void zastav()
{
 this.rychlost=0;
 System.out.println(this.rychlost);
}

public void zrychli(double v)
{
 if(this.rychlost<max_rychlost-v)
  {
     this.rychlost+=v;
  }
 else this.rychlost=max_rychlost;
 }
}
```

Finální třída může obsahovat jak metody finální, tak i nefinální.

6.4 Abstraktní metody a abstraktní třídy

Abstraktní metody jsou takové metody, které musí být v odvozené třídě předefinovány. Pokud v rodičovské třídě deklarujeme metodu jako abstraktní, vynucujeme její předefinování v odvozené třídě. Neprovedeme -li v odvozené třídě předefinování takové metody, program nebude přeložen. Abstraktní metody jsou tedy protikladem finálních metod.

Jestliže je v rodičovské třídě umístěna alespoň jedna abstraktní metoda, musí být i rodičovská třída deklarována jako abstraktní. Od abstraktní třídy tedy není možné vytvořit instanci. Jedinou možností manipulace s takovou třídou je její kompletní předefinování v odvozené třídě.[1]

Abstraktní třídy se často používají při tvorbě šablon, podle kterých budou vytvářeny odvozené třídy. V odvozených třídách se pak budou navržené metody implementovat. Konkrétně řečeno: pokud víme, že metoda bude muset být v každé odvozené třídě navržena jinak, implementujeme ji jako abstraktní.

Princip práce s abstraktní třídy ukážeme na příkladu. Vytvoříme abstraktní metodu s názvem `srot()`, která bude signalizovat, že auto již patří do šrotu. Základním kritériem bude počet odjezděných kilometrů. Jiná kritéria budou pro auta osobní, nákladní, sportovní, atd.

```
abstract class Auto
  {
    ...
   abstract bool srot(float km){}
  }
```

V odvozených třídách je nutno metodu `srot()` předefinovat. Ve třídě `Osobni` může vypadat takto.

```
public class Osobni extends Auto
   {
```

[1] Abstraktní metody proto neobsahují žádný výkonný kód. Nemělo by cenu ho uvádět, protože metoda bude stejně předefinována.

```
...
abstract bool srot(float km){}
{
 if (km>200000) return true;
 else return false;
}
```

6.5 Konverze mezi instancemi rodičovské a odvozené třídy

Při operacích s třídami tvořících dědickou posloupnost můžeme provádět konverzi mezi instancemi rodičovské a odvozené třídy. Vytvoříme -li instanci rodičovské a odvozené třídy, platí:

- potomek na předka může být přetypován implicitně.
- předek na potomka musí být přetypován explicitně.

Přetypování předka na potomka nazýváme *přetypováním nahoru*, přetypování na předka *přetypováním dolů*. Přetypování směrem nahoru představuje přetypování konkrétnějšího datového typu na obecnější. Přetypování směrem dolů pak přetypování z obecnějšího datového typu na konkrétnější.

```
Auto a = new Auto();
Osobni o = new Osobni();
a=o;
o=(Osobní)a;
```

Princip si lze pamatovat mnemotechnickou pomůckou. Osobní, nákladní, sportovní vozy jsou auto, ale každé auto není osobní.

6.6 Polymorfismus

Polymorfismus je jednou ze základních vlastností objektově orientovaného programování. Určuje chování instance na základě jejího typu.

Základní princip polymorfismu lze vystihnout takto: Potomek vždy může zastoupit předka. Pomocí referenční proměnné potomka lze pracovat i s metodami předka.

Zkusme se zamyslet nad otázkou, jaká z metod bude vyvolána pro objekt rodičovské nebo odvozené třídy. Odpověď není jednoznačná. Existují dva přístupy:

- Časná vazba
- Pozdní vazba

Za normálních podmínek se kompilátor řídí typem objektu:

```
Auto a = new Auto();
Osobni o= new Osobni();
a.zastav();                 //volana metoda rodic. tridy
o.zastav();                 //volana metoda odvoz. tridy
```

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

Tento přístup nazýváme *časnou* neboli statickou vazbou. Kompilátor rozhoduje podle typu objektu, která z metod bude spuštěna, již před překladem.

Pokud napíšeme následující příkaz

```
Auto a=new Osobni();
```

je vytvořena nová instance třídy `Osobni`, odkaz je konvertován na typ `Auto` (všimněme si, že potomek `Osobni` zastupuje předka `Auto`). Aplikujeme -li na instanci metodu `zastav()`

```
a.zastav();
```

je volána metoda odvozené třídy. Metoda je tedy vyvolána na základě skutečného typu instance. Tento mechanismus nazýváme *pozdní vazbou*.

Dynamická, neboli pozdní vazba provádí rozhodování o tom, která z metod (rodičovská nebo odvozená) bude spuštěna až za běhu programu. Metoda je vybírána podle typu objektu, na který reference odkazuje. Tento postup nazýváme polymorfismem.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"

Kapitola 7

Výjimky

Doposud jsme předpokládali, že naše programy budou fungovat za všech situací, tj. že v nich nemůže dojít za běhu k chybě. Toto je však poměrně idealizovaný model, programátor by měl mít možnost reakce jak na chyby vlastní (odmocnina ze záporného čísla, dělení nulou, atd.), tak na chyby vyplývající z komunikace programu se svým okolím (špatné zadání hodnoty u formátovaného vstupu, nenalezení potřebného souboru, atd.). Pokud by tyto stavy nebyly ošetřeny, mohlo by dojít ke ztrátám dat a z nich vyplývajícím škodám. K takovému programu nebude mít uživatel důvěru a nebude ho používat, navíc může být i nebezpečný (týká se to zejména software pro průmyslové zpracování dat).

Pokud k takové chybě dojde, lze na ní reagovat různými způsoby. Program můžeme ukončit a sdělit, že došlo k chybě. Dojde však ke ztrátě dat, takové řešení je proto nevhodné. Můžeme se také pokusit ošetřit všechny možné chybové stavy, ke kterým by potenciálně mohlo dojít. Tato varianta je lepší, v řadě případů zabráníme ztrátám dat, její implementace však není jednoduchá. Navíc znepřehledňujeme vlastní zdrojový zaváděním pomocných proměnných indikujících, zda výpočet proběhl úspěšně. Nejvýhodnější je použití tzv. *mechanismu výjimek*. Umožňuje chybu zachytit a ošetřit ji na vhodném místě.

Pro úplnost uveďme, že práce s výjimkami v jazyce Java je podobná práci s výjimkami v jazyce C++, navíc ho rozšiřuje o nové principy. V zájmu bezpečnosti Java nutí programátora používat u některých potenciálně nebezpečných konstrukcí výjimky, jinak není možné program zkompilovat.

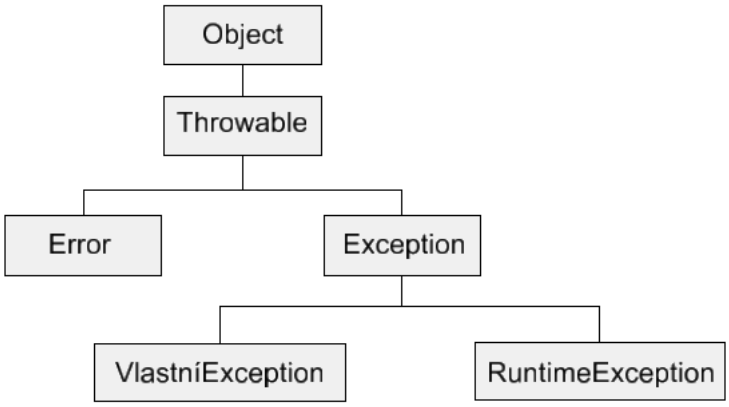
Výjimku můžeme definovat jako stav programu v okamžiku, kdy dojde při jeho vykonávání k chybě. Při vyvolání výjimky v Javě je vytvořen objekt, který nese informaci o typu chyby (a mnohé další informace). Na výjimku JVM upozorní zasláním zprávy ve formě chybového hlášení. Pro vyvolání výjimky je často používán termín *throw exception*, tj. vyhození výjimky; někdy tuto činnost můžeme provádět "uměle".

Ukázka výjimky generované JVM při pokusu o konverzi řetězce na desetinné číslo.

```
Exception in thread "main" java.lang.NumberFormatException:
For input string: "dw" at sun.misc.FloatingDecimal.
readJavaFormatString(FloatingDecimal.java:1224)
at java.lang.Double.parseDouble(Double.java:510)
at balicek.Nacti.getDouble(Nacti.java:31)
at balicek.Obdelnik.<init>(Obdelnik.java:13)
at balicek.Main.main(Main.java:16)
```

7.1 Druhy výjimek

V Javě existují tři základní typy tříd (potomků třídy `Object`), které slouží pro přenos informací o vzniklých výjimkách. Jsou přímými potomky třídy `Throwable`. Jejich hierarchie vypadá takto.



Obrázek 7.1: Hierarchie tříd pro práci s výjimkami.

Podívejme se podrobněji na jejich popis.

Třída Throwable. Se třídou `Throwable` ve velké většině případů nepracujeme (pracujeme však s jejími potomky), neposkytuje žádné konkrétní informace o typu výjimky.

Třída Error. Třída `Error` a výjimky od ní odvozené se používají většinou pouze v souvislosti s chybami JVM. Tyto chyby nedokážeme opravit, ani jim efektivně předcházet. V takovém případě dojde k ukončení programu a zobrazení příslušného chybového hlášení. Touto třídou se proto dále zabývat nebudeme.

Třída RuntimeException. S výjimkami odvozenými od této třídy můžeme již pracovat. Představují výjimky, které vznikají za běhu programu. Typickým příkladem jsou výjimky vznikající jako důsledek provádění aritmetických operací s nevhodnými operandy (např. dělení nulou, odmocnina ze záporného čísla: `ArithmeticException`, atd.) nebo konverzních operací s nevhodnými operandy (konverze textového řetězce na číslo: `NumberFormatException`). Poměrně často se jedná o nevhodné vstupní údaje zadané uživatelem v místech, kde programátor zapomněl takto vzniklé chyby ošetřit. V programu je proto vhodné tyto stavy ošetřit (nic nás k tomu však nenutí), aby např. při zadání nevhodných vstupních údajů nedošlo ke ztrátě dat. Výjimky zařazené do této třídy nazýváme *asynchronní*.

Třída Exception. Výjimky zařazené do této třídy je na rozdíl od výjimek zařazených do třídy `RuntimeException` nutné *vždy* ošetřit, v opačném případě nebude proveden překlad. Jedná se zejména o potenciálně nebezpečné operace týkající se např. prací se soubory, formátovaným vstupem, databázemi. Java je proto poměrně bezpečný jazyk, nutí programátora na tato místa v programu reagovat. Výjimky nazýváme *synchronní*, pracuje se s nimi stejně jako s výjimkami asynchronními.

7.2 Ošetření výjimek

Výjimky lze v Javě ošetřovat třemi základními způsoby:

1. Propagace výjimky: výjimka není ošetřena v metodě, ve které vznikla, je předána do nadřazené úrovně.
2. Ošetření metodou chráněných bloků: Výjimka je ošetřena v metodě, ve které vznikla.

3. Kombinace obou způsobů: Výjimka (částečně) ošetřena v metodě, kde vznikla. Dále je předána do nadřazené úrovně. Třetí metoda je nejčastěji používána, umožňuje komplexní obsloužení výjimky.

Postupně se podíváme na všechny výše uvedené postupy.

7.2.1 Ošetření výjimky propagací

Tento postup je poměrně jednoduchý, výjimku neošetřujeme v místě, kde vznikla, její zpracování předáváme do nadřízené úrovně. Hierarchie předávání je následující: `metoda->main->JVM`. Pokud výjimku neošetříme "před úrovní" JVM, dojde k automatickému ukončení programu a zobrazení informace o chybě.

Tento přístup je často používán, pokud opakovaně provádíme nějakou operaci (cyklus), a nechceme opakovaně chybu řešit (v těle cyklu). Uživatele tedy neinformujeme 50x o tom, že k chybě došlo. Předáme tuto informaci při prvním výskytu výjimky metodě `main()` a zde rozhodneme o dalším běhu programu.

V deklaraci hlavičky metody použijeme klíčové slovo `throws` s identifikátorem výjimky, kterou chceme ošetřit.

```
throws typ_vyjimky;
```

Podívejme se na následující příklad, se kterým jsme se již seznámili v kapitole věnované formátovanému vstupu. Vytvoříme třídu `Nacti`, realizující načítání řetězců z formátovaného vstupu a jejich konverzi na číslo typu `double`.

```
public class Nacti
  {
    private byte [] pole;
    public double getDouble() throws IOException
    {
     pole=new byte[128];
     System.in.read(pole);
     String ret=new String(pole);
     ret=ret.trim();
     double cislo=Double.valueOf(ret).doubleValue();
     return cislo;
    }
   public static void main(String [] args) throws IOException
    {
     Nacti n=new Nacti();
     double cislo=n.getDouble();
    }
  }
```

V hlavičce metody `getDouble()` přibyl kód

```
public double getDouble() throws IOException
```

sdělující, že metoda počítá, že by při jejím vykonávání mohlo dojít k výjimce `IOException`. Mohla by být způsobena špatně provedenými nebo přerušenými operacemi se standardním vstupem. Tato výjimka však, na rozdíl od `NumberFormatException`, neošetřuje konverzi řetězce, ve kterém by se kromě číslic vyskytly další znaky: písmena, speciální znaky, ... (např. uživatel místo 123 zadá 12#). Pokud by k takové výjimce došlo, nebude na této úrovni nijak řešena, bude odsunuta do nadřazené úrovně, tj. do metody main().

```
public static void main(String [] args) throws IOException
```

Zde není také nijak řešena, proto by program byl ukončen JVM. Vhodnější by bylo v tomto místě např. uživateli sdělit, že došlo k chybě vstupu (s tím se seznámíme v následující kapitole).

Který typ výjimky bude ošetřen? Upozorněme, že tímto způsobem je ošetřena pouze výjimka `IOException` a všechny výjimky od této třídy odvozené. Uplatňuje se zde princip, že potomek může zastoupit předka. Pokud by došlo k "jiné" výjimce, program se bude chovat, jako kdyby kód za `throws` neexistoval. Pečlivého programátora by mohlo napadnout, že řešením by bylo ošetření všech chyb. Těch je však cca 30, takový program by se stal nepřehledným a zbytečně složitým. Vyplatí se proto strategie ošetřování pouze nejvíce bolavých míst programu. Jinou možností by bylo použít společného předka, třídu `Exception`.

```
public static void main(String [] args) throws Exception
```

V takovém případě však není možné jednoduše detekovat, k jaké výjimce došlo.

7.2.2 Ošetření výjimky metodou chráněných bloků

Tento postup umožňuje ošetřit výjimku v místě, kde vznikla. Využívá tzv. *chráněného bloku*, uvnitř kterého je uzavřen kód, při jehož vykonávání může dojít k výjimce. Chráněný blok je uveden příkazem `try`. Za ním následuje blok `catch` informující, jaký typ výjimky bude ošetřen. V případě výskytu výjimky v chráněném bloku `try` (typ výjimky musí být shodný s typem výjimky v části `catch`) dojde k jejímu zachycení a začne se vykonávat blok `catch`. Konstrukce `try-catch` může být doplněna i o nepovinný blok `finally`, který se provede vždy.

```
try
 {
  //chraneny blok
 }
catch (typ vyjimky)
 {
  //osetreni vyjimky
 }
finally
 {
  //provede se vzdy
 }
```

Většinou se používá konstrukce `try-catch`, blok `finally` je využíván ve specifických případech (např. při práci se soubory). Pokusme se upravit předchozí příklad tak, aby v okamžiku vzniku výjimky došlo k jejímu ošetření v bloku `catch`.

```
public class Nacti
  {
    private byte [] pole;
    public double getDouble()
    {
     try
      {
```

```
        pole=new byte[128];
        System.in.read(pole);
        String ret=new String(pole);
        ret=ret.trim();
        double cislo=Double.valueOf(ret).doubleValue();
        return cislo;
       }
      catch (IOException e)
       {
        System.out.prinln("Chyba pri zadavani udaju");
        return 0;
       }
     }
    public static void main(String [] args)
     {
      Nacti n=new Nacti();
      double cislo=n.getDouble();
     }
   }
```

Pokud dojde při operacích se standardním vstupem k výjimce typu k výjimce `IOException`, je ošetřena v bloku `catch`. Tímto způsobem lze odchytit pouze výše uvedenou výjimku a výjimky od této třídy odvozené. Upozorněme, že opět nebude odchycena případná výjimka vzniklá konverzí `String` na `double`.

Ošetřená výjimka. Výše uvedené informace lze zobecnit. Jestliže je nalezen příslušný blok `catch` a je do něj vstoupeno, považujeme výjimku za ošetřenou. Pokud by došlo ke vzniku jiného typu výjimky (tj. nebyl by nalezen odpovídající blok catch), nebude výjimka ošetřena. Blok `catch` by měl obsahovat nějaký výkonný kód, který vykoná akci v závislosti na typu výjimky (např. informuje uživatele, že došlo k chybě). Pozor: nikdy *nepoužívejme* prázdné bloky `catch`, v takovém případě by nedošlo k žádné reakci na výjimku.

V případě, kdy bychom chtěli zachytit libovolnou výjimku, museli bychom jako typ ošetřované výjimky použít společného předka, třídu `Exception`.

```
catch (Exception e)
  {
   System.out.prinln("Doslo k nejake vyjimce");
   return 0;
  }
```

Následně však není možné zjistit, k jakému typu výjimky došlo, a na základě toho se rozhodnout, jak výjimku ošetřit (různé typy výjimek zpravidla ošetřujeme různě).

Výpis informací o výjimce. Ošetříme-li výjimku konstrukcí `try-catch`, za běhu programu informace o (ošetřené) výjimce běžným způsobem nezískáme. Je vykonán blok `catch`, nedojde k zastavení programu a výpisu informace o výjimce. Tyto informace lze získat dodatečně použitím metody `printStackTrace()`.

```
catch (NumberFormatException e)
  {
   e.printStackTrace();
  }
```

Získáme tak informace, které lze použít např. při ladění programu.

Kombinování výjimek. Doposud jsme v souvislosti s třídou `Nacti` ošetřovali pouze výjimky typu IOException a "ignorovali" jsme výjimky vzniklé konverzí načteného řetězce. Jeden blok `try` může být ošetřen několik bloky `catch`. Tento postup umožňuje reagovat na více výjimek, jejich počet není v Javě omezen. Jakmile výjimka svým typem odpovídá typu výjimky uvedené v bloku `catch`, je tento blok následně proveden.

```
try
  {
   //nejaky kod
  }
catch (NumberFormatException e)
  {
    //Osetreni vyjimky
  }
catch (IOException e)
  {
    //Osetreni vyjimky
  }
```

Pozor. Ošetřované výjimky musí být seřazeny *vzestupně* dle hierarchie. Pokud bychom tento postup nedodrželi, výjimky budou odchyceny v handleru "nadřazené" třídy a blok catch ošetřující potomka takové třídy neproběhne; vznikne tak *unreachable* (nedostupný) kód. Takový kód se nikdy neprovede!

```
try
  {
   //nejaky kod
  }
catch (Exception e)  //Predek, vyjimka bude odchycena zde
  {
    //Osetreni vyjimky
  }
catch (NumberFormatException e) //Potomek, tento blok jiz neprobehne
  {
    //Osetreni vyjimky
  }
```

7.2.3 Ošetření výjimky metodou chráněných bloků a její předání výše

Tato metoda je kombinací předchozích dvou postupů. Umožňuje výjimku ošetřit v místě, kde vznikla a následně ji předat výše. Výhodou je fakt, že na výjimku lze reagovat opakovaně na různých úrovních programu.

Vyvolání výjimky. Nejprve se seznámíme s klíčovým slovem `throw`, které se používá pro vyvolání a následné šíření výjimky. Obsahuje jako parametr odkaz na objekt nesoucí informaci o chybě. Výjimka je šířena dle výše uvedené hierarchie do "nadřazených" komponent. Neplet'me si toto klíčové slovo s klíčovým slovem `throws`.

```
throw odkaz;
```

Lze použít odkaz na již existující objekt nebo vytvořit nový za použití `new`.

```
catch (IOException e)
  {
    throw e; //pouziti existujiciho odkazu
  }
```

Druhá varianta se často pojí s nějakou podmínkou.

```
if (podminka)
  {
   throw new IOException; // vytvoreni noveho odkazu
  }
```

Postup šíření výjimky prakticky naznačíme při výpočtu faktoriálu. Třída `Faktorial` bude obsahovat dvě metody: `nacti()` realizující načtení hodnoty z formátovaného vstupu a `faktorial()` pro výpočet faktoriálu. Pokud je na vstupu zadán řetězec obsahující jiný znak než číslo, dojde k vyvolání výjimky `IOException`, která je následně pomocí příkazu `throw` předána výše.

```
public int nacti() throws IOException
 {
  try
   {
    byte [] pole=new byte[128];
    System.in.read(pole);
    String ret=new String(pole);
    ret=ret.trim();
    int cislo=Double.valueOf(ret).doubleValue();
    return cislo;
   }
  catch (IOException e)
   {
    System.out.println("Chyba.");
    throw e;
   }
 }
```

Vlastní výpočet faktoriálu je realizován za použití rekurze takto:

```
public int faktorial (int n)
 {
  if (n>1) return n*faktorial(n-1);
  else return 1;
 }
```

Výpočet faktoriálu proběhne v metodě `main()`. Pokud nebyla vyvolána výjimka, je spočtena hodnota faktoriálu v bloku `try`. V případě, kdy došlo k šíření výjimky příkazem `throw`, je nalezen odpovídající blok `catch`, výjimka je opakovaně ošetřena a pomocí metody `printStackTrace()` jsou uživateli sděleny informace o výjimce. Pokud je zadáno číslo číslo neležící v intervalu $<0, 40>$ je vyvolána výjimka `ArithmeticException`.

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

```
        {
           Faktorial f=new Faktorial();
           int c=f.nacti();
           if ((c<0)||(c>40)) throw new ArithmeticException;
           int fakt=f.faktorial(c);
           System.out.println(c+"!="+fakt);
        }
       catch (IOException e)
        {
          System.out.println("Chyba pri vstupu dat.");
          e. printStackTrace();
        }
       catch (ArithmeticException e)
       {
         System.out.println("Hodnota nelezi v intervalu.");
         e. printStackTrace();
       }
   }
```

7.3 Tvorba vlastních výjimek

V Javě lze vytvořit vlastní třídy výjimek. Postup se používá v případě, kdy chceme sami definovat a ošetřit u jiné třídy stavy, které budeme považovat za chybové. Takového stavu může být dosaženo při splnění/nesplnění nějaké podmínky (např. vstupní hodnoty ležící mimo zadaný interval) a není možno ho odchytit běžným mechanismem výjimek, se kterým jsme se seznámili výše.

Tyto třídy budou vytvářeny děděním od třídy `Exception`, bude se jednat o *synchronní* výjimky; princip jejich ošetřování je stejný. Často je používáme proto, abychom odlišili chybové stavy "skutečné" (konverzní, aritmetické operace) a chybové stavy námi "navržené" (při splnění či nesplnění nějaké podmínky), kdy by běh programu (tj. výpočet) nemohl efektivně pokračovat.

Praktický postup ukážeme naznačíme při výpočtu faktoriálu, kdy vstupní údaje budou muset ležet v intervalu <0,40>. Na rozdíl od předchozího příkladu nebudeme volat výjimku třídy `ArithmeticException`, ale výjimku třídy vlastní, kterou nazveme `FaktorialException`.

```
public class FactorialException extends Exception
 {
  public FactorialException()
   {
    super ("Zadana hodnota mimo interval");
   }
 }
```

Třída `ArithmeticException` obsahuje konstruktor volající konstruktor rodičovské třídy `Exception` s parametrem představujícím řetězec nesoucí informaci pro uživatele. Pokud vstupní hodnota neleží v intervalu <0,40> je vyvolána výjimka třídy `FactorialException`. Upravme metodu `faktorial()` tak, aby na tento stav reagovala.

```
public int faktorial (int n) throws FactorialException
    {
     if (n>40)||(n<0) throw new FactorialException();
```

```
        else if n>0 return n*faktorial(n-1);
        else return 1;
       }
```

V metodě `main()` nesmíme zapomenout ošetřit výjimku třídy `FactorialException` v bloku `catch`.

```
public static void main(String[] args) {
        try
         {
            Faktorial f=new Faktorial();
            int c=f.nacti();
            int fakt=f.faktorial(c);
            System.out.println(c+"!="+fakt);
         }
        catch (IOException e)
         {
           System.out.println("Chyba pri vstupu dat.");
           e. printStackTrace();
         }
        catch (FactorialException e)
         {
           System.out.println("Nelze vypocitat faktorial");
           e. printStackTrace();
         }

    }
```

Zadáme -li hodnotu ležící mimo zadaný interval, obdržíme chybové hlášení o výjimce.

```
Nelze vypocitat faktorial
Balicek.FaktorialException: Zadana hodnota mimo interval
at Balicek.Obdelnik.<init>(Faktorial.java:14)
at Balicek.Main.main(Main.java:29)
```

Závěr. Jak jsme z výše uvedených příkladů viděli, práce s výjimkami v Javě je velmi podobná práci s výjimkami v jazyce C++. Java je na rozdíl od C++ bezpečnější, provádí kontrolu ošetření výjimek na úrovni kompilátoru, nikoliv jen jejich ošetření za běhu.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

Kapitola 8

Rozhraní

Rozhraní je nástrojem, který umožňuje *částečně* nahradit neexistenci vícenásobné dědičnosti v Javě. Práce s ním je podobná práci s abstraktními třídami. V rozhraní deklarujeme metody, o jejich definici se musí postarat třída rozhraní implementující. Při definici musí být překryty *všechny* metody deklarované v rozhraní, nikoliv pouze některé. Rozhraní však na rozdíl od třídy nesmí obsahovat deklarace proměnných s výjimkou konstant. Třída může implementovat více než jedno rozhraní. Můžeme si ho tedy představit jako abstraktní třídu tvořenou pouze abstraktními metodami.

Kdy rozhraní použít? Nejčastěji v případech, kdy třída implementuje vlastnosti několika jiných tříd nebo máme vytvořit více tříd s podobnými vlastnostmi. V takových případech může být klasické řešení s využitím dědičnosti poměrně složité, vzniká složitá struktura vzájemně dědících tříd. V současné době je rozhraní využíváno více než abstraktní třídy.

8.1 Implementace jednoho rozhraní třídou

Práce s rozhraním a jeho deklarace připomíná práci s třídami. Rozhraní lze vytvořit s využitím klíčového slova `interface` konstrukcí

```
public interface identifikator {seznam_metod};
```

Modifikátor `public` znamená, že je rozhraní přístupné. Stejně jako třída musí být uloženo v souboru stejného jména jako identifikátor rozhraní. Příklad rozhraní.

```
public interface Rozhrani
 {
  public void metoda1();
  public void metoda2();
 }
```

Ve svém těle obsahuje deklarace metod, které musí být v těle třídy implementující rozhraní překryty, nepostačuje jejich přetížení. Pokud tak neučiníme, program nepůjde spustit. Implementace rozhraní ve třídě je provedena s použitím příkazu `implements` takto

```
implements rozhrani;
```

Třída implementující rozhraní `Rozhrani` může vypadat takto:

```
public class Trida implements Rozhrani
 {
  public void metoda1()
   {
    //predefinovani metody
   }
  public void metoda2()
   {
    //predefinovani metody
   }
 ....
 }
```

Výjimku představuje použití abstraktní třídy, abstraktní třída implementující rozhraní nemusí provádět předefinování metod rozhraní. Tato konstrukce však není příliš často používána.

```
public abstract class Trida implements Rozhrani
 {
 ....
 }
```

Práce s rozhraním. Práci s rozhraním si ukážeme na konkrétním příkladu. Vytvoříme rozhraní `Zvirata`, tvořeném jedinou metodou `pocetNohou()` vracející počet nohou zvířete.

```
public interface Zvire {
 public int pocetNohou();
}
```

Rozhraní bude implementováno dvěma třídami: `Selma` a `Ptak.` V každé z těchto tříd předefinujeme metodu tak, aby vracela správný počet nohou uložený ve stejnojmenné proměnné inicializované v implicitním konstruktoru.

```
public class Selma implements Zvire {
    private int pocet_nohou;
    public Selma() {this.pocet_nohou=4;}
    public int pocetNohou() {return this.pocet_nohou;}
  }
public class Ptak implements Zvire{
    private int pocet_nohou;
    public Ptak() {this.pocet_nohou=2;}
    public int pocetNohou() {return this.pocet_nohou;}
}
```

Ve třídě `Main()` vytvoříme instance obou tříd a aplikujeme na ně metodu `pocetNohou()`. V prvním případě obdržíme hodnotu 4, ve druhém 2, což odpovídá skutečnosti :-).

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

```
public static void main(String[] args) {
    Selma s=new Selma();
    Ptak p=new Ptak();
    System.out.println(s.pocetNohou());
    System.out.println(p.pocetNohou());
  }
```

Proměnná rozhraní. Ve třídě implementující rozhraní je možné vytvořit proměnnou tohoto rozhraní ve formě referenční proměnné, v dalším kroku jí zpravidla inicializujeme objektem stejného typu. S takto vytvořenou instancí lze pracovat, má některé zajímavé vlastnosti (viz dále).

```
Zvire z1=new Selma();
Zvire z2=new Ptak();
```

Aplikujeme-li na instanci metodu `pocetNohou()`, spustí se metoda příslušející třídě dle typu instance. Tato vlasnost je typická pro i dědičnost.

```
z1.pocetNohou(); //Pocet nohou selmy
z2.pocetNohou(); //Pocet nohou ptaka
```

Instance rozhraní a tříd implementujících rozhraní můžeme vzájemně přiřazovat: Instanci rozhraní můžeme přiřadit přímo instanci třídy implementující rozhraní. Opačný postup, tj. přiřazení instance rozhraní instanci třídy, funguje pouze s explicitním přetypováním.

```
z1=s; //Pretypovani potomka na rodice primo
z1=p; //Pretypovani potomka na rodice primo
s=(Selma) z1; //Pretypovani rodice na potomka explicitne
p=(Ptak) z1; //Pretypovani rodice na potomka explicitne
```

První případ představuje přetypování potomka na rodiče, druhý přetypování rodiče na potomka. Tento přístup je stejný, s jakým jsme se již setkali v kapitole věnované dědičnosti tříd.

Instance rozhraní a přístup k metodám. S instancí typu rozhraní deklarované ve třídě implementující rozhraní můžeme přistupovat ke všem metodám deklarovaných ve stejném rozhraní. Nelze přistupovat k metodám, které jsou definovány pouze v těchto třídách ani k jejím proměnným, rozhraní je nezná. S využitím instance rozhraní můžeme tedy manipulovat pouze s metodami tříd v rozhraní implementovaných. Tato vlastnost je v praxi často používána.

Vytvořme metodu `rychlost()` ve třídě `Selma` vracející její rychlost.

```
 public int rychlost()
    {
     return this.rychlost;
    }
```

Na instanci třídy `Selma` můžeme metodu aplikovat, při pokusu o aplikaci metody na instanci `z1`, kompilátor oznámí chybu.

```
s.rychlost(); //spravne
z1.rychlost(); //rozhrani metodu nezna
```

Na základě výše uvedených poznatků můžeme prohlásit, že práce s rozhraními se příliš neliší od práce s abtraktními třídami.

8.2 Implementace více rozhraní třídou

Třída může implementovat více než jedno rozhraní, počet implementovaných rozhraní není omezen. Tento přístup částečně nahrazuje vícenásobnou dědičnost známou např. z jazyka C++. V takovém případě uvádíme za klíčovým slovem implements seznam všech implementovaných rozhraní oddělených čárkou. Všechny metody implementovaných rozhraní musí být ve třídě předefinovány

```
public class Trida implements Rozhrani1, Rozhrani2, Rozhrani3
 {
 ....
 }
```

Vytvořme rozhraní s názvem `Savec`, které bude obsahovat jedinou metodu `jeSavec()` vracejí informaci, zda je dané zvíře savec.

```
public interface Savec {
 public String jeSavec();
}
```

Obě třídy `Selma` a `Ptak` budou implementovat rozhraní `Zvire` a `Savec`. Musejí předefinovávat metody `pocetNohou()` a `jeSavec()`.

```
public class Selma implements Zvire, Savec {
    private int pocet_nohou;
    private String savec;
    public Selma() {
     this.pocet_nohou=4;
     this.savec="Je savec";
    }
    public int pocetNohou() {return this.pocet_nohou;}
    public String jeSavec() {return this.savec;}
  }
public class Ptak implements Zvire, Savec{
    private int pocet_nohou;
    private String savec;

    public Ptak() {
        this.pocet_nohou=2;
        this.savec="Neni savec";
    }
    public int pocetNohou() {return this.pocet_nohou;}
    public String jeSavec() {return this.savec;}
```

Metoda `main()` vypadá takto.

```
public static void main(String[] args) {
  Selma s=new Selma();
  Ptak p=new Ptak();
  System.out.println(s.pocetNohou());
  System.out.println(p.pocetNohou());
```

```
    System.out.println(s.jeSavec());
    System.out.println(p.jeSavec());
}
```

Pro instanci `s` vrací informaci, že je savcem, pro instanci `p`, že savcem není.

8.3 Rozhraní a dědičnost

Práci s rozhraním je možno kombinovat s dědičností tříd. Pokud třída implementuje rozhraní a odvodíme od takové třídy potomky, budou metody rozhraní přístupné i v odvozených třídách. Pokud v nich neprovedememe její předefinování, bude se vždy jednat o metodu rodiče. V odvozených třídách již není povinnost metody rozhraní předefinovávat, máme však tuto možnost k dispozici.

Vytvoříme třídu `Netopyr`, která bude potomkem třídy `Ptak` (netopýr přece také létá :-))).

```
public class Netopyr extends Ptak {
    public Netopyr() {
    }
```

Pokud bychom v metodě `main()` vytvořili novou instanci typu `Netopyr` a metodou `jeSavec()` zjišťovali, zda je netopýr savcem, obdrželi bychom zápornou odpověď.

```
Netopyr n=new Netopyr();
System.out.println(n.jeSavec());
```

Metodu `jeSavec()` proto budeme muset v odvozené třídě předefinovat.

```
public class Netopyr extends Ptak {
    private String savec;

    public Netopyr() {}
    public String jeSavec() {return this.savec;}
}
```

Poté již obdržíme správný a očekávaný výsledek: netopýr je savec.

8.4 Vnitřní třídy

Třídy jsou tvořeny komponentami představovanými metodami a datovými položkami. V deklaraci jedné třídy se může vyskytnout i deklarace jiné třídy. Takovou třídu označujeme jako *vnitřní*, popř. *vnořenou*. Třída, která ji obaluje, se nazývá *vnější*. Tento typ tříd je v Javě používán nejčastěji při práci s grafickým rozhraním. Možnost používání vnitřních tříd je v Javě k dispozici od JDK 1.1. Zmíníme se o nich pouze velmi stručně, podrobnosti lze nalézt ve specializované literatuře.

Deklarace vnější a vnitřní třídy vypadá takto

```
class vnejsi
  {
    ...
    class vnitrni
     {
      ...
     }
  }
```

Přístup ke komponentám vnitřní a vnější třídy. Vnitřní třída může přímo přistupovat ke všem komponentám vnější třídy, tj. jak k metodám, tak proměnným. Ke komponentám vnitřní třídy však z vnější třídy není možné přistupovat přímo ani prostřednictvím referenční proměnné vnější třídy, nýbrž pouze prostřednictvím referenční proměnné vnitřní třídy.

Přistupujeme-li ke komponentám vnější třídy z jiné třídy, je tak možné učinit běžným způsobem prostřednictvím referenční proměnné. Ke komponentám vnitřní třídy však z jiné než vnější třídy není možné přistoupit. Tento typ tříd je v Javě používán nejčastěji při práci s grafickým rozhraním.

Podívejme se na následující příklad. Vytvoříme vnější a vnitřní třídu, vnější třída má jednu datovou položku typu `int a`, vnitřní třída má jednu datovou položku typu `int b`.

```
class Vnejsi {
    private int a;
    public Vnejsi() {}

    class Vnitrni
     {
        private int b;
        public void setB(int b) {this.b = b;}
        public void Nastav1()
        {
         b=10; //OK
         setB(10); //OK
         a=10; //OK
         setA(10); //OK
         Vnejsi v=new Vnejsi(); //OK
         v.setA(10);a=10;  // OK
         setA(10); //OK
         Vnejsi v=new Vnejsi(); //OK
        }
     }
    public void setA(int a) {this.a = a;}

    public void Nastav2()
     {
       b=10; //spatne
       setB(10); //spatne
       a=10; //OK
       setA(10); //OK
       Vnejsi v=new Vnejsi(); //OK
       v.setA(10);//OK
       Vnitrni v2=new Vnitrni();   //OK
```

```
        v2.setB(10);Vnitrni v=new Vnitrni();   //OK
        Vnejsi t2=new Vnejsi(); //OK
      }
}
```

Pokud se budeme pokoušet přistoupit ke komponentám vnitřní a vnější třídy z jiné třídy, výsledek bude vypadat takto.

```
public class Pristup {
...
    public static void main(String[] args) {
        Vnejsi t=new Vnejsi();  //OK
        Vnitrni v=new Vnitrni(); //Spatne
    }
}
```

V Javě jsou vnitřní třídy používány nejčastěji společně s rozhraním.

8.4.1 Vnitřní třídy a rozhraní

Vnitřní třídy zpravidla implementují rozhraní. Vnitřní třída implementující rozhraní musí předefinovat všechny metody rozhraní. Metody rozhraní budou z vnější třídy přístupné *pouze* přes referenční proměnnou typu rozhraní, nebude je možno volat přes referenční proměnnou vnější třídy.

Pro ilustraci práce s rozhraním použijeme třídy z předcházející kapitoly. Vytvoříme opět třídu `Vnejsi` a `Vnitrni`, druhá z nich implementuje rozhraní `Zvire`.

```
public class Vnejsi {
    public Vnejsi(){
    Zvire z1 =new Vnitrni(); //OK
    Vnitrni v=new Vnitrni(); //OK
    Zvire z2 =new Zvire();  //Spatne
    }

    class Vnitrni implements Zvire
     {
        private int pocet_nohou;
        public Vnitrni() {this.pocet_nohou=4;}
        public int pocetNohou() {return pocet_nohou;}
     }
}
```

Přístup k předefinovaným metodám z jiné než vnější třídy. V tomto případě se situace stane složitější. Všimněme si, že pro přístup k předefinovaným metodám rozhraní z jiné než vnější třídy nemůžeme použít instanci vnější třídy ani vnitřní třídy, ani rozhraní:

```
public class Pristup {
...
    Vnejsi v1=new Vnejsi(); //OK
    v1.pocetNohou(); //Spatne
```

```
    Vnitrni v2=new Vnitrni(); //Spatne
    Zvire z1=new Vnejsi(); //Spatne
    Zvire z2=new Vnitrni(); //Spatne
    }
}
```

V takovém případě budeme muset zdrojový kód poněkud upravit. Jediná "správná" konstrukce umožňuje vytvořit instanci třídy `Vnejsi`. Mohli bychom vytvořit metodu, která by prostřednictvím metody této instance vracela instanci třídy `Vnitrni`.

```
public class Vnejsi {
    public Vnejsi(){
    }
    public Vnitrni vratInstanci(){return new Vnitrni()}
    class Vnitrni implements Zvire
     {
        private int pocet_nohou;
        public Vnitrni() {this.pocet_nohou=4;}
        public int pocetNohou() {return pocet_nohou;}
     }
}
```

Upravený kód třídy `Pristup` bude vypadat takto

```
public class Pristup {
...
    Vnejsi v1=new Vnejsi(); //OK
    (v1.vratInstanci()).pocetNohou();
    }
}
```

8.4.2 Anonymní vnitřní třídy

V praxi je často používán koncept tzv. anonymních vnitřních tříd. Ten vychází z faktu, že v metodě `main()` vnější třídy nás nezajímá jméno vnitřní třídy, ale jen její instance. Výsledný kód však nebývá příliš přehledný.

Změna se bude týkat pouze metody `vratInstanci()`. V této metodě nebudeme vytvářet instanci vnitřní třídy, ale instanci rozhraní. Budeme volat konstruktor rozhraní, uvnitř leží jediná předefinovaná metoda `pocetNohou()`.

```
public Vnitrni vratInstanci()
 {
  return new Zvire()
   {
     public int pocetNohou() {return pocet_nohou;};
   }
 }
```

Tento přístup je používán při práci s grafickým uživatelským rozhraním Swing (ale i AWT) v Javě, je nutné se s ním alespoň rámcově seznámit. Konkrétně slouží pro ošetřování událostí přicházejících od více komponent stejného typu, kdy s využitím anonymních vnitřních tříd rozlišujeme původce události.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

Překlad programu s anonymními vnitřními třídami. Při překladu programu obsahujícího anonymní vnitřní třídu kromě souboru `Vnejsi.class` vzniká i soubor `Vnejsi$.class` obsahující kód anonymní vnitřní třídy. Nezapomeňme tedy při distribuci programu obsahujícího anonymní vnitřní třídy přiložit všechny potřebné soubory.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"

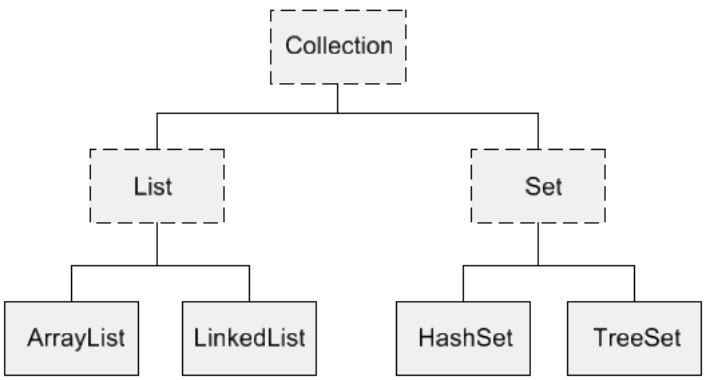
Kapitola 9

Dynamické datové struktury

První otázka, která začínajícího programátora často napadne, zní: K čemu lze použít dynamické datové struktury? Takový uživatel již umí pracovat s polem jak primitivním datových typů, tak i s polem objektů. Zdánlivě tedy nevidí důvod k použití dynamických datových struktur. Ale co když nastane situace, kdy bude chtít do již vytvořeného pole přidat další prvek? Nepůjde to, protože délka pole je definována při jeho vytvoření, tuto hodnotu nelze za běhu programu měnit. Jakmile je délka pole překročena, JVM nás o této situaci informuje vyvoláním výjimky. V takovém případě je výhodné využít dynamických datových struktur.

Dynamické datové struktury používáme pro ukládání takových dat, u kterých předem neznáme rozsah. Dynamické datové struktury mohou za běhu měnit svoji velikost a počet prvků v nich uložených. Podporují navíc některé pokročilé operace s uloženými daty; data lze třídit, transponovat, odstraňovat duplicitní prvky, atd.

Dynamické datové struktury a Java. Dynamické datové struktury nalezneme ve většině programovacích jazyků.[1]Na rozdíl od jazyka C++, kdy se s dynamickými datovými strukturami zachází poněkud nestandardně, v Javě nás žádné překvapení nečeká. Pracujeme s nimi objektově, dynamické datové struktury se chovají jako objekty. Třídy dynamických datových struktur najdeme v balíčku `java.util`. Tyto třídy bývají někdy nazývány jako tzv. *kontejnerové třídy*. Jsou analogií ke skutečnému kontejneru, do kterého můžeme věci odhazovat nebo z něj jiné věci vybírat :-).



Obrázek 9.1: Hierarchická struktura rozhraní Collection. Čárkovanou čarou znázorněna rozhraní, plnou čarou třídy, které je implementují.

[1]Předpokládáme, že čtenář je seznámen se základními typy dynamických datových struktur, nebudeme je popisovat.

Kontejnerové třídy a rozhraní. V Javě je princip práce s kontejnerovými třídami na první pohled poněkud složitější. Základem jsou rozhraní, které jsou implementována několika třídami. Instance těchto tříd představují kontejnery. Rozhraní a třídy vytvářejí stromovou strukturu, která je znázorněna na obr. 9.1 a 9.2. Existují dvě základní rozhraní implementovaná kontejnerovými třídami:

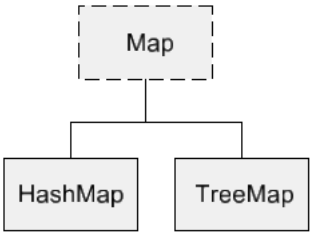
- *Collection*

  Rozhraní `Collection` implementují rozhraní `List` a `Set`. `List` představuje množinu samostatných prvků uspořádaných dle určitého klíče `Set` množinu neuspořádaných prvků. Rozhraní `Set` implementují třídy `ArrayList`, `LinkedList`, `HashSet`, `TreeSet` představující jednotlivé kontejnery. ArrayList je nejpoužívanějším kontejnerem.

- *Map*

  Rozhraní `Collection` implementují třídy `HashMap` a `TreeMap`. Slouží k ukládání uspořádané dvojice klíč-hodnota; s podobným přístupem jsme se seznámili při práci s databázemi.

Kontejnerové třídy mají řadu společných vlastností.



Obrázek 9.2: Hierarchická struktura rozhraní Map. Čárkovanou čarou znázorněno rozhraní, plnou čarou třídy, které ho implementují.

9.1 Kontejnery

Jak jsme již uvedli, kontejnery v Javě představují instance tříd implementujících rozhraní. Řada operací s prvky nad těmito třídami je analogická, mají společného předka.

Typy objektů ukládaných do kontejnerů. Při práci s kontejnery používáme dva přístupy. Ukládáme do nich objekty buď objekty obocného typu nebo konkrétního předem specifikovaného typu.

- *objekty obecného typu*

  V kontejneru mohou být uložena data libovolného datového typu. Po uložení do kontejneru objekt ztrácí informaci o svém typu. Před použitím musí být objekt explicitně přetypován. Výhodou je vysoká míra abstrakce a univerzálnost, do kontejneru můžeme uložit prakticky cokoliv, nevýhodou nižší přehlednost práce. Kontejner tedy může obsahovat jablka i hrušky, při pokusu o přetypování hrušky na jablko může dojít k výjimce.

- *objekty konkrétního typu*

  Do kontejneru jsou ukládány objekty jednoho konkrétního datového typu. Při manipulaci nemusí být přetypovány, práce s takovým kontejnerem je bezpečnější. V kontejneru si tímto přístupem udržujeme pořádek, obsahuje pouze objekty zadaného typu. Tento princip budeme ilustrovat na níže uvedených příkladech.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

Metody společné pro kontejnery. Řada metod je společná pro všechny kontejnery. Každý z kontejnerů má kromě nich i metody speciální. V následující tabulce uvedeme seznam nejčastěji používaných metod společných rozhraní *Collection* s jejich stručným popisem.

Metoda	Popis
`add(Object)`	Přidání objektu zadaného typu do kontejneru.
`clear()`	Odstranění všech prvků z kontejneru.
`isEmpty()`	Dotaz, zda je kontejner prázdný
`remove(Object)`	Odstranění objektu z kontejneru.
`size()`	Velikost kontejneru.
`toArray()`	Konverze prvků kontejneru na pole.

Tabulka 9.1: Přehled nejpoužívanějších metod rozhraní Collection.

9.2 Iterátory

Iterátory představují speciální objekty třídy `Iterator,` které umožňují sekvenční procházení obsahem kontejneru. Toto procházení je velmi rychlé, může být jak jednosměrné, tak obousměrné. Práce s iterátory je jednoduchá, je založena na trojici následujících metod `hasNext()`, `next()`, `remove()`.

Metoda	Popis
`hasNext()`	Testuje, zda se v kontejneru vyskytuje další objekt.
`next()`	Získání hodnoty dalšího objektu.
`remove()`	Vymazání aktuálního objektu.

Tabulka 9.2: Popis metod třídy Iterator.

Podrobněji se s nimi seznámíme v dalším textu.

9.3 Rozhraní List

Rozhraní `List` zahrnuje nejčastěji používanou skupinu kontejnerových tříd. Umožňuje vytvořit a používat dynamické pole, frontu či zásobník. Prvky v těchto kontejnerech nemusí být jedinečné, kontejnery mohou obsahovat i duplicitní položky. Rozhraní je implementováno dvojicí tříd:

- *ArrayList*

  Dynamické pole, k jednotlivým prvkům lze přistupovat přímo i prostřednictvím iterátoru. Přímý přístup je rychlejší, sekvenční přístup by proto neměl by při používán k přidávání, mazání, či editování prvků. ArrayList představuje nejpoužívanější kontejner v Javě.

- *LinkedList*

  Speciální typ seznamu optimalizovaný pro sekvenční přístup dat. Může být používán jako fronta, zásobník, či oboustranná fronta.

Obě třídy disponují řadu pokročilých metod pro práci s daty. V následující kapitole se budeme nejvíce věnovat kontejneru ArrayList. Problematika dynamických datových struktur je velmi rozsáhlá, zájemce nalezne řadu pokročilejších informací v odborné literatuře.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

9.3.1 ArrayList

Kontejner ArrayList představuje dynamické pole. Kontejner je instancí třídy `ArrayList`, pracujeme s ním jako s jakýmkoliv jiným objektem v Javě. Jeho obdobou je v jazyce C++ kontejner vector. V Javě existuje také, ale nedoporučuje se příliš používat, v dokumentaci je uváděn jako obsolete.

Chceme -li vytvořit ArrayList, do kterého mohou být ukládána objekty bez specifikace typu, máme k dispozici několik konstruktorů. Uveďme dva nejčastěji používané

```
ArrayList();
ArrayList(int pocet);
```

První konstruktor vytvoří prázdný ArrayList s výchozí kapacitou deset prvků, druhý konstruktor prázdný ArrayList se zadanou výchozí kapacitou. Pokud budeme do ArrayListu vkládat data velkého rozsahu, můžeme ho na tento krok připravit použitím druhého konstruktoru.

Chceme -li vytvořit ArrayList, do kterého mají být ukládána objekty konkrétního datového typu, lze konstruktory zapsat ve tvaru

```
ArrayList<Objektovy_datovy_Typ>();
ArrayList<Objektovy_datovy_Typ>(int pocet);
```

Přehled základních metod třídy ArrayList. V následující tabulce uvedeme přehled základních metod pro práci s ArrayListem.

Metoda	Popis
`add(index, Object o)`	Na specifikovanou pozici přidá zadaný prvek.
`get(int index)`	Vrací hodnotu prvku se zadanou pozicí v seznamu.
`remove(int index)`	Prvku se zadanou pozicí odstraní ze seznamu.
`set(int index, Object o)`	Prvku se zadanou pozicí v seznamu nastaví zadanou hodnotu.

Tabulka 9.3: Přehled nejpoužívanějších metod pro práci s ArrayListem.

Nevýhodu ArrayListu představuje pomalost některých operací, zejména hledání prostřednictvím metody `get()`. proto jsou používány efektivnější kontejnery, např. HashMap.

Příklad 1. Implementace několika operací ilustrujících práci s ArrayListem: vytvoření seznamu, jeho vytisknutí několika metodami. Editace a odstranění položky. ArrayList bude představovat seznam příjmení.

Pro uložení příjmení použijeme datový typ String. Vytvoříme prázdný ArrayList typu String, s jehož instancí budeme pracovat.

```
ArrayList <String> seznam;
seznam=new ArrayList<String>();
```

V dalším kroku prostřednictvím metody `add()` do ArrayListu přidáme údaje.

```
seznam.add("Novak");
seznam.add("Vopicka");
seznam.add("Vonasek");
seznam.add("Hujer");
seznam.add("Hlinik");
```

Vytisknutí obsahu ArrayListu provedeme třemi metodami. První, s využitím instance třídy `Iterator,` lze zapsat takto:

```
Iterator i = seznam.iterator();
while(i.hasNext())
  {
    System.out.println(i.next());
  }
```

Druhá využívá metody `get()`. Počet prvků ArrayListu určíme metodou `size()` a pomocí cyklu for je postupně procházíme.

```
 for (int i=0;i<seznam.size();i++)
   {
     System.out.println(seznam.get(i));
   }
```

Přidejme na druhou pozici novou položku "Plha". Použijeme k tomu přetíženou metodu `seznam.add(1,"Plha")`. Výsledek vypadá takto:

```
Novak
Plha
Vopicka
Vonasek
Hujer
Hlinik
```

Poslední položku ArrayListu budeme editovat, změníme příjmení z "Hlinik" na "Hlinikova"; poslední položka Arraylistu má index [seznam.size()-1].

```
seznam.set(seznam.size()-1,"Hlinikova");
```

Další ukázku práce s ArrayListem nalezneme v kapitole 10.1.9.2.

9.4 Rozhraní Set

Rozhraní `Set` implementuje stejné rozhraní jako `Collection`. Na rozdíl od `Collection` nemohou kontejnery obsahovat duplicitní položky. Rozhraní je implementováno dvojicí tříd:

- *HashSet*

  Setříděný kontejner využívající techniku hašování. Každý objekt musí mít přiřazenu metodou `hashCode()` hodnotu hašovací funkce. Hodnotu hašovací funkce používá pro vyhledávání prvku. V tomto kontejneru jsou jednotlivé položky nesetříděné. Princip hašování patří mezi pokročilé techniky, nebudeme se jím podrobněji zabývat.

- *TreeSet*

  Setříděný kontejner využívající operace se stromem. Položky v kontejneru jsou setříděny.

Na rozdíl od rozhraní list komponenty HashSet a TreeSet nedisponují žádnými novými nástroji umožňují pouze sekvenční přístup prostřednictvím iterátoru. Neexistuje u nich metoda `get()`, pomocí které lze přistoupit přímo ke konkrétnímu prvku.

9.4.1 TreeSet

Kontejner se používá pro uchovávání setříděných dat neobsahujících duplicitní prvky. Pro vytvoření instance třídy `TreeSet` je k dispozici několik konstruktorů, uvedeme pouze dva nejčastěji používané:

```
TreeSet();
TreeSet<Objektovy_typ>();
```

Jejich význam je podobný jako v případě `ArrayListu`, vytvářejí prázdný `TreeSet` libovolného nebo konkrétního datového typu. Princip fungování kontejneru TreeSet ukážeme na následujícím příkladu.

Příklad 2. Vytvoření seznamu jmen a jejich setřídění prostřednictvím kontejneru TreeSet.

```
TreeSet <String> t;
t=new TreeSet<String>();
t.add("Novak");
t.add("Vopicka");
t.add("Vonasek");
t.add("Hujer");
t.add("Hlinik");
```

Vypíšeme -li prostřednictvím iterátoru obsah tohoto kontejneru, vidíme, že je automaticky setříděn.

```
Iterator it2 = t.iterator();
while(it2.hasNext())
   {
    System.out.println(it2.next());
   }
```

Výsledek vypadá takto:

```
Hlinikova
Hlinik
Hujer
Novak
Vonasek
Vopicka
```

9.5 Rozhraní Map

Rozhraní Map umožňuje uchovávat uspořádané dvojice klíč-hodnota, slouží k operacím s hodnotami na základě jejich klíčů. Toto tvrzení lze zobecnit: v kontejnerech Map vyhledáváme objekty na základě hodnot jiných objektů. Takové kontejnery bývají někdy označovány jako kontejnery *sdružovací*. V současné době jsou velmi populární. Hodnoty klíče nesmějí být duplicitní, každý z klíčů musí mít unikátní hodnotu.

Rozhraní je implementováno dvojicí tříd:

- *HashMap*

  Tento kontejner využívá hašovací tabulku. Výhodou tohoto kontejneru je velmi rychlé vyhledávání, v řádech rychlejší než u ArrayListu. Parametry hašování ovlivňující výkon kontejneru lze nastavit v konstruktoru.

- *TreeMap*

  TreeMap využívá speciální strukturu nazývanou červeno-černý strom. Uspořádané dvojice klíč-hodnota jsou setříděny na základě klíče.

Rozhraní Entry. Při práci s mapami je často používáno rozhraní Entry. Umožňuje manipulace s uspořádanými dvojicemi klíč-hodnota. Implementuje tyto důležité metody:

Metoda	Popis
`getKey()`	Vrací aktuální klíč
`getValue()`	Vrací aktuální hodnotu

Tabulka 9.4: Metody rozhraní Entry.

9.5.1 TreeMap

Na rozdíl od HashMap jsou dvojice klíč-hodnota setříděny, a to vzestupně. Pokud požadujeme jiný typ setřídění, je nutno implementovat rozhraní `Comparable`.[2] K dispozici je několik konstruktorů, uveďme dva nejjednodušší.

```
TreeMap();
TreeMap(pole1,pole2)
```

První konstruktor vytvoří prázdný TreeMap, druhý konstruktor TreeMap takový, že `pole1` tvoří klíče a `pole2` hodnoty. Obě pole musí mít stejnou délku. Přehled nejdůležitějších metod pro práci s kontejnerem TreeMap nalezneme v následující tabulce.

Metoda	Popis
`ContainsKey(Object key)`	Vrací hodnotu true, pokud TreeMap obsahuje klíč key.
`ContainsValue(Object value)`	Vrací hodnotu true, pokud TreeMap obsahuje hodnotu ovalue.
`get(Object key)`	Vrátí hodnotu odpovídající klíči.
`put(Object key, Object value)`	Přidá dvojici klíč-hodnota.
`remove(Object key)`	Odtraní z TreeMap klíč a jemu odpovídající hodnotu.
`entrySet()`	Vrací obsah TreeMap ve formě Setu.

Tabulka 9.5: Přehled metod pro práci s kontejnerem TreeMap.

Práci s kontejner TreeMap neumožňuje přístup k prvkům prostřednictvím iterátoru. Pokud však chceme při práci s kontejnerem TreeMap použít iterátory, můžeme si pomoci menší oklikou. Vytvoříme Set, obsah TreeMap převedeme metodou `entrySet()` do Setu, který již operace s iterátory podporuje. Vlastnosti TreeMap budeme ilustrovat na jednoduchém příkladu.

[2] Problematikou rozhraní Comparable se zabývat nebudeme, zájemce ji najde v odborné literatuře.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"

Příklad 3. Vytvoření seznamu telefonních čísel. Klíč bude představovat příjmení, hodnotu telefonní číslo. Vytvoření kontejneru TreeMap provedeme pomocí

```
TreeMap tels;
tels=new TreeMap();
```

Naplníme ho uspořádanými dvojicemi klíč-hodnota.

```
tels.put("Vopicka","602123456");
tels.put("Vonasek","603111222" );
tels.put("Hujer", "605987654");
tels.put("Hlinik","608500500");
```

Jinou možnost naplnění TreeMap poskytuje druhý konstruktor. Vytvoříme dvojicí polí

```
String [] jmena = {"Vopicka", "Vonasek", "Hujer", "Hlinik"};
String [] telefony = {"602123456", "603111222", "605987654",
  "608500500"};
```

TreeMap vytvoříme pomocí

```
TreeMap tels;
tels=new TreeMap(jmena, telefony);
```

Obsah TreeMap převedeme do Setu.

```
Set s=tels.entrySet();
```

Vypsání obsahu `TreeMap` prostřednictvím iterátoru. Vytvoříme referenční proměnnou rozhraní `Entry` a pomocí metod `getKey()` a `getValue()` získáme aktuální klíč a hodnotu.

```
Iterator it2 = s.iterator();
while(it2.hasNext())
   {
    Map.Entry e=(Map.Entry)it2.next();
    Object jm = e.getKey();
    Object te = e.getValue();
    System.out.println(jm);
    System.out.println(te);
   }
```

Chceme -li nalézt hodnotu odpovídající klíči, použijeme metodu `get()`.

```
System.out.println(tels.get("Vopicka"));
```

Kapitola 10

Java, grafické uživatelské rozhraní

Programovací jazyk Java obsahuje disponuje grafickým uživatelským rozhraním (Graphic User Interface) nezávislým na zařízení. Grafické operace poskytují stejné výsledky bez ohledu na typ počítače či používaný operační systém. Tato vlastnost je v Javě velmi důležitá, uplatňuje se při tvorbě aplikací, jejichž vzhled je pod různými operačními systémy (velmi) podobný. Pro vývoj GUI aplikací není nutné mít k dispozici žádné vývojové prostředí, vizuální komponenty lze vytvářet poměrně jednoduše zápisem zdrojového kódu. Práce ve vývojovém prostředí RAD je příjemnější a pohodlnější, umožňuje vytvořit grafické uživatelské rozhraní bez nutnosti zápisu kódu pouze metodou drag and drop. V této kapitole se budeme zabývat první variantou.

Grafické uživatelské rozhraní. V našich aplikacích můžeme používat řadu vizuálních komponent: formuláře, tlačítka, přepínače, dialogy, grafické komponenty, multimediální komponenty, tiskové komponenty. Tyto komponenty tvoří rozhraní aplikace, umožňují vzájemnou komunikaci mezi uživatelem a aplikací (zadávání dat nebo znázorňování výsledků různých výpočetních či grafických operací). Popsaný přístup je pro Javu přirozenější, práce s formátovaným vstupem či v konzoli není na rozdíl od C/C++ pro Javu typická. Výhodou práce s GUI je snadnější ovladatelnost, přehlednost, názornost. Při návrhu grafického uživatelského rozhraní je nutno zohlednit, že aplikace se nemá přizpůsobovat uživateli a nikoliv uživatel aplikaci.

Typy grafických rozhraní v Javě. Java disponuje dvěma grafickými rozhraními. Starší z nich, AWT (Abstract Windows Toolkit), se již používá jen výjimečně (applety). Jeho nevýhodou byla značná náročnost na hardware a nepříliš pěkný vzhled (subjektivní, leč rozšířené tvrzení), výhodou podpora prakticky všech browserů, AWT rozhraní je vláknově zabezpečené. Aplikace přejímala vzhled OS, na kterém běžela.

Toto rozhraní bylo nahrazeno rozhraním JFC (Java Foundation Classes) obsahujícím jak vizuální, tak nevizuální komponenty. Nevizuální komponenty nazýváme Java Beans (kávové boby), vizuální komponenty představují grafické rozhraní Swing. Swing je bohatší na komponenty, vychází však z AWT. Využívá stejné události, je postaven na podobných komponentách, práce s oběma rozhraními je více-méně analogická. Rozhraní Swing je rychlejší, nevýhodou je fakt, že ho neumí zobrazit korektně všechny browsery, je vláknově nezabezpečené. Vzhled aplikací a jejich komponent běžících nad rozhraním Swing může být měněn, aplikace nemusí přejímat vzhled OS.

Názvy tříd. Názvy tříd v knihovně AWT vycházejí ze standardizovaných jmen komponent (Button, Table, Frame, ...), názvy tříd v rozhraní Swing se odlišují přidáním písmene J před název komponenty (JButton, JTable, JFrame,...). V této kapitole se budeme zabývat výhradně popisem rozhraní Swing.[1] V českém jazyce

[1] Obecně je však rozhraní Swing poměrně pomalé. Jsou proto používána i jiná grafická rozhraní, např. SWT (v prostředí Eclipse).

existuje velmi málo materiálů zabývajících se touto problematikou, podrobné informace o rozhraní Swing lze nalézt na stránkách společnosti Sun: `http://java.sun.com/docs/books/tutorial/uiswing/index.html`

Událostmi řízené programování. Tento způsob programování je značně odlišný od přístupu, který jsme doposud používali. Program je tvořen řadou různých grafických komponent, které jsou za jeho běhu zobrazovány.

Událost (event) vzniká činností uživatele, programu nebo OS (stisk, posun myši, klávesy, maximalizace, minimalizace, zavření okna, změna rozměrů okna, stisk tlačítka, překreslení obrazovky, ...). Každá z vizuálních komponent má vlastní události, na které můžeme reagovat, tzv. je *obsloužit*.

Obslužný kód je psán do metod, které nazýváme *handlery*. Tento způsob programování nazýváme událostmi řízeným programováním.

Vytvoření aplikace zpravidla představuje několik činností

- Implementace výkonné části programu.
- Navržení a vytvoření grafického rozhraní.
- Zajištění možnosti korektního ukončení programu.
- Definice událostí, které mají být v programu obslouženy,
- Zápis kódu pro obsluhu událostí formou handlerů.

Upozorněme, že návrh grafického rozhraní by měl být vždy oddělen od výkonné části programu. Tuto zásadu budeme ještě několikrát opakovat.

10.1 Základní komponenty a události

V této kapitole se seznámíme s některými často používanými vizuálními komponentami knihovny `Swing` a událostmi s nimi spojenými. Popis všech komponent by vydal na samostatnou knihu, nebudeme ho proto uvádět. Nejprve je nutno importovat balíček `javax.swing.*`.

10.1.1 Rozdělení komponent

V Javě je k dispozici velké množství různých typů grafických komponent. Lze je rozdělit do několika skupin, uveďme nejpoužívanější členění na základě hierarchické struktury komponent:

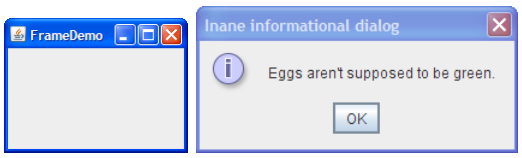
- Top-level komponenty
- Kontejnerové komponenty
- Základní komponenty
- Needitovatelné informační komponenty
- Interaktivní komponenty

V následujícím přehledu uvedeme nejdůležitější zástupce jednotlivých skupin komponent.

Obrázek 10.2: Ukázka komponent JPanel a JScrollPane.

10.1.1.1 Top-level komponenty

Tyto komponenty jsou v hierarchii komponent na nejvyšším stupni. Každá aplikace musí obsahovat alespoň jednu top-level komponentu. Můžeme si je představit jako komponenty, na které mohou být umisťovány jiné komponenty (tj. komponenty podkladové) nebo na ně naopak již žádnou další komponentu umístit nelze.



Obrázek 10.1: Ukázka komponent JFrame a JDialog.

Mezi nejčastěji používané top-level komponenty patří `JFrame` (formulář) a `JDialog`.

10.1.1.2 Kontejnerové komponenty

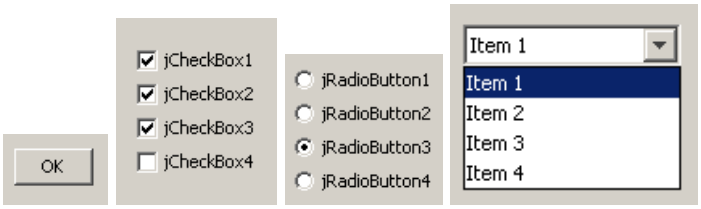
Tyto komponenty se nacházejí na nižší hierarchické úrovni než top-level komponenty. Slouží k seskupování komponent, mohou na ně být umisťovány jiné komponenty stejné nebo nižší hierarchické úrovně. Kontejnerové komponenty bývají zpravidla umístěny na top-level komponentách.

Do této skupiny patří: `JPanel`, `JScrollPane` (panel s posuvníky pro zobrazování obrázků), `JTabbedPane` (panel se záložkami) či `JToolbar` (panel nástrojů).

Obrázek 10.3: Ukázka komponent JTabbedPane a JToolbar.

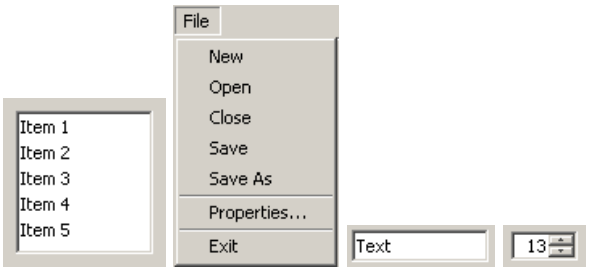
10.1.1.3 Základní komponenty

Tyto komponenty slouží pro zajištění komunikace mezi uživatelem a aplikací. Vyskytují se prakticky v každé aplikaci, napříč programovacími jazyky a operačními systémy.



Obrázek 10.4: Ukázka komponent JButton, JCheckBox, JRadioButton, JComboBox.

Do této skupiny patří: `JButton` (tlačítko), `JCheckBox` (zaškrtávací pole), `JRadioButton` (přepínací tlačítko), `JComboBox` (rozbalovací seznam), `JList` (seznam), `JMenu`, `JTextField` (textové pole), `JSpinner` (zadávání číselných údajů).



Obrázek 10.5: Ukázka komponent JList a JMenu, JTextField, JSpinner.

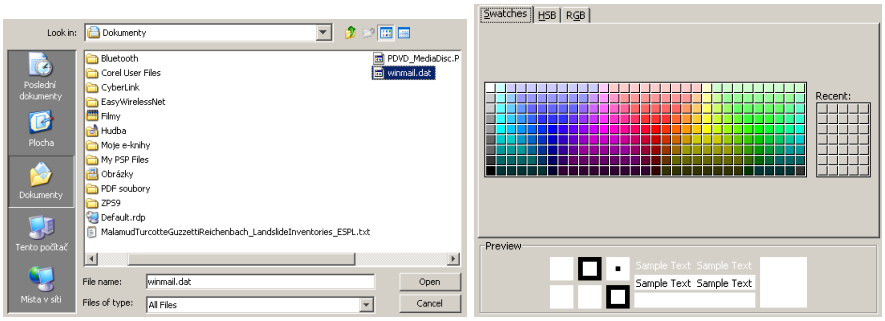
10.1.1.4 Needitovatelné informační komponenty

Někdy bývají nazývány jako informační komponenty. Jejich obsah nemůže být uživatelem měněn. Do této skupiny lze zařadil `JLabel` (popis) či `JProgressBar` (zobrazení průběhu běžícího procesu).

Obrázek 10.6: Ukázka komponent JLabel a JProgressBar.

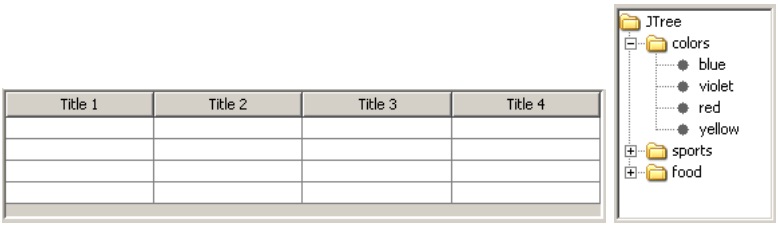
10.1.1.5 Interaktivní komponenty

Jedná se o komponenty představované zpravidla standardními dialogy umožňujícími provádět často opakované činnosti: výběr souboru, volba barvy, hledání souboru. Mohou také zobrazovat výsledky výpočtů ve formátovaném tvaru ve formě tabulek či stromů.



Obrázek 10.7: Ukázka komponent JFileChooser a JColorChooser.

Do této skupiny patří `JFileChooser` (výběr souboru), `JColorChooser` (výběr barvy), `JTree` (Strom), `JTable` (Tabulka).



Obrázek 10.8: Ukázka komponent JTable a JTree.

10.1.2 Formuláře

Patří do skupiny top-level komponent. Jelikož se jedná o nejčastější top-level komponentu, úvodní výklad budeme směřovat právě na formuláře. Pro práci s formuláři je používána třída `JFrame`.

V praxi je výhodnější pracovat se třídou odvozenou od třídy `JFrame`, ve které můžeme s jednotlivými komponentami a jejich parametry efektivněji manipulovat. Tento postup umožňuje oddělit od sebe *grafické* a *aplikační* rozhraní aplikace, v praxi je velmi často používán.

Nejprve se pokusíme vytvořit formulář a zobrazit ho na obrazovce; k tomuto účely použijeme třídu `Okno`.

```
import javax.swing.*;
public class Okno extends JFrame{
    public Okno() {}
 }
```

Ve třídě `Komponenty` budeme pracovat s instancí třídy `Okno`. Instanci nestačí pouze vytvořit, je nutné za použití metody `setVisble(boolean status)` nastavit její zobrazení.

```
public class Komponenty {
    public static void main (String [] args)
    {
      Okno o=new Okno();
      o.setVisible(true);
    }
```

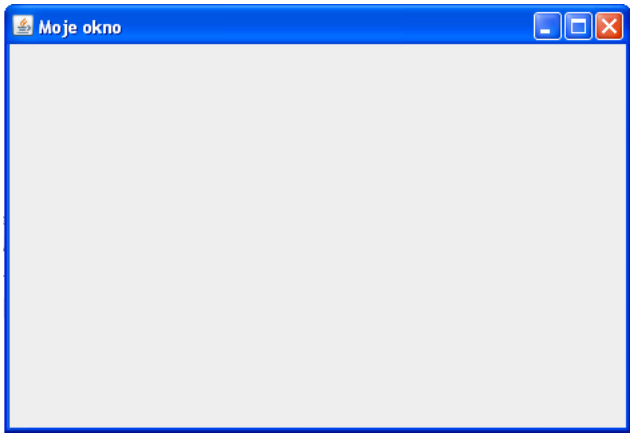
Z takto vzniklého okna bude vidět pouze jeho titulek, není v něm však uvedena žádná informace, což není pro praktickou práci vhodné.

Obrázek 10.9: Ukázka formuláře bez nastavených rozměrů.

Proto prostřednictvím metody `setDimension(int width, int height)` nastavíme formuláři šířku a výšku, hodnoty jsou uváděny v pixelech a prostřednictvím metody `setTitle(String title)` zadáme titulek.

```
public class Komponenty {
    public static void main (String [] args)
    {
      Okno o=new Okno();
      o.setSize(640, 480);
      o.setTitle("Moje okno");
      o.setVisible(true);
    }
```

Výsledek vypadá takto



Obrázek 10.10: Ukázka formuláře s nastavenými rozměry.

Okno však má jeden nedostatek, při jeho vypnutí prostřednictvím křížku sice zmizí, aplikace však běží dále. Do programu musíme začlenit kód zajišťující ukončení programu při zavření okna. Slouží k tomu metoda `setDefaultCloseOperation(int o)`, která by měla být volána ihned po vytvoření instance. Oknu dále nastavíme metodou `setLocation(int x, int y)` polohu na obrazovce.

```
public static void main (String [] args)
{
Okno o=new Okno();
o.setSize(640, 480);
o.setTitle("Moje okno");
o.setLocation(100,100);
o.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
o.setVisible(true);
}
```

Vlastnosti komponent. Pro získání vlastností komponent či jejich nastavení používáme dle výše uvedených konvencí gettery a settery. Jejich názvy jsou v tomto případě tvořeny kombinací vlastnosti, např.: `Size`, `Title`, `Name`, `Font`, `Visible`, `Location` a označení `set` nebo `get`. Gettery jsou bezparametrické, settery obsahují seznam parametrů, které je potřeba komponentě nastavit. Vzniknou tak metody `getSize()`, `getTitle()`, `getName()`, `getFont()`, `getLocation() ...`, či `setSize()`, `setTitle()`, `setName()`, `setFont()`, `setLocation()`. Upozorněme, že veškerá nastavení komponenty je nutné provést *před* tím, než bude komponenta zobrazena.

Zaměření komponenty. Důležitou vlastností komponenty je její *zaměření*, tzv. fokus. V každém okamžiku je jedna z komponent nacházejících se na formuláři aktivní, říkáme o ní, že je zaměřená (tj. má fokus). Tento stav se projeví grafickým zvýrazněním komponenty. Zaměřená komponenta může reagovat i na vstup z klávesnice (např. klávesové zkratky).

Obrázek 10.11: Ukázka zaměření komponenty, tlačítko OK.

10.1.2.1 Nastavení vzhledu komponent

Každému formuláři ale i ostatním ostatním komponentám na něm umístěným, můžeme nastavit různý vzhled, tzv. *look and feel*. Komponenty v Javě jsou skinovatelné. Existuje velké množství různých skinů, které jsou volně dostupné na Internetu. Velikosti, tvary, barvy jednotlivých komponent mohou být různé. Pro správu vzhledů používáme statickou třídu `UIManager`, kterou je nutno importovat. Nejčastěji používanou metodou je metoda `setLookAndFeel()`, které umožňující nastavit vzhled aplikace. Kód je obalen v sekvenci `try-catch`, jedná se o synchronní výjimku.

```
try
 {
  UIManager.setLookAndFeel(UIManager.
  getSystemLookAndFeelClassName());
 }
catch (Exception e) {
 // handle exception
}
```

K dispozici jsou dvě základní varianty:

- `getCrossPlatformLookAndFeelClassName()`:Aplikace bude mít stejný vzhled nezávisle na operačním systému, na kterém běží; tento vzhled nazýváme `MetalLookAndFeel`. Jedná se o výchozí variantu, která bude použita, pokud uživatel neprovede žádné nastavení.
- `getSystemLookAndFeelClassName()`: Aplikace bude mít vzhled obvyklý v daném operačním systému. Na platformě Windows `WindowsLookAndFeel` a na platformě Linux/Unix `MotifLookAndFeel`. Ve většině případů je vhodnější použít tuto variantu.

Nastavení look and feel metodou `setLookAndFeel` musí být provedeno *před* zobrazením formuláře. Existují i metody, s jejichž použitím je možno měnit vzhled aplikace "za běhu". Další podrobnosti o možnostech nastavování vzhledu komponent lze nalézt v dokumentaci na

`http://java.sun.com/docs/books/tutorial/uiswing/lookandfeel/plaf.html`.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

Ukázky vzhledu téže aplikace s výše uvedenými looks and feels jsou zobrazeny na následujícím obrázku.

Obrázek 10.12: Ukázka Look and Feel: MetalLookAndFeel, MotifLookAndFeel, WindowsLookAndFeel.

10.1.2.2 Nastavování parametrů grafického rozhraní v konstruktoru jiné třídy

Přístup, při kterém jsou parametry grafického rozhraní nastavovány v metodě `main()` třídy, ve které byla instance s grafickým rozhraním vytvořena, není vhodný. Často potřebujeme vytvořit několik instancí téže třídy, z nichž každá bude disponovat grafickým rozhraním se stejnými parametry. Z tohoto důvodu je vhodné provést nastavení vlastností instance v konstruktoru třídy.

```
import javax.swing.UIManager;
public class Komponenty {
  public Komponenty()
   {
    Okno o=new Okno();
    o.setSize(640, 480);
    o.setTitle("Moje okno");
    o.setLocation(100,100);
    o.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    o.setVisible(true);
   }
  public static void main (String [] args)
   {
    try
     {
      UIManager.setLookAndFeel(UIManager.
      getSystemLookAndFeelClassName());
     }
    catch (Exception e)
     {
      e.printStackTrace();
     }
    new Komponenty();
   }
```

Nastavení look and feelu je provedeno v metodě `main()`. Pokud se nepovede, zpravidla pokračujeme běžným způsobem, tj. vytvoříme grafické rozhraní a smíříme se s tím, že nevypadá tak, jak požadujeme.

10.1.2.3 Nastavování parametrů grafického rozhraní v konstruktoru téže třídy

Tento přístup umožňuje lépe oddělit třídu vytvářející okno od třídy, která bude toto okno využívat. Vytvoření formuláře je provedeme v konstruktoru třídy `Okno`. Při použití implicitního konstruktoru budou při vytvoření nové instance třídy `Okno` vždy vytvořeny formuláře stejných vlastností.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

```
import javax.swing.UIManager;
public class Okno extends JFrame {
  public Okno()
   {
    this.setSize(640, 480);
    this.setTitle("Moje okno");
    this.setLocation(100,100);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setVisible(true);
   }
public static void main (String [] args)
   {
    try
     {
      UIManager.setLookAndFeel(UIManager.
      getSystemLookAndFeelClassName());
     }
    catch (Exception e)
     {
      e.printStackTrace();
     }
    new Okno();
   }
```

Pokud bychom chtěli velikosti oken nastavit při vytváření instance, bylo by nutno tvorbu formuláře provést v explicitním konstruktoru, jehož formální parametry představují parametry formuláři nastavované.

```
public Okno(int sirka, int vyska)
   {
    this.setSize(sirka, vyska);
    this.setTitle("Moje okno");
    this.setLocation(100,100);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setVisible(true);
   }
```

Ikona programu. Na formuláři je možno zobrazit i ikonu programu představovanou obrázkem, a to v některém z následujících formátů: JPG, GIF,PNG. Lze si tak vytvářet i ikony vlastní a nespoléhat se jen na výchozí. Standardní velikost ikony je 16x16 pixelů. K těmto operacím slouží třída `ImageIcon`. V konstruktoru instance třídy zadáme cestu k obrázku a metodou `setIconImage(Image i)` ho nastavíme jako ikonu

```
ImageIcon i=new ImageI-
con("C:\\Tomas\\ikona.png");       this.setIconImage(i.getImage());
```

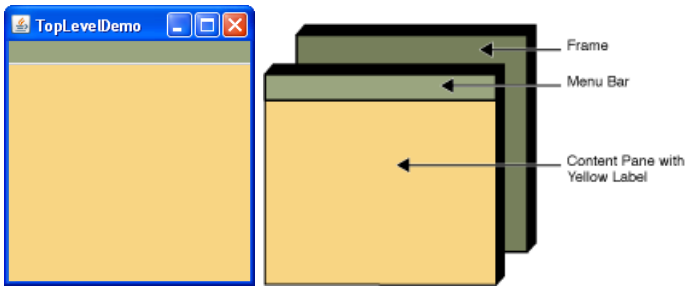
Výsledek je znázorněn na obr. 10.3.

Vytvoření druhého frame. V aplikace je možné vytvořit i další frame. Není to typické, aplikace by měla mít pouze jeden frame a např. okna zobrazovat formou dialogu. Takto vzniklému framu je nutné také ošetřit události při jeho zavírání. Pokud bychom tak neučinili, bylo by ho možno zavřít pouze zavřením prvního okna.

Obrázek 10.13: Ukázka formuláře s ikonou.

10.1.3 Přidávání dalších komponent na formulář

Každá top level-komponenta má zvláštní kontejner zvaný *content pane* obsahující všechny grafické komponenty "ležící" na top level komponentě. Tvoří ho pouze část část komponenty, podívejme se na následující obrázek. Znázorňuje formulář s komponentou Menu. Content pane představuje vnitřní část formuláře bez titulkového pruhu a menu.



Obrázek 10.14: ContentPane a jeho vztah k ostatním částem formuláře.

Obsah content pane je dostupný prostřednictvím metody `getContentPane()`. Do kontejneru content pane můžeme další komponenty přidávat metodou `add()`. Grafické komponenty jsou na top level komponenty přidávány hierarchicky, vytvářejí stromovou strukturu. Kořen stromu vytváří top-level komponenta.

Souřadnicový systém komponent. Grafické komponenty používají vlastní souřadnicový systém. Jeho počátek je v levém horním rohu okna. Osa y směřuje dolů, je představována svislou hranu monitoru, osa x vpravo, je představována vodorovnou hranou monitoru. Hodnoty souřadnic jsou zadávány v pixelech. Velikost souřadnice x je dána hodnotou šířky komponenty, můžeme ji zjistit metodou `getWidth()`. Velikost souřadnice je dána výškou komponenty, můžeme ji zjistit metodou `getHeight()`.

Obrázek 10.15: Souřadnicový sytém komponenty.

Layout managery. Pro přidávání nových komponent jsou používány správce uspořádání, tzv. *layout managery*. Na různých platformách jsou velikosti grafických komponent různé. Layout managery zajistí, že při přechodu mezi platformami budou velikosti a poloha jednotlivých komponent automaticky přepočteny (tj. komponenty nebudou nevhodně překryty či vůči sobě posunuty) a aplikace bude vypadat neustále hezky :-). V Javě se s layout managery pracuje prostřednictvím tříd. Pro nastavení uspořádání komponent používáme příkaz

```
setLayout(LayoutManager manager).
```

Parametrem je odkaz ukazující na objekt typu `LayoutManager`. Pro práci s layout managerem je nutno importovat balík `java.awt.*`. S layout managery se seznámíme podrobněji v dalších kapitolách.

Vytvoření formuláře s tlačítkem. Pokusme se na formulář vytvořený v předchozí kapitole přidat novou komponentu, a to tlačítko `JButton` a to nejprve bez využití layout manageru. O vytvoření tlačítka se postará následující kód

```
JButton but=new JButton("AHOJ");
```

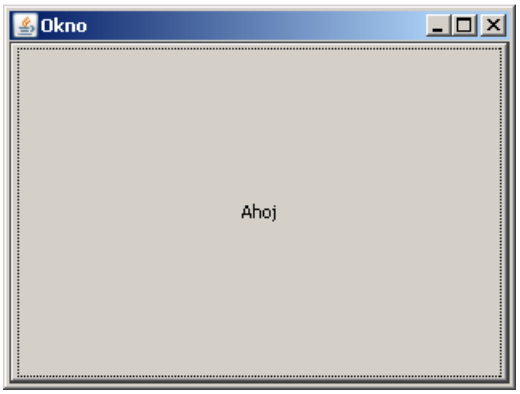
V konstruktoru provádíme inicializace prostřednictvím řetězce představujícího popis tlačítka. Metodou `getContentPane()` lze získat kontejner představující plochu okna, do kterého přidáme metodou `add( Component c)` novou komponentu.

```
o.getContentPane.add(but);
```

Celý kód bude vypadat takto

```
import javax.swing.*;
public class Komponenty {
  public Komponenty()
   {
    Okno o=new Okno();
    o.setSize(320, 240);
    o.setTitle("Moje okno");
    JButton but=new JButton("AHOJ");
    o.getContentPane.add(but);
    o.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    o.setVisible(true);
   }
 }
```

Výsledek nás poněkud překvapí, tlačítko bude zabírat celou plochu okna.



Obrázek 10.16: Okno s tlačítkem zabírajícím celou jeho plochu.

Postup zopakujeme s využitím layout manageru. Vytvoříme novou proměnou typu FlowLayout obsahující odkaz na instanci správce rozložení komponent

```
FlowLayout f=new FlowLayout();
```

Metodou `setLayout(LayoutManager lm)` nastavíme pro zvolenou komponentu tento správce rozložení

```
o.setLayout(f);
```

Oba kroky lze spojit do jednoho.

```
o.setLayout(new FlowLayout());
```

Výsledný kód vypadá takto

```
import javax.swing.*;
import java.awt.*;
public class Komponenty {
  public Komponenty()
   {
    Okno o=new Okno();
    o.setSize(320, 240);
    o.setTitle("Moje okno");
    o.setLayout(new FlowLayout());
    JButton but=new JButton("AHOJ");
    o.getContentPane().add(but);
    o.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    o.setVisible(true);
   }
 }
```

Přeložíme -li a spustíme vytvořený program, velikost tlačítka již bude normální.

Obrázek 10.17: Okno s tlačítkem "normálních" rozměrů.

Příklad: Pokusme se na formuláři vytvořit více tlačítek, která pojmenujeme. Otázkou je, jak budou tlačítka na komponentě `JFrame` uspořádána. V případě, kdy se nejvejdou do jedné řádky, pokračují na řádce další. Výsledek je patrný na následujícím obrázku. Tlačítka jsou v každé řadě vycentrována vzhledem ke komponentě, na které leží. Podrobněji se s layout managery seznámíme v kapitole 1.5.

Obrázek 10.18: Okno s vytvořenými tlačítky stejných rozměrů.

10.1.4 Práce s událostmi

Na formuláři jsme vytvořili několik tlačítek, zatím jsme jim však nepřiřadili žádnou akci. V této kapitole se seznámíme s obsluhou událostí v Javě. Jak jsme již v úvodu uvedli, událost vzniká jako důsledek činnosti uživatele, programu či operačního systému.

Listener. Každá komponenta může vyvolat řadu různých událostí. V Javě existují speciální instance, které události zachycují. Nazýváme je *listenery*, tj. posluchači. Určitý typ listeneru může naslouchat pouze jemu odpovídajícímu typu události, listenery tedy fungují adresně. Zdroje událostí jsou potomky třídy `Java.awt.Event`. Instance, která chce přijímat události, se musí nejprve jako posluchač zaregistrovat. Zaregistrovaný objekt musí implementovat rozhraní `XXXListener`. Lze tak učinit pomocí metody `addXXXListener()`, kde XXX představuje typ události.

Handler. Událost způsobí zaslání zprávy posluchači. Důsledkem je vyvolání metody ošetřující tuto událost, kterou nazýváme *handler*. V handleru následně provedeme obsluhu události, tj. nějakým způsobem na ni reagujeme.

Princip ošetření události V praxi postupujeme tak, že zjistíme název rozhraní XXX pro listener zvoleného objektu (na základě typu události, kterou budeme ošetřovat). Následně nalezneme jména metod tohoto rozhraní (jejich počet bývá různý dle typu události) a *všechny* je implementujeme. Metody nemůžeme přetěžovat, pouze předefinovat, smíme je tedy implementovat pouze *jednou*. Odkaz na instanci třídy, jejíž jméno se shoduje s typem události, předáváme jako formální parametr handleru ošetřujícímu tuto událost.

Seznam metod a rozhraní. V následující tabulce uvedeme přehled nejčastěji používaných událostí, rozhraní a jejich metod. Podrobněji se s nimi seznámíme v dalších kapitolách.

Událost/Třída	Rozhraní	Popis	Metoda
ActionEvent	ActionListener	Provedení akce, např. stisk tlačítka	actionPerformed()
ComponentEvent	ComponentListener	Změna stavu komponenty	componentHidden() componentMoved() componentResized() componentShown()
ContainerEvent	ContainerListener	Detekce přidání/odebrání komponenty	containerAdded() containerRemoved()
FocusEvent	FocusListener	Změna zaměření komponenty	focusGained() focusLost()
ItemEvent	ItemListener	Vybrání komponenty	itemStateChanged()
KeyEvent	KeyListener	Práce s klávesnicí	keyPressed() keyReleased() keyTyped()
MouseEvent	MouseListener	Práce s tlačítky myši	mouseClicked() mouseEntered() mouseExited() mousePressed() mouseReleased()
MouseMotionEvent	MouseMotionListener	Pohyb myši	mouseMotionDragged() mouseMotionMoved()
MouseWheelEvent	MouseWheelListener	Práce s rolovacím kolečkem myši	mouseWheelMoved()
WindowEvent	WindowListener	Změna stavu okna	windowActivated() windowClosed() windowClosing() windowDeactivated() windowOpened()
WindowFocusEvent	WindowFocusListener	Změna zaměření okna	windowFocusGained() WindowFocusLost()

Tabulka 10.1: Přehled událostí, tříd, rozhraní a metod.

Tento na první pohled poněkud "krkolomný" avšak poměrně logický přístup naznačíme na praktickém příkladě. Budeme provádět obsluhu činnosti tlačítka, které jsme vytvořili v předcházející kapitole. Na formulář

přidáme novou komponentu `JLabel` představující poznámku. V konstruktoru je uveden text, který má být zobrazen.

```
JLabel lab=new JLabel ("Text");
o.getContentPane().add(lab);
```

Po stisknutí tlačítka zobrazíme text s vhodnou informací. V tomto případě použijeme rozhraní ActionListener. Nejprve musíme pro třídu `Okno` implementovat rozhraní `ActionListener`. Pro práci s událostmi je nutno importovat balíček java.awt.event.* s příslušnými událostmi.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Okno extends JFrame implements ActionListener{
...
}
```

Poté prostřednictvím metody `addActionListener()` zaregistrujeme pro příslušnou instanci (tj. komponentu) posluchače. Tento krok je vhodný provést v konstruktoru třídy `Komponenty`.

```
but.addActionListener(this);
```

Nyní vytvoříme metodu `actionPerformed()` ošetřující událost `Action` komponenty `but` třídy `JButton`.

```
public void actionPerformed(ActionEvent e)
 {
  lab.setText("Stisknuto tlacitko");
 }
```

Výsledný kód bude vypadat takto:

```
public class Okno extends JFrame implements ActionListener{
    private JButton but;
    private JLabel lab;

    public Okno(int sirka, int vyska) {
     this.setSize(sirka,vyska);
     this.setTitle("Okno");
     this.setLayout(new FlowLayout());
     but=new JButton("Tlacitko");
     lab=new JLabel("Text");
     this.getContentPane().add(but);
     this.getContentPane().add(lab);
     but.addActionListener(this);
     this.setVisible(true);
    }
  public void actionPerformed(ActionEvent e)
  {
   lab.setText("Stisknuto tlacitko");
  }
}
```

Po stisknutí tlačítka bude výsledek vypadat takto.

Obrázek 10.19: Vlevo výchozí formulář, vpravo formulář po stisknutí tlačítka.

Důležité upozornění. Pokud program obsahuje anonymní vnitřní třídu, vznikají při jeho překladu soubory obsahující před příponou class znak $, které obsahují přeložené kódy anonymních vnitřních tříd. Nezapomeňme proto při distribuci našeho programu tyto třídy přiložit.

10.1.4.1 Ošetření stejné události u více komponent

V programu se často vyskytuje více komponent, které generují stejné typy událostí. Na každou z událostí zpravidla budeme chtít reagovat jinak. Typickým příkladem jsou různá tlačítka, comboboxy či radiobuttony.

Komponenty generují stejný typ události, handler, který je ošetřuje, nemůže být přetížen, může být implementován pouze jednou. Metoda `actionPerformed()` se ve třídě smí vyskytnout pouze jednou. Doporučené řešení v tomto případě představuje použití *vnitřních tříd.* popř. *anonymních vnitřních tříd.* My se seznámíme s druhým přístupem, který je využíván ve většině nástrojů pro RAD (JBuilder, NetBeans,...).

Příklad s jedním tlačítkem. Rozhraní, které je možno implementovat, jsou uvedena v tabulce v úvodu kapitoly 1.4. Budeme se snažit ošetřit událost `ActionEvent` rozhraní `ActionListener`. Rozhraní již nebude implementováno třídou Okno, ale nějakou anonymní vnitřní třídou. Toto rozhraní obsahuje metodu `actionPerformed()`, která bude uživatelem předefinována. Nejprve provedeme registraci posluchače.

```
but.addActionListener(new ActionListener()
  {
     public void actionPerformed(ActionEvent e)
       {
         tlStisknuto(e);
       }
   });
```

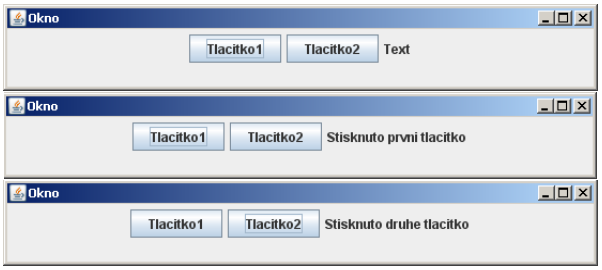
Metodě `addActionListener()` předáváme jako parametr instanci anonymní třídy implementující rozhraní `ActionListener`. Metoda `tlStisknuto()`představuje *handler*, který je vyvolán, pokud dojde k příslušné události. V našem případě obsahuje pouze kód, který do komponenty `JLabel` umístí informaci o tom, že tlačítko bylo stisknuto.

```
public void tlStisknuto(ActionEvent e)
   {
    lab.setText("Stisknuto tlacitko");
   }
```

Příklad se dvěma tlačítky. Stejným způsobem budeme postupovat i v případě, kdy se budeme snažit rozlišit, které z tlačítek bylo stisknuto. Vytvoříme dvě tlačítka, zaregistrujeme jim posluchače a u obou předefinujeme metody `actionPerformed()`. Uvedeme celý zdrojový kód.

```
public class Okno2 extends JFrame {
    private JButton but1, but2;
    private JLabel lab;

    public Okno2(int sirka, int vyska)
    {
     this.setSize(sirka,vyska);
     this.setTitle("Okno");
     this.setLayout(new FlowLayout());
     but1=new JButton("Tlacitko1");
     but2=new JButton("Tlacitko2");
     lab=new JLabel("Text");
     this.getContentPane().add(but1);
     this.getContentPane().add(but2);
     this.getContentPane().add(lab);
     but1.addActionListener(new ActionListener()
       {
          public void actionPerformed(ActionEvent e)
            {
              tl1Stisknuto(e);
            }
       });
      but2.addActionListener(new ActionListener()
      {
        public void actionPerformed(ActionEvent e)
        {
         tl2Stisknuto(e) ;
        }
      });
       this.setVisible(true);
    }
   public void tl1Stisknuto(ActionEvent e)
   {
    lab.setText("Stisknuto prvni tlacitko");
   }
   public void tl2Stisknuto(ActionEvent e)
   {
    lab.setText("Stisknuto druhe tlacitko");
   }
}
```

Podobný kód je generován prostřednictvím RAD nástrojů. Výsledek vypadá takto:



Obrázek 10.20: Vlevo výchozí formulář, uprostřed po stisknutí levého tlačítka, vpravo po stisknutí pravého tlačítka.

10.1.4.2 Několik posluchačů téže události

Často potřebujeme, aby jedna událost měla více posluchačů. Např. při stisku tlačítka potřebujeme, aby se provedlo několik různých činností. V tomto případě nemusíme rozlišovat zdroj akce, není třeba používat anonymní vnitřní třídy jako v předchozím případě.

```
public Okno2(int sirka, int vyska)
    {
     this.setSize(sirka,vyska);
     this.setTitle("Okno");
     this.setLayout(new FlowLayout());
     but1=new JButton("Tlacitko1");
     but2=new JButton("Tlacitko2");
     lab=new JLabel("Text");
     this.getContentPane().add(but1);
     this.getContentPane().add(but2);
     this.getContentPane().add(lab);
     but1.addActionListener(but1);
     but1.addActionListener(but2);
    ...
   }
```

10.1.5 Barvy, fonty, grafický vzhled komponent

V této kapitole se seznámíme se základními informacemi o barvách a fontech v jazyce Java.

10.1.5.1 Barvy v Javě

Java pracuje s modelem RGB, který používá tři základní barevné složky: červenou (Red), zelenou (Green), modrou (Blue). Každá ze složek může nabývat intenzitu v intervalu <0,255>. Existuje tedy 256^3různých barevných odstínů. Pro práci s barvami je použita třída `java.awt.Color`. V Javě je předdefinováno 13 nejčastěji používaných barevných odstínů, představují objekty třídy `Color`. Jejich přehled je uveden v následující tabulce.

Barva	Popis
Color.black	černá
Color.white	bílá
Color.red	červená
Color.green	zelená
Color.blue	modrá
Color.yellow	žlutá
Color.cyan	azurová
Color.magenta	fialová
Color.orange	oranžová
Color.pink	růžová
Color.lightGray	světle šedá
Color.gray	šedá
Color.darkGrey	tmavě šedá.

Chceme -li zkonstruovat barvu vlastní, je nutno vytvořit instanci třídy Color. V konstruktoru uvedeme hodnoty jednotlivých RGB složek.

```
Color zelena=new Color (140,140,140);
```

Každá z komponent má popředí (foreground) a pozadí (background). Pro popředí i pozadí lze nastavit barvu nebo tyto údaje získat. Barva popředí představuje barvu textu, s níž je komponenta popsána, barva pozadí pak barvu vlastní komponenty. Pro práci s barvou popředí jsou použity metody `setForeground(Color col)` a `getForeground()`, pro práci s pozadím metody `setBackground(Color col)` a `getBackground()`.

V prvním příkladu se pokusíme změnit barvu pozadí tlačítka a jeho popisu na některý z předdefinovaných odstínů.

```
public class Okno3 extends JFrame{
    public Okno3(int sirka, int vyska) {
      this.setSize(sirka, vyska);
      this.setTitle("Okno");
      JButton b=new JButton("Pokus");
      this.getContentPane().add(b);
      this.setLayout(new FlowLayout());
      b.setBackground(Color.blue);
      b.setForeground(Color.red);
      this.setVisible(true);
    }
  }
```

Výsledek vypadá takto

Obrázek 10.21: Barva popředí tlačítka: červená, barva pozadí tlačítka: modrá.

V dalším příkladu se pokusíme změnit barvu tlačítka na vlastní hodnotu. Nepoužijeme předdefinovaný odstín, vytvoříme vlastní objekt třídy Color.

```
b.setBackground(new (140,140,140));
b.setForeground(new Color(150,250,180));
```

Hodnoty barevných složek Často potřebujeme zjistit hodnoty jednotlivých barevných složek. K tomu slouží tři metody: `getRed()`, `getGreen()`, `getBlue()`. Výsledkem jsou hodnoty typu `int`.

10.1.5.2 Fonty v Javě

Java může běžet na různých operačních systémech, na každém z nich se mohou vyskytovat jiné typy fontů. Java proto používá systém tzv. symbolických fontů, které jsou nezávislé na platformě. Existuje pět základních skupin fontů (tzv. rodin), od kterých lze odvodit 4 řezy (`plain, bold, italic, bolditalic`). Jejich přehled a srovnání s fonty OS Windows je uveden v následující tabulce.

OS Windows	Java
TimesNewRoman	Serif
Arial	SansSerif
CourierNew	Monospaced
Arial	Dialog
CourierNew	DialogInput

Tabulka 10.2: Přehled základních fontů a jejich řezů.

Pro práci s fonty Java používá třídu `java.awt.Font`. Údaje o fontu se nacházejí v instanci třídy `Font`. Její konstruktor vypadá takto.

```
Font f =new Font("Serif", Font.bold,10);
```

Získání informací o fontu. Pro získání informací o fontu použijeme metodu `getFont()` vracející hodnotu třídy font. Na ní lze aplikovat řadu dalších metod: `getFamily()`, `isPlain()`, `isBold()`, `isItalic()`.

10.1.5.3 Základní vlastnosti komponent

V této kapitole se seznámíme se základními vlastnostmi, které jsou společné většině komponent.

Zjištění velikosti a polohy komponenty. Pro zjištění velikosti komponenty slouží metody `getWidth()` a `getHeight()`. Výsledkem jsou hodnoty typu `int`. Existuje i metoda `getSize()` vracející obě hodnoty současně. Polohu komponenty vzhledem k levému hornímu rohu okna můžeme určit prostřednictvím metod `getX()` a `getY()`. Do této hodnoty se nepočítá výška titulkového pruhu formuláře. Velikost komponenty můžeme změnit prostřednictvím metody `setSize(int width, int height)`, polohu komponenty prostřednictvím `setLocation()`. Existují i jejich varianty s get, `getSize()` a `getLocation()`.

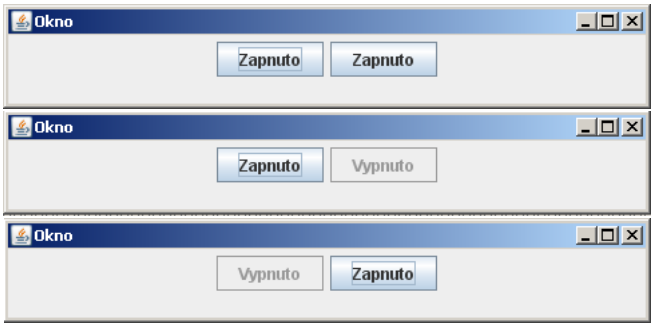
Nastavení extrémní velikosti komponenty Každé z komponent je možno nastavit minimální, maximální, či preferovanou velikost. Slouží k tomu metody `setMinimumSize(int width, int height)`, `setMaximumSize( int width, int height)`, `setPrefferedSize(int width, int height)`. Preferovaná velikost komponenty je výchozí velikost komponenty, defaultně je nastavena jako minimální velikost komponenty. Pro zjištění výše nastavených údajů slouží metody `getMinimumSize()`, `getMaximumSize()`, `getPrefferedSize()`.

Zpřístupnění/znepřístupnění komponenty. Komponenta se může vyskytovat ve dvou základních stavech: komponenta je aktivní, jsou nad ní generovány události, označujeme ji jako přístupnou. V opačném případě je komponenta nepřístupná a tudíž neaktivní, je znázorněna šedou barvou. Tato vlastnost komponenty je používána, pokud chceme uživateli zakázat/povolit nějakou akci. Není vhodné komponenty odebírat, nepřispívá to k přehlednosti programu. Uživatel bude zmaten, kam zmizely jeho oblíbené nástroje. Pro zpřístupnění/znepřístupnění komponenty je používána metoda `setEnabled(boolean status)`. Metoda má jako parametr hodnotu typu `boolean`. Zjištění stavu, zda je komponenta přístupná či ne, použijeme příkaz `getEnabled()`.

Podívejme se na jednoduchý program, který ilustruje přístupnost komponenty na příkladu dvou tlačítek. Po stisknutí jednoho z nich se druhé tlačítko stane nepřístupným. Opakovaným stisknutím tlačítka se druhé tlačítko opět stává aktivním. Výsledný kód může vypadat např. takto.

```
public class Okno4 extends JFrame{
    private JButton b1, b2;
    public Okno4(int sirka, int vyska) {
        this.setSize(sirka, vyska);
        this.setTitle("Okno");
        this.setLayout(new FlowLayout());
        b1=new JButton();
        b2=new JButton();
        b1.setText("Zapnuto");
        b2.setText("Zapnuto");
        this.getContentPane().add(b1);
        this.getContentPane().add(b2);
        this.setVisible(true);
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                        tl1Stisknuto(e);
            }
        });
        b2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e ) {
                tl2Stisknuto(e);
              }
         });
        }
       public void tl1Stisknuto(ActionEvent e)
        {
         b2.setEnabled(!b2.isEnabled());
         if (b2.isEnabled()) b2.setText("Zapnuto");
         else b2.setText("Vypnuto");
        }
       public void tl2Stisknuto(ActionEvent e)
        {
         b1.setEnabled(!b1.isEnabled());
         if (b1.isEnabled()) b1.setText("Zapnuto");
         else b1.setText("Vypnuto");
        }
    }
```

A výsledek:



Obrázek 10.22: Ukázky zpřístupnění a znepřístupňování komponent.

Skrytí/zobrazení komponenty. Komponentu je možno zobrazit či skrýt. Slouží k tomu metoda `setVisible(boolean status)` s parametrem typu boolean. Stav komponenty lze zjistit metodou `isVisible()`.

Obrázek 10.23: Skrytí komponenty.

10.1.6 Layout managery

Při rozmist'ování komponent na formuláře (či jiné komponenty) zpravidla v Javě nedefinujeme polohu komponent absolutně. Důvodem je fakt, že Java může běžet na různých prostředích, kde jednotlivé grafické komponenty nemusejí mít stejnou velikost či tvar. Takový program by na některých platformách nevypadal vždy pěkně, mohlo dojít např. k překrytům některých jeho grafických komponent, jiné by se na formulář nemusely vejít celé.

Místo toho používáme tzv. *layout managery* (správce uspořádání), které jsou zodpovědné za rozmístění komponent. Při přechodu mezi různými prostředími provádějí přepočet polohy a tvaru komponent. Existuje pět typů správců uspořádání, každá z komponent má svůj vlastní implicitní správce uspořádání; ten však můžeme změnit za použití metody `setLayout(LayoutManager lm)`.

Typy layout managerů. LayoutManager představuje objekt třídy `java.awt.LayoutManager`, tvoří ji několik dalších tříd. Komplexnější popis této problematiky přesahuje rozsah tohoto materiálu, podíváme se na tři nejčastěji používané layout managery:

- FlowLayout
- GridLayout
- BorderLayout

První dva layout managery jsou poměrně jednoduché, třetí umožňuje provádět pokročilejší rozvržení komponent.

Okno a kontejner. Připomeňme, že každé okno obsahuje kontejner, jehož obsah můžeme získat metodou `getContentPane()`. Do kontejneru můžeme přidávat komponenty metodou `add(Component c)`.

Návrh vzhledu aplikace. S použitím layout managerů můžeme navrhovat i značně vzhledově složitá grafická rozhraní tvořená velkým množstvím komponent. Tento postup je však poměrně náročný, vytvoření takového rozhraní zabere spoustu času. Je proto vhodné ho provádět některým z nástrojů pro RAD obsahujícího tzv. GUI buildery. Do této skupiny patří např. JBuilder nebo NetBeans.

V praxi se provádí při "ručním" návrhu rozdělení okna na několik různých částí, které obsahují malé množství komponent. Vzhled každé takto vzniklé části je navrhován samostatně. Často nevystačíme s jedním layout managerem, můžeme je proto vzájemně kombinovat. My budeme činnost layout managerů ilustrovat pouze na jednoduchých příkladech.

10.1.6.1 FlowLayout

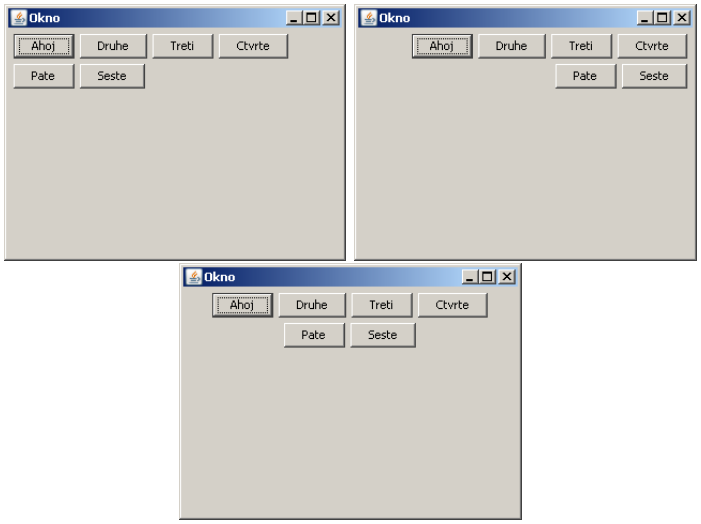
FlowLayout představuje nejjednodušší layout manager. V předcházejících kapitolách jsme se s ním již seznámili. Připomeňme si, že jednotlivé komponenty uspořádává do řádky za současného vycentrování; mezi komponentami jsou ponechány mezery. Pokud se již komponenty na řádek nevejdou, pokračuje se na dalším řádku. Výška řádku je dána nejvyšší komponentou. Komponenty mohou být zarovnány také nalevo nebo napravo. K dispozici je několik konstruktorů

```
FlowLayout();
FlowLayout(typ_zarovnani);
FlowLayout(typ_zarovnani, vod_mezera, svisla_mezera);
```

Typ zarovnání lze zvolit prostřednictvím konstant: `FlowLayout.CENTER`, `FlowLayout.RIGHT`, `FlowLayout.LEFT`. Podívejme se na následující příklad.

```
public class Okno extends JFrame{
    public Okno() {
     this.setSize(320,240);
     this.setTitle("Okno");
     this.setLayout(new FlowLayout(FlowLayout.RIGHT,5,5));
     JButton but1=new JButton("Ahoj");
     this.getContentPane().add(but1);
     JButton but2=new JButton("Druhe");
     this.getContentPane().add(but2);
     JButton but3=new JButton("Treti");
     this.getContentPane().add(but3);
     JButton but4=new JButton("Ctvrte");
     this.getContentPane().add(but4);
     JButton but5=new JButton("Pate");
     this.getContentPane().add(but5);
     JButton but6=new JButton("Seste");
     this.getContentPane().add(but6);
     this.setVisible(true);
    }
```

Změnou typu zarovnání dostaneme následující výsledek.



Obrázek 10.24: Změna zarovnání komponent při Flow Layoutu.

10.1.6.2 GridLayout

GridLayout představuje layout manager, který jednotlivé komponenty zarovnává do mřížky tvořené stejně velkými poli. Komponenta se pomyslně rozdělí na stejně velké částí tvořené rovnoběžkami z hranami formuláře, do každého dílu je umístěna jedna komponenta. Pokud jsou všechna políčka v řádku zaplněna, přechází se na další řádek. Velikost jednoho políčka je nastavena na základě největší komponenty. Uveďme opět některé důležité konstruktory.

```
GridLayout();
GridLayout(radky, sloupce);
GridLayout(radky, sloupce, vod_mezera, svisla_mezera);
```

První konstruktor umožní vložit pouze jednu komponentu roztaženou přes celé okno. Pokud použijeme druhý konstruktor, budou se jednotlivé komponenty dotýkat, tj. mezery mezi nimi budou stejné. Použijeme stejný kód jako v předchozím případě, pouze změníme layout manager. V prvním případě použijeme konstruktory `GridLayout(3,2)`, ve druhém případě `GridLayout(3,2,10,10)`. Výsledky budou vypadat takto.

Obrázek 10.25: Použití dvou konstruktorů třídy GridLayout.

10.1.6.3 BorderLayout

Border Layout představuje layout manager, který jednotlivé komponenty umist'uje podle světových stran na základě hodnot konstant `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.EAST`, `BorderLay-`

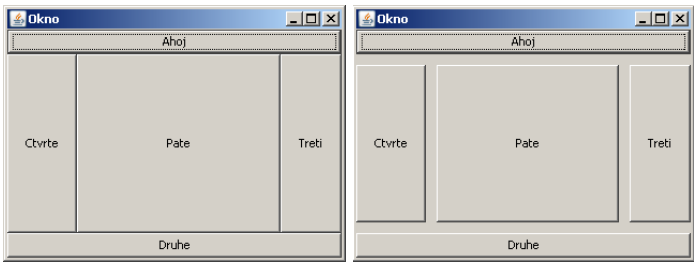
`out.WEST`, `BorderLayout.CENTER`. Komponenty nejsou, na rozdíl od předcházejících správců, řazeny podle podle pořadí. Velikosti jednotlivých komponent jsou různé: severní a jižní komponenta jsou roztaženy na maximální šířku, severní a jižní komponenty na maximální výšku, velikost středové komponenty se nemění. Použití nalezne např. při navrhování ovládacích tlačítek pro pohyb kurzoru. K dispozici jsou dva konstruktory:

```
BorderLayout();
BorderLayout(vod_mezera, svisla_mezera);
```

Při použití prvního konstruktoru nejsou mezi jednotlivými komponentami mezery, u druhého konstruktoru lze nastavit vodorovné a svislé rozestupy mezi komponentami. Zdrojový kód lze zapsat např. takto

```
public Okno() {
    this.setSize(320,240);
    this.setTitle("Okno");
    this.setLayout(new GridLayout());
    this.setLayout(new BorderLayout(10,10));
    JButton but1=new JButton("Ahoj");
    this.getContentPane().add(but1, BorderLayout.NORTH);
    JButton but2=new JButton("Druhe");
    this.getContentPane().add(but2, BorderLayout.SOUTH);
    JButton but3=new JButton("Treti");
    this.getContentPane().add(but3, BorderLayout.EAST);
    JButton but4=new JButton("Ctvrte");
    this.getContentPane().add(but4, BorderLayout.WEST);
    JButton but5=new JButton("Pate");
    this.getContentPane().add(but5, BorderLayout.CENTER);
    this.setVisible(true);
   }
```

Výsledek vypadá takto.



Obrázek 10.26: Použití dvou konstruktorů třídy BorderLayout.

Poznámka: `CardLayout` manager nebudeme z důvodu omezené použitelnosti uvádět, `GridBackLayout` manager je poměrně složitý. Zájemce se s nimi může seznámit v dostupné literatuře. Pro vývoj složitějších GUI aplikací budeme používat nástroje RAD, nemusíme tak ztrácet čas a energii při návrhu grafického vzhledu, ušetřený čas a prostředky mohou být věnovány na zdokonalování aplikace.

10.1.7 Modely v Javě

V GUI Javy je často využívaným postupem práce s *modelem*. Model představuje implementaci nějakého rozhraní jinou (např. grafickou) třídou. Přes toto rozhraní přistupuje grafická třída ke svým metodám. Použití modelů umožňuje důsledně oddělit grafický návrh aplikace od funkční části.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

Modely se používají nejen pro práci s grafikou, setkáme se s nimi i při manipulacích s běžnými komponentami. Koncept rozhraní v Javě je tvořen třemi skupinami rozhraní:

- Základní rozhraní.
- Abstraktní implementace rozhraní.
- Výchozí implementace rozhraní.

Základní rozhraní pro práci s modely představují např. `TableModel`, `ListModel`, ... Abstraktní implementace (implementace rozhraní abstraktní třídou) používáme v případech, kdy chceme předefinovávat funkcionalitu rozhraní (např. `AbstractTableModel`), výchozí implementaci rozhraní (implementace rozhraní neabstraktní třídou) v případech, kdy funkcionalitu rozhraní neměníme např. `DefaultTableModel`). Problematika modelů v Javě jde nad rámec základního kurzu, proto se o ní zmiňujeme pouze velmi stručně.

Modely umožňují také pracovat s událostmi. Pokud dojde k jakékoliv změně v datech modelu, je tato informace oznámena všem zaregistrovaným posluchačům. Modely se používají často při práci s následujícími komponentami:

- JList (Seznam)
- JTable (Tabulka)
- JTree (Strom)

Uveďme pro úplnost, že s těmito komponentami můžeme pracovat i "klasicky", tj. bez použití modelů.

10.1.8 Přehled nejpoužívanějších komponent

V této kapitole se seznámíme s nejčastěji používanými vizuálními komponentami a jejich událostmi. Jsou potomky třídy `javax.Swing.*`, kromě této třídy musíme připojit i třídu `java.awt.event.*`. Název komponenty začínající na písmeno J je současně i názvem třídy, kterou komponenta představuje.

Zopakujme, že starší komponenty z rozhraní AWT mají podobné názvy, pouze nezačínají písmenkem J. Společnou vlastností všech komponent popsaných v kapitolách 10.8.1-10.8.8 je to, že nemohou být zobrazeny samostatně (představují základní komponenty). Není možné vytvořit program, který zobrazí pouze jednu tuto komponentu (např. `JLabel`) bez hlavního okna.

10.1.8.1 JLabel

Tato komponenta je používána jako popis. Neobsahuje -li žádný text, je neviditelná. Komponentě lze nastavit barvu popředí nebo barvu pozadí. Standardně je v ní text zarovnán směrem vlevo. Většinou je používána jako komponenta "statická", kdy zpravidla neošetřujeme jeho události, ale pouze s ním popisujeme jiné komponenty. Existují tři základní konstruktory třídy `JLabel`.

```
JLabel();
JLabel(String text);
JLabel (String text, JLabel alignment);
```

Zarovnání lze ovlivnit třemi konstantami: `JLabel.LEFT`, `JLabel.RIGHT`, `JLabel.CENTER`, `JLabel.LEADING`, `JLabel.TRAILING`. Přehled nejpoužívanějších metod spolu se stručným popisem jsou umístěny v následující tabulce.

Metoda	Popis
`setText()`	Nastavení textu zobrazovaného labelem
`getText()`	Získání textu zobrazovaného labelem.
`setHorizontalAlignment()`	Nastavení horizontálního zarovnání textu v labelu.
`getHorizontalAlignment()`	Získání horizontálního zarovnání textu v labelu.
`setVerticalAlignment()`	Nastavení vertikálního zarovnání textu v labelu.
`getVerticalAlignment()`	Získání vertikálního zarovnání textu v labelu.

Tabulka 10.3: Přehled metod třídy JLabel.

Praktické příklady uvádět nebudeme, použití `JLabelu` je triviální. Navíc jsme si práci s touto komponentou v předchozím textu již vyzkoušeli.

10.1.8.2 JButton

JButton neboli tlačítko představuje jednu z nejpoužívanějších komponent, zpravidla slouží ke spouštění nějaké akce. K dispozici jsou dva základní konstruktory:

```
JButton();
JButton(String text);
```

Tlačítko po svém stisknutí generuje událost, kterou je nutné odchytit prostřednictvím rozhraní `ActionListener` za použití metody `actionPerformed`. Předhled metod používaných u tlačítka.

Metoda	Popis
`setText()`	Nastavení popisu tlačítka.
`getText()`	Získání popisu tlačítka.

Tabulka 10.4: Přehled metod třídy JButton.

10.1.8.3 JCheckBox

`JCheckBox` představuje zaškrtávací pole, může nabývat logické hodnoty true/false (realizované zaškrtnutím/odškrtnutím). CheckBoxy je možno seskupovat, komponenty jsou na sobě nezávislé; mohou být zaškrtnuty/odškrtnuty v libovolné kombinaci. Základní konstruktory vypadají takto:

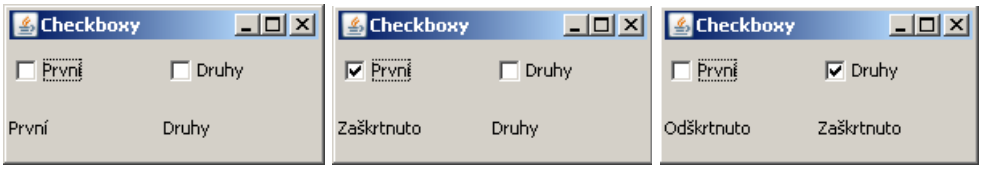
```
JCheckBox();
JCheckBox(String text);
JCheckBox(String text, boolean status);
```

První konstruktor vytvoří `JCheckBox` bez popisu a bez zaškrtnutí, druhý s popisem a bez zaškrtnutí, třetí s popisem a možností volby zaškrtnutí. Přehled metod používaných u check boxů nalezeneme v následující tabulce

Metoda	Popis
`setText( text)`	Nastavení popisu checkboxu.
`getText()`	Získání popisu checkboxu.
`setSelected(boolean status)`	Nastavení stavu checkboxu.
`isSelected()`	Získání stavu checkboxu.

Tabulka 10.5: Přehled metod třídy JCheckBox.

Při změně stavu checkboxu je generována událost, kterou lze odchytit prostřednictví rozhraní `ItemListener` za použití metody `itemStateChanged()`. Podívejme se na následující příklad, který bude zobrazovat stav zaškrtnutí dvou `JCheckBoxů` ve dvou `JLabelech`. Výsledek vypadá takto.



Obrázek 10.27: Ukázka práce s checkboxy.

Zdrojový kód příkladu, využíváme GridLayout.

```
public class Checkboxy extends JFrame {
   private JCheckBox c1, c2;
   private JLabel l1,l2;
    public Checkboxy(int vyska, int sirka) {
     this.setSize(vyska, sirka);
     this.setTitle("Checkboxy");
     c1=new JCheckBox("První");
     c2=new JCheckBox("Druhy");
     l1=new JLabel(c1.getText());
     l2=new JLabel(c2.getText());
     this.getContentPane().add(c1);
     this.getContentPane().add(c2);
     this.getContentPane().add(l1);
     this.getContentPane().add(l2);
     this.setLayout(new GridLayout(2,2));
     c1.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
              l1Changed(e);
            }
        });
     c2.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                l2Changed(e);
            }
        });
    }
    public void l1Changed(ItemEvent e)
    {
```

```
        if(c1.isSelected()) l1.setText("Zaškrtnuto");
        else l1.setText("Odškrtnuto");
       }

       public void l2Changed(ItemEvent e)
       {
        if(c2.isSelected()) l2.setText("Zaškrtnuto");
        else l2.setText("Odškrtnuto");
       }
}
```

10.1.8.4 JRadioButton

Tato komponenta se svým tvarem podobá `JCheckBoxu`, tlačítko je však kulaté. Od check boxů se odlišuje způsobem výběru, v případě skupiny radio buttonů může být v jednom okamžiku vybráno pouze jedno tlačítko. Radiobuttony mohou být používány samostatně nebo sdružovány do skupiny prostřednictvím komponenty `JButtonGroup`. Pokud umístíme radio buttony na formulář, aniž je sdružíme do skupiny, mohou být vybírány nezávisle na sobě (nevědí vzájemně o svém stavu). To, že je sdružíme do skupiny, způsobí, že vybraný radiobutton "informuje" ostatní radio buttony skupiny, takže již nemohou být následně vybrány. K dispozici je větší množství konstruktorů, uveďme tyto:

```
JRadioButton();
JRadioButton(String text);
JRadioButton(String text, boolean selected);
```

První konstruktor vytvoří prázdný nevybraný radio button, druhý konstruktor nevybraný radio button s popisem, třetí konstruktor radio button s popisem, u kterého můžeme zvolit výchozí stav. V praxi postupujeme nejčastěji tak, že vytvoříme požadovaný počet radio buttonů a následně každý z nich přiřadíme do skupiny. Přehled nejčastěji používaných metod u `JRadioButonů`.

Metoda	Popis
`isSelected()`	Získání stavu checkboxu.
`setSelected()`	Nastavení stavu checkboxu.

Tabulka 10.6: Přehled metod třídy JRadioButton.

Při práci s třídou `ButtonGroup` je k dispozici jediný konstruktor

```
ButtonGroup();
```

vytvářejí prázdnou skupinu radiobuttonů. Nejpoužívanější metody při práci s třídou `ButtonGroup`.

Metoda	Popis
`add(AbstractButton b)`	Přidání tlačítka do skupiny.
`remove(AbstractButton b)`	Odstranění tlačítka ze skupiny.
`getButtonCount()`	Zjistí počet kompnent ve skupině.
`getSelection()`	Vrací odkaz na zvolené tlačítko.
`isSelected(ButtonModel m)`	Získání stavu tlačítka.
`setSelected(ButtonModel m, boolean status)`	Nastavení stavu tlačítka.

Tabulka 10.7: Přehled metod třídy JButtonBroup.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

Při změně stavu radiobuttonu je generována událost, kterou lze odchytit prostřednictví rozhraní `ItemListener` za použití metody `itemStateChanged()`.

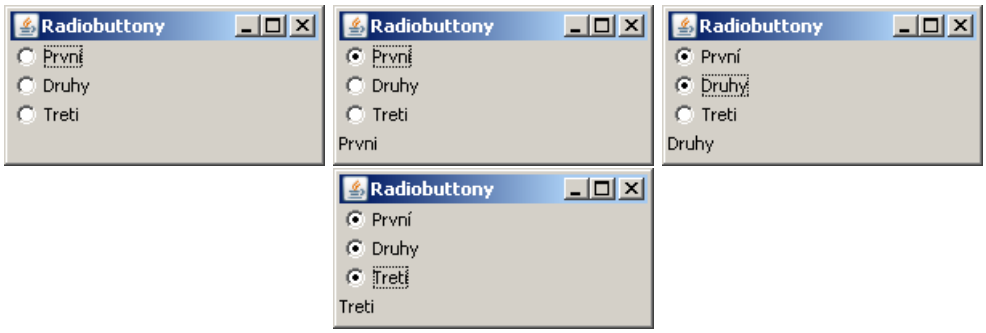
Práci s radiobuttony si ukážeme na následujícím příkladu: na formuláři bude trojice radiobuttonů sdružených v buttongroup a jeden label informující, které tlačítko bylo stlačeno. Nové radiobuttony vytvoříme pomocí konstrukce

```
JRadioButton rb1=new JRadioButton("Prvni");
JRadioButton rb2=new JRadioButton("Druhy");
JRadioButton rb3=new JRadioButton("Treti");
```

Požadujeme, aby mohlo být stisknuto pouze jedno z těchto tlačítek. Vytvoříme proto nový buttongroup a přidáme do něj vytvořené radiobuttony.

```
ButtonGroup bm=new ButtonGroup();
bm.add(rb1);
bm.add(rb2);
bm.add(rb3);
```

Výsledek vypadá takto



Obrázek 10.28: Práce s radio buttony.

Zdrojový kód k příkladu.

```
public class Radiobuttony extends JFrame {
   private JRadioButton rb1, rb2, rb3;
   private JLabel l;
    public Radiobuttony(int vyska, int sirka) {
     this.setSize(vyska, sirka);
     this.setTitle("Radiobuttony");
     rb1=new JRadioButton ("První");
     rb2=new JRadioButton ("Druhy");
     rb3=new JRadioButton ("Treti");
     l=new JLabel();
     this.getContentPane().add(rb1);
     this.getContentPane().add(rb2);
     this.getContentPane().add(rb3);
     this.getContentPane().add(l);
```

```
        this.setLayout(new GridLayout(4,1));

        rb1.addItemListener(new ItemListener() {
              public void itemStateChanged(ItemEvent e) {
                 rb1Changed(e);
               }
           });

        rb2.addItemListener(new ItemListener() {
               public void itemStateChanged(ItemEvent e) {
                   rb2Changed(e);
               }
           });

        rb3.addItemListener(new ItemListener() {
               public void itemStateChanged(ItemEvent e) {
                   rb3Changed(e);
               }
           });
     }

    public void rb1Changed(ItemEvent e)
     {
     if(rb1.isSelected()) l.setText("Prvni");
     }

    public void rb2Changed(ItemEvent e)
     {
     if(rb2.isSelected()) l.setText("Druhy");
     }

    public void rb3Changed(ItemEvent e)
     {
     if(rb3.isSelected()) l.setText("Treti");
     }
}
```

10.1.8.5 JTextField

Komponenta představuje editační políčko, do kterého mohou být zadávány jednořádkové údaje z klávesnice. Potvrzení údajů je provedeno stiskem klávesy Enter. Zadávaný text může být delší než šířka `JTextFieldu`. Komponenta se používá, pokud potřebujeme od uživatele získat vstupní údaje nutné pro běh programu. Každý text field by měl být popsán pomocí labelu tak, aby jeho význam byl jasný (v programu by se neměly vyskytovat nepopsané text fieldy). K dispozici jsou následující konstruktory.

```
JTextField();
JTextField(String text);
```

První konstruktor vytvoří prázdný textfield, druhý konstruktor textfield obsahující zadaný text. Přehled často používaných metod komponenty `JTextField`.

Metoda	Popis
`setText(String text)`	Nastavení popisu.
`getText()`	Získání popisu.
`setEditable(bool b)`	Nastavení, zda komponenta je/není read only.

Tabulka 10.8: Přehled metod třídy JBTextField.

Po stisku klávesy Enter je generována událost, kterou lze odchytit prostřednictví rozhraní `ActionListener` za použití metody `actionPerformed()`. Práce s `JTextFieldem` bude ilustrována na příkladu, kdy do dvou vygenerovaných text fieldů zadáme čísla, jejich součet bude zobrazen v `JLabelu`. Za použití metody `setText()` inicializujeme výchozí hodnoty v textfieldech na 0.

```
public class Textfieldy extends JFrame {
    private double prvni,druhe,soucet;
    private JLabel l1,l2,l3;
    private JTextField t1, t2;
    private JButton b;
    public Textfieldy(int vyska, int sirka) {
      this.setSize(vyska,sirka);
      this.setTitle("Scitani cisel");
      this.setLayout(new GridLayout(3,2));
      l1=new JLabel("Prvni cislo");
      l2=new JLabel("Druhe cislo");
      t1=new JTextField();
      t2=new JTextField();
      t1.setText("0");
      t2.setText("0");
      b=new JButton("Soucet");
      l3=new JLabel();
      this.getContentPane().add(l1);
      this.getContentPane().add(l2);
      this.getContentPane().add(t1);
      this.getContentPane().add(t2);
      this.getContentPane().add(b);
      this.getContentPane().add(l3);
      this.setVisible(true);
      t1.addActionListener(new ActionListener()
       {
        public void actionPerformed(ActionEvent e)
        {
         prvniZadej(e);
        }
       });
       t2.addActionListener(new ActionListener()
        {
         public void actionPerformed(ActionEvent e)
         {
          druheZadej(e);
         }
        });
```

```
        b.addActionListener(new ActionListener()
         {
          public void actionPerformed(ActionEvent e)
          {
           soucet(e);
          }
         });
    }

public void prvniZadej(ActionEvent e)
{
 this.prvni=Integer.parseInt(t1.getText());
}
public void druheZadej(ActionEvent e)
{
 this.druhe=Integer.parseInt(t2.getText());
}
public void soucet(ActionEvent e)
 {
  soucet=this.prvni+this.druhe;
  l3.setText(String.valueOf(soucet));
 }
}
```

Použití události FocusLost Pohodlnější pro uživatele by bylo, kdyby po zadání nebyl nucen potvrzovat hodnotu stiskem klávesy Enter. Pokud bychom u obou text fieldů neošetřovali událost `ActionEvent` ale `FocusLost`, odpadlo by potvrzování. Využijeme rozhraní FocusListener. Tento přístup je běžně využíván v praxi. Všimněme si, že musíme předefinovat i druhou metodu, `focusGained()`.

```
 t1.addFocusListener(new FocusListener() {
            public void focusLost(FocusEvent e) {
                prvniZadej(e);
            }
            public void focusGained(FocusEvent e) {};
        });
t2.addFocusListener(new FocusListener() {
            public void focusLost(FocusEvent e) {
                druheZadej(e);
            }
            public void focusGained(FocusEvent e){};
        });
```

Výsledek vypadá takto.

Obrázek 10.29: Součet dvojice čísel s použitím JTextFieldů.

Metoda	Popis
`addItem(Object o)`	Přidání položky do seznamu na poslední pozici.
`insertItem(Object o, int position)`	Přidání položky do seznamu na uvedenou pozici.
`getItem()`	Získání položky na požadované pozici.
`getSelectedItem()`	Získání označené položky.
`removeAllItems()`	Odstranění všech položek seznamu.
`removeItemAt(int position)`	Odstranění požadované položky seznamu.
`getItemCount()`	Počet položek seznamu.
`getSelectedIndex()`	Index označené položky.
`setSelectedIndex()`	Označení požadované položky seznamu.
`setEditable(boolean b)`	Nastavení, zda položky combo boxu mohou být editovatelné.

Tabulka 10.9: Přehled metod třídy JButton.

10.1.8.6 JComboBox

Komponenta `JComboBox` představuje rozbalovací seznam. Seznam je tvořen jednotlivými položkami setříděných podle předem známého klíče. Viditelná je pouze jedna řádka, po kliknutí na šipku se seznam rozbalí a umožní zvolit kliknutím některou z položek seznamu. Použití je podobné jako v případě radio buttonů, komponenta je vhodná pro výběr z většího množství variant. V takovém případě při použití radio buttonů vznikají okna příliš velkých rozměrů, na které se pak již nevejdou další komponenty.

Combo boxy mohou mít poměrně odlišný vzhled i způsob práce s nimi. Jejich položky mohou být editovatelné popř. lze položky přidávat i za běhu programu. My se budeme zabývat nejběžnější variantou-needitovatelnými combo boxy. Položky combo boxu jsou číslovány, lze k nimi přistupovat za použití indexu; první položka má index 0. Třída má k dispozici několik konstruktorů

```
JComboBox();
JComboBox(Object o);
JComboBox(BVector v);
```

První konstruktor vytvoří prázdný combo box, druhý naplněný položkami představovanými objekty třídy `Object`, třetí naplněný položkami z dynamické datové struktury vector. Přehled často používaných metod komponenty `JComboBox` nalezneme v následující tabulce.

Při práci s combo boxem je nejčastěji používáno rozhran `ActionListener`. Metoda `actionPerformed()` je vyvolána v okamžiku, kdy uživatel zvolí některou z položek combo boxu nebo po editaci některé položky stiskne klávesu Enter. Upravíme příklad pracující s radio buttony, místo radiobuttonu použijeme combo box. Ukázka zdrojového kódu.

```
public class Comboboxy extends JFrame
 {
   private JLabel l;
   private JComboBox c;

   public Comboboxy(int vyska, int sirka) {
```

```
        this.setSize(vyska, sirka);
        this.setTitle("ComboBox");
        c=new JComboBox();
        c.addItem("Prvni");
        c.addItem("Druhy");
        c.addItem("Treti");
        l=new JLabel();
        this.getContentPane().add(c);
        this.getContentPane().add(l);
        this.setLayout(new FlowLayout());
        c.addActionListener(new ActionListener() {
               public void actionPerformed(ActionEvent e) {
                 cVyber(e);
               }
           });
        this.setVisible(true);
       }

    public void cVyber(ActionEvent e)
       {
        l.setText(c.getSelectedItem().toString());
       }
    }
```

Výsledek vypadá takto.

Obrázek 10.30: Práce s JComboBoxem.

10.1.8.7 JList

Jedná se o víceřádkovou komponentu představující seznam. Na rozdíl od `JComboBoxu` umožňuje trvale zobrazit celý seznam a nejen pouze jednu jeho položku. V případě, že je délka seznamu větší než délka komponenty nebo šířka položky větší než šířka komponenty, pro listování jsou použity scrollbary.

Výběr ze seznamu. Pro výběr ze seznamu je používána instance třídy `ListSelectionModel`. Výběr ze seznamu lze provádět třemi způsoby:

- Výběr jedné položky
- Výběr souvislého intervalu více položek
- Výběr nesouvislého intervalu více položek.

Nejčastěji je používána první varianta. Způsob výběru lze nastavit metodou `setSelectionMode(int const)` za požití jedné ze tří níže uvedených konstant: `SINGLE_SELECTION, SINGLE_INTERVAL_SELECTION, MULTIPLE_INTERVAL_SELECTION`. Ukázky variant výběrů nalezneme na následujícím obrázku.

Obrázek 10.31: Vlevo jednoduchý výběr, uprostřed souvislý výběr, vpravo nesouvislý výběr.

Třída má k dispozici několik konstruktorů, uvedeme:

```
JList();
JList(vector v);
```

První konstruktor vytvoří prázdný seznam, druhý seznam naplněný položkami vectoru.

Přidávání položek do seznamu. Přidávání položek do seznamu je možné provádět dvěma způsoby.

- S použitím modelu.
- Bez použití modelu.

V dalším výkladu naznačíme stručně obě varianty. Třída `JList` obsahuje mnoho metod, přehled nejčastěji používaných nalezneme v následující tabulce.

Metoda	Popis
`addElement(Object o)`	Přidání položky do seznamu (s využitím modelu).
`add(Component c)`	Přidání položky do seznamu (bez využití modelu).
`setVisibleRowCount(int rows)`	Nastavení počtu současně viditelných řádek seznamu.
`getVisibleRowCount()`	Získání počtu současně viditelných řádek seznamu.
`setSelectedMode(ListModel l)`	Nastavení způsobu výběru položek na jednu ze tří výše uvedených.
`setSelectedIndex(int)`	Nastavení vybrané položky seznamu.
`setSelectedIndices(int[])`	Nastavení více vybraných položek seznamu.
`getSelectedIndex()`	Index vybrané položky v seznamu.
`int[] getSelectedIndices()`	Index vybraných položek seznamu uložený v poli.
`getSelectedValues()`	Vrací seznam všech označených hodnot.
`setListData(Object o)`	Do seznamu vloží požadovaná data.
`clearSelection()`	Smazání všech smazaných položek seznamu.
`removeAll()`	Smazání všech položek seznamu.
`isSelectionEmpty()`	Zjistí, zda byla vybrána nějaká položka.

Tabulka 10.10: Přehled metod třídy JList.

Při práci se seznamy jsou používána dvě rozhraní `ActionListener` a `ItemListener`. Událost `ItemEvent` je generována při jednom kliknutí na položku seznamu, událost `ActionEvent` při dvojkliku na některou z položek seznamu.

V následujícím příkladu vytvoříme dva seznamy. V druhém seznamu budou znázorněny všechny položky, které budou v prvním seznamu zvýrazněny. V tomto případě budeme tedy pro `JList` používat rozhraní `ItemListener` (na položky nebudeme klikat dvojklikem, ale pouze je vybírat).

Přidání dat do seznamu s použitím modelu. V tomto případě využíváme rozhraní `ListModel`. Jeho implementací je třída `DefaultListModel`, která disponuje několika metodami pro práci s modelem. V prvním kroku vytvoříme instanci třídy `DefaultListModel`.

```
DefaultListModel lm=new DefaultListModel();
```

dalším kroku použijeme metodu `addElement()` a přidáme do modelu pět prvků.

```
lm.addElement("Prvni polozka");
lm.addElement("Druha polozka");
lm.addElement("Treti polozka");
lm.addElement("Ctvrta polozka");
lm.addElement("Pata polozka");
```

Vytvoříme novou instanci třídy `JList` (v našem případě dvě) a metodou `setSelectionMode` nastavíme způsob výběru položek v komponentě.

```
l1=new JList();
l2=new JList();
l1.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
l2.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
```

Vytvořený model následně "spojíme" se seznamem.

```
l1.setModel(lm);
```

Mohli bychom použít i explicitní konstruktor

```
l1=new JList(lm);
```

Přidání dat do seznamu bez použití modelu Nejprve vytvoříme stejně jako v předchozím případě obě instance třídy `JList`. Přidání položek do seznamu provedeme takto:

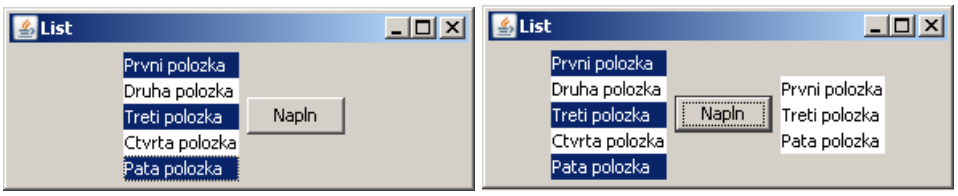
```
lm.add("Prvni");
lm.add("Druhy");
lm.add("Treti");
lm.add("Ctvrty");
lm.add("Paty");
```

Další postup bude v obou případech stejný. V handleru obsluhujícím událost `ActionEvent` zjistíme metodou `getSelectedValues()` všechny zvýrazněné položky a za použití metody `setListData(Object o)` je nastavíme druhému seznamu.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

```
public void cPrevod(ActionEvent e)
 {
   Object [] o=l1.getSelectedValues();
   l2.setListData(o);
 }
```

Výsledek vypadá takto.



Obrázek 10.32: Práce se seznamem.

Ukázka zdrojového kódu:

```
public class Lists extends JFrame
 {
   private JList l1, l2;
   private JButton b;

    public Lists(int vyska, int sirka) {
     this.setSize(vyska, sirka);
     this.setTitle("List");
     l1=new JList();
     l2=new JList();
     JScrollPane sp = new JScrollPane(l1,
     JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
     JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
     DefaultListModel lm=new DefaultListModel();
     lm.addElement("Prvni polozka");
     lm.addElement("Druha polozka");
     lm.addElement("Treti polozka");
     lm.addElement("Ctvrta polozka");
     lm.addElement("Pata polozka");
     b=new JButton("Napln");
     l1.setModel(lm);
     JScrollPane sp2 = new JScrollPane(l2,
     JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
     JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);

     this.getContentPane().add(l1);
     this.getContentPane().add(b);
     this.getContentPane().add(l2);;
     this.setLayout(new FlowLayout());
     b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
              cPrevod(e);
```

```
            }
        });
     this.setVisible(true);
    }
    public void cPrevod(ActionEvent e)
    {
     Object [] o=l1.getSelectedValues();
     l2.setListData(o);
    }
}
```

Scrollování seznamem. Pokud je seznam dlouhý, je vhodný umožnit jeho scrollování, tj. posunování obsahu seznamu prostřednictvím komponenty `JScrollPane`. Scrollování může být jak vodorovné, tak svislé. Komponenta má několik konstruktorů, z nichž nejčastěji je používán tento

```
JScrollPane(Component c, int vertical, int horizontal);
```

Je vytvořen nový scrollovací panel scrollující obsahem zadané komponenty ve směru horizontálním, vertikálním, obou či žádném na základě hodnot proměnných vertical a horizontal. Ty mohou nabývat následujících hodnot.

Položka	Popis
`HORIZONTAL_SCROLLBAR_ALLWAYS`	Vodorovný posuvník vždy.
`HORIZONTAL_SCROLLBAR_AS_NEEDED`	Vodorovný posuvník jen když jepotřeba.
`HORIZONTAL_SCROLLBAR_NEVER`	Vodorovný posuvník nikdy
`VERTICAL_SCROLLBAR_ALLWAYS`	Svislý posuvník vždy.
`VERTICAL_SCROLLBAR_AS_NEEDED`	Svislý posuvník jen když je potřeba.
`VERTICAL_SCROLLBAR_NEVER`	Svislý posuvník nikdy.

Tabulka 10.11: Přehled metod třídy JScrollPane.

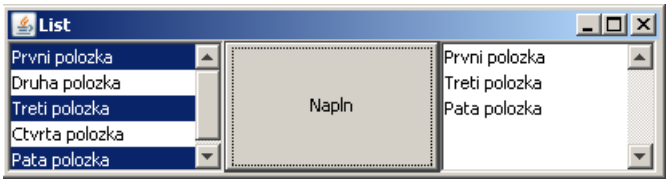
Vytvoříme nový scrollovací panel, který bude pohybovat obsahem seznamu v obou směrech. Výsledek bude vypadat takto.

```
JScrollPane sp1 = new JScrollPane(l1,
     JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
     JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
```

dalším kroku již *nesmíme* na `ContentPane` formuláře přidat seznam, ale jen nově vytvořený scrollovací panel. Stejným způsobem postupujeme i v případě druhého `ListBoxu`.

```
 JScrollPane sp1 = new JScrollPane(l1,
     JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
     JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
     JScrollPane sp2 = new JScrollPane(l2,
     JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
     JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
     this.getContentPane().add(sp1);
     this.getContentPane().add(b);
     this.getContentPane().add(sp2);
```

Komponenty s posuvníky vypadají “estetičtěji”.



Obrázek 10.33: Použití posuvníků při práci se seznamem.

10.1.8.8 JTable

Komponenta představuje tabulku, často se používá pro zobrazování výsledků výpočtů, výpisu seznamů či zadávání vstupních dat. Data v tabulce mohou, ale nemusí být, editovatelná. S tabulkou se zpravidla pracuje za použití dvojice modelů. První model je používán pro sloupce (z časového hlediska se mu nebudeme věnovat), druhý model pro celou tabulku. Model sloupce ovlivňuje způsob zobrazení dat ve sloupci či možnosti jejich editace, model tabulky disponuje řadou metod pro práci s vlastní tabulkou. Možnosti práce s tabulkami jsou v Javě velmi rozsáhlé (některé pasáže jsou poměrně obtížné a komplikované), uvedeme pouze nejdůležitější informace. K dispozici je několik konstruktorů

```
JTable ();
JTable (int rows,int columns);
JTable (TableModel t);
JTable(Object[][] data, Object[] column_names)
JTable(Vector row_data, Vector column_names)
```

První z konstruktorů vytvoří prázdnou tabulku, druhý prázdnou tabulku se zadaným počtem řádků a sloupců, třetí tabulku naplněnou komponentami z instance třídy `TableModelu`. Čtvrtý a pátý konstruktor vytvoří tabulku naplněnou daty, včetně názvů sloupců. Všechny položky tabulky jsou editovatelné, jednotlivé prvky tabulky musí být uloženy v dynamické datové struktuře představované vektorem.

Přidání tabulky na JScrollPane. Při práci s rozsáhlými tabulkami je vhodné provést jejich přidání na `JScrollPane`. Učiníme to podobným způsobem jako v případě seznamů.

```
ScrollPane sp = new JScrollPane(table,
     JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
     JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
```

Implementace modelu. Chceme -li vytvořit vlastní model tabulky, použijeme abstraktní třídu `AbstractTableModel`. Astraktní metody třídy je nutné předefinovat, můžeme tak ovlivnit jejich funkčnost podlem potřeby. V tomto případě je nutné předefinovat následující trojici metod: `getRowCount()`, `getColumnCount()` a `getValueAt()`. Dále je vhodné předefinovat metodu `getColumnName()` vracející jméno sloupce. Chceme -li tabulku následně editovat, musíme přetížit metody `setValueAt(int row,int column)` a `isCellEditable(bool status)`. Při změně dat je nutné tuto informaci poskytnout všem zaregistrovaným posluchačům.

Pro jednodušší případy, kdy vystačíme s výchozí implementací, je vhodnější použít třídu `DefaultTableModel`. Jednotlivé řádky jsou ukládány do dynamické struktury `Vector`. Model umožňuje automatické generování událostí při změně dat uživatelem, buňky tabulky lze tedy editovat.

Vytvoření tabulky. V příkladu věnovaném práci s tabulkou vytvoříme třídu, která je potomkem abstraktní třídy `AbstractTableModel`. Tabulka bude tvořena 5 sloupci, které budou zobrazovat mocniny čísel prvních pět členů aritmetických posloupností čísel 1-5. Třídu nazveme `Tabulka`. Obsah tabulky tvoří položky 2D pole typu `Object`, sloupce položky pole typu `String`. Deklarujeme odkazy na tyto proměnné jako datové položky třídy.

```
public class Tabulka extends AbstractTableModel{
    private Object [][] data;
    String [] sloupce;
....
}
```

Jejich inicializaci provedeme v konstruktoru. Konstruktor bude explicitní, abychom mu mohli předat počet řádek a sloupců takto vytvořené tabulky.

```
 public Tabulka(int rows, int columns) {
   data=new Object[rows][columns];
   sloupce=new String[columns];
...
}
```

Generování posloupnosti provedeme ve speciální metodě nazvané `getPosloupnost()`, popis sloupců v metodě `Sloupce()`. Obě metody budou vykonány v konstruktoru.

```
public Tabulka(int rows, int columns) {
   data=new Object[rows][columns];
   sloupce=new String[columns];
   this.Sloupce();
   this.getPosloupnost();
}
```

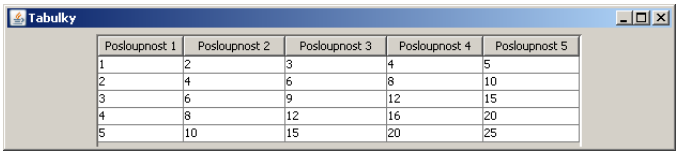
Zdrojový kód metod vypadá takto.

```
 public void getPosloupnost()
    {
     for (int i=0;i<data.length;i++)
         {
          for (int j=0;j<data[0].length;j++)
           {
            data[i][j]=(i+1)*(j+1);
           }
         }
    }
public void Sloupce()
    {
     sloupce[0]="Posloupnost 1";
     sloupce[1]="Posloupnost 2";
     sloupce[2]="Posloupnost 3";
     sloupce[3]="Posloupnost 4";
     sloupce[4]="Posloupnost 5";
    }
```

Nyní provedeme předefinování všech výše uvedených abstraktních metod třídy `AbstractTableModel` tak, aby poskytovaly použitelné výsledky.

```
public int getRowCount() {return data.length;}
public int getColumnCount() {return data[0].length;}
public Object getValueAt(int rowIndex, int columnIndex) {
return data[rowIndex][columnIndex];
}
```

Výsledek vypadá takto.



Obrázek 10.34: Tabulka se záhlavím sloupců naplněná hodnotami.

Nastavení šířky sloupců. U tabulky často potřebujeme nastavit různé šířky sloupců v závislosti na typu zobrazovaných dat. K tomuto účelu slouží metoda `setPreferredWidth(int sirka)`. Šířka je zadávána v pixelech. Metoda je volána pro konkrétní sloupec získaný metodou `getColumn(int index)`.

```
t.getColumnModel().getColumn(0).setPreferredWidth(50);
```

Měnit šířku sloupců je též možno provádět prostřednictvím myši. Povolit či zakázat tuto možnost lze provést prostřednictvím metody `setAutoResizeMode(int mode)`.

Mód	Popis
`AUTO_RESIZE_SUBSEQUENT_COLUMNS`	Změna šířky sloupce tabulky, nemá vliv na sousední sloupce.
`AUTO_RESIZE_NEXT_COLUMN`	Změna šířky sloupce na úkor šířky následujícího sloupce.
`AUTO_RESIZE_ALL_COLUMNS`	Změna šířky sloupce ovlivní šířku všech ostatních sloupců.
`AUTO_RESIZE_LAST_COLUMN`	Změna šířky sloupce na úkor šířky předchozího sloupce.
`AUTO_RESIZE_OFF`	Šířku sloupce není možné měnit.

Tabulka 10.12: Módy změny šířky sloupců u tabulky.

Výběr řádků tabulky. V tabulce je možno označovat (tj. vybírat) řádky, se kterými můžeme v dalších krocích provádět různé manipulace. Existují tři možnosti výběru (shodné s komponentou `JList`), lze je nastavit metodou `setSelectionMode()`. Detekci řádky/řádek, které byly vybrány, lze provést s využitím instance třídy `ListSelectionModel`. Vytvoříme instanci třídy `ListSelectionModel`, kterou inicializujeme prostřednictvím metody `getSelectionModel()`.

```
ListSelectionModel vyber=t.getSelectionModel();
```

Při výběru řádky dochází k události `ListSelection`, pro práci s ní využijeme rozhraní `ListSelectionListener`. Zaregistrujeme posluchače

```
vyber.addListSelectionListener(new ListSelectionListener());
```

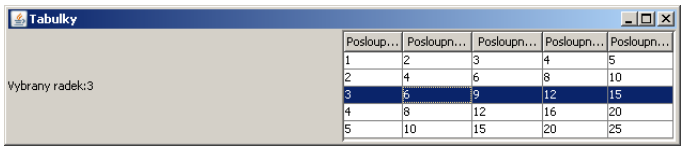
a následně implementujeme abstraktní metody rozhraní. V metodě vytvoříme inicializujeme odkaz třídy `ListSelectionModel` aktuálním modelem za použití metody `getSource()`. Na něj následně aplikujeme metodu `isSelectionEmpty()` sdělující, zda je výběr prázdný. Pokud není, získáme vybranou řádku metodou `getMinSelectionIndex()`. Postup je, jak vidíme, poměrně komplikovaný.

```
int radka;
vyber.addListSelectionListener(new ListSelectionListener() {
 public void valueChanged(ListSelectionEvent e) {
    int radka;
    ListSelectionModel ls=(ListSelectionModel) e.getSource();
    if (!l.isSelectionEmpty()) radka=ls.getMinSelectionIndex()
   }
});
```

Číslo vybrané řádky sdělíme v `JLabelu`, který na formulář přidáme. Kód upravíme, výsledek vypadá takto:

```
public class Tabulky3 extends JFrame{
    private int radka;
    private JLabel l;
    public Tabulky3(int width, int height)
    {
     this.setSize(width, height);
     this.setTitle("Tabulky");
     l=new JLabel();
     TableModel tm=new TabulkyDataAbstract(5,5);
     JTable t=new JTable(tm);
     this.setLayout(new GridLayout(1,2));
     JScrollPane sp = new JScrollPane(t,JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
     JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
     this.getContentPane().add(l);

     t.getColumnModel().getColumn(0).setPreferredWidth(50);
     ListSelectionModel vyber = t.getSelectionModel();
     t.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
     vyber.addListSelectionListener(new ListSelectionListener() {
       public void valueChanged(ListSelectionEvent e) {
        ListSelectionModel ls=(ListSelectionModel) e.getSource();
        if (!ls.isSelectionEmpty())
          {
            radka=ls.getMinSelectionIndex();
            l.setText("Vybrany radek:"+String.valueOf(radka+1));
          }
        }
    });
    t.setAutoResizeMode(t.AUTO_RESIZE_SUBSEQUENT_COLUMNS);
    this.getContentPane().add(sp);
    this.setVisible(true);
    }
}
```

Výsledek vypadá takto.



Obrázek 10.35: Výběr řádku tabulky.

Detekce změn v tabulce. Chceme -li detekovat změny v tabulce, použijeme rozhraní `TableModelListener`. Rozhraní implementuje jednu metodu a to `tableChanged()`. Tuto metodu musíme předefinovat.

```
t.getModel().addTableModelListener(new TableModelListener()
   {
           public void tableChanged(TableModelEvent e) {
           }

   });
```

Sloupec, ve kterém se upravovaná buňka nachází, můžeme detekovat prostřednictvím metody `getColumn()`, řádek, ve kterém se upravovaná buňka nachází, prosřednictvím metody `getRow()`. Upravenou hodnotu lze získat prostřednictvím metody `getValueAt(int row, int column)`.

```
public void tableChanged(TableModelEvent e) {
 int radek = e.getRow();
 int sloupec = e.getColumn();
 TableModel m = (TableModel)e.getSource();
 Object data = model.getValueAt(row, column)
}
```

Další operace s `JTable` již nebudeme uvádět, lze je nalézt ve specializované literatuře. Přehled nejpoužívanějším metod u komponenty `JTable`.

10.1.8.9 Dialogová okna

Pokud potřebujeme v aplikaci vytvořit nějaké další okno, zpravidla k tomu nepoužíváme třídu `JFrame`. Jak jsme uváděli výše, aplikace by měla disponovat jedním framem. Ostatní okna jsou vytvářena formou dialogu. Práce s dialogy je podobná jako s framy. Dialogová okna dělíme do dvou základních skupin:

- Modální okna.
- Nemodální okna.

Modální okno je takové okno, které při otevření zůstává neustále navrchu, překryje všechna ostatní okna aplikace. Dokud není uzavřeno, znemožňuje práci s jinými okny aplikace. Modální okna se často používají pro dialogy. *Nemodální* okna tuto vlastnost nemají, lze s nimi pracovat bez omezení. Práce s dialogy je podobná práci s formuláři.

Dialogy jsou vhodné pro nastavování parametrů programu, umožňují uživateli zareagovat na činnost programu, informují ho o aktuálním stavu programu. Dialogy lze navrhovat dle vlastní potřeby, můžeme však využívat některé standardizované dialogy.

Metoda	Popis
`getRowCount()`	Získání počtu řádků tabulky.
`getColumnCount()`	Získání počtu sloupců tabulky.
`getValueAt(int row, int column)`	Získání hodnoty v zadaném řádku a sloupci.
`getColumnName(int column)`	Získání názvu sloupce.
`isCellEditable(int row, int column)`	Je hodnota v zadaném řádku a sloupci editovatelná.
`setPreferredWith(int width)`	Nastavení požadované šířky sloupce v pixelech.
`getColumn(Object o)`	Vrací instanci typu TableColumn.
`setAutoResizeMode(int mode)`	Nastavení módu změny šířky sloupce myší.
`setSelectionMode(int mode)`	Nastavení možnosti výběru řádek u tabulky (jedna, více,..)
`getEditingRow()`	Vrací číslo řádky, která je editována.
`getEditingColumn()`	Vrací číslo sloupce, který je editován.
`setEditingRow()`	Nastaví číslo řádky, která bude editována.
`setEditingColumn()`	Nastaví číslo sloupce, který bude editován.
`setModel()`	Nastavení modelu pro tabulku.

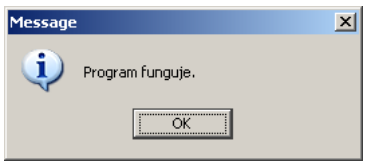
Tabulka 10.13: Přehled metod třídy JTable.

Předdefinované dialogy. V knihovně Swing lze nalézt řadu předdefinovaných dialogů. Třída `JOptionPane` obsahuje statické metody, prostřednictvím kterých lze dialogy vyvolávat. Podíváme se na nejčastěji používané dialogy.

ShowMessageDialog. Tento modální dialog slouží ke sdělování zpráv a důležitých informací ve směru program->uživatel. Je nejjednodušším dialogem, zobrazuje jediné tlačítko s popisem OK. Vzhled dialogu může být různě modifikován, metoda `JOptionPane.showMessageDialog()` je přetížena. Použijeme -li

```
JOptionPane.showMessageDialog(JComponent c, String text);
```

je vytvořen jednoduchý dialog zobrazující zadaný text. V titulku je zobrazen text "Message", okno nedisponuje žádnou ikonou.



Obrázek 10.36: ShowMessage dialog.

Okno může mít vlastní titulek, v závislosti na informaci, kterou sděluje; lze v něm zobrazovat několik typů ikon které naznačují charakter okna (informační, výstražné, chybové...). Syntaxe vypadá takto

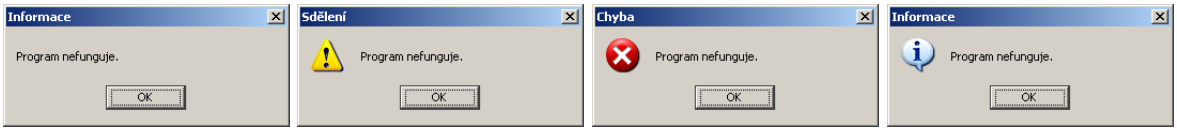
```
JOptionPane.showMessageDialog(JComponent c, String text, String nadpis,
int JOptionPane);
```

Hodnoty konstant jsou uvedeny v následující tabulce.

Konstanta	Popis
`PLAIN_MESSAGE`	Okno neobsahuje žádnou ikonu, pouze prostý text.
`WARNING_MESSAGE`	Výstražné okno
`ERROR_MESSAGE`	Chybové okno.
`INFORMATION_MESSAGE`	Informační okno.

Tabulka 10.14: Přehled konstant třídy JOptionPane.

Výsledky vypadají takto.



Obrázek 10.37: Okno s prostým textem, výstražné okno, chybové okno, informační okno.

showOptionDialog. Tento modální dialog má širší možnosti než pouhé zobrazení zprávy. Jeho vzhled i funkčnost mohou být výrazněji modifikovány. Zobrazíme ho metodou `JOptionPane.showMessageDialog()`, která je opět přetížena. Popíšeme ho pouze stručně, řadu dalších informací lze nalézt v dokumentaci.

```
int s showOptionDialog(Component parent, Object text, String title,
int optionType, int messageType,Icon icon, Object[] options,
Object initialValue);
```

Metoda obsahuje řadu parametrů, jejich stručný popis je uveden v následující tabulce.

Parametr	Popis
`Component parent`	Jméno nadřazené komponenty.
`Object text`	Text zobrazený v dialogovém okně.
`String title`	Nadpis dialogového okna.
`int optionType`	Typ tlačítek, které budou v dialogu zobrazeny.
`int messageType`	Typ ikony, která je v dialogu zobrazena.
`Icon icon`	Ikona zobrazená ve stavovém řádku okna.
`Object [] options`	Popisy jednotlivých tlačítek dialogu.
`Object initialValue`	Výchozí hodnota, která může být nastavena (součástí okna může být i combo box)

Tabulka 10.15: Přehled parametrů ShowOption dialogu.

Práce s ikonami je stejná jako v případě message dialogu, nebudeme ji proto uvádět. Typy tlačítek zobrazených v dialogu jsou dány konstantou, která může nabývat následujících hodnot.

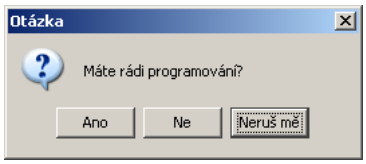
Konstanta	Popis
NO_OPTION	Tlačítko No
OK_CANCEL_OPTION	Tlačítka OK a Cancel
OK_OPTION	Tlačítko OK
YES_NO_CANCEL_OPTION	Tlačítka YES, NO, CANCEL
YES_NO_OPTION	Tlačítka YES, NO
CANCEL_OPTION	Tlačítko CANCEL
YES_OPTION	Tlačítko YES

Tabulka 10.16: Typy tlačítek zobrazovaných v ShowOption dialogu.

Ukázka zdrojového kódu:

```
Object[] options = {"Ano","Ne","Neruš mě"};
int s=JOptionPane.showOptionDialog(this,Máte rádi programování?",
Otázka",JOptionPane.YES_OPTION,JOptionPane.QUESTION_MESSAGE, null,
options, options[2]);
```

Výsledek vypadá takto.

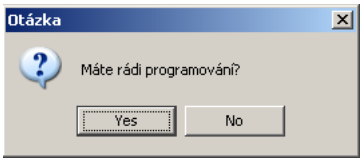


Obrázek 10.38: Třítlačítková varianta.

Chceme -li vzhled okna upravit, použijeme jinou variantu přetížené metody `showOptionDialog()`. Tlačítka nemusí mít vlastní popis, lze použít defaultní hodnoty.

```
int s=JOptionPane.showConfirmDialog(this, "Máte rádi programování?",
"Otázka",JOptionPane.YES_NO_OPTION);
```

Kombinace českého a anglického jazyka nepůsobí dobře, měli bychom se jí vyhýbat.



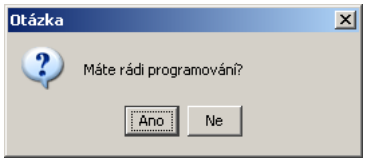
Obrázek 10.39: Dvoutlačítková česko-anglická varianta.

Ukázka třetí varianty přetížené metody se dvěma tlačítky:

```
Object[] opt = {"Ano","Ne"};
int s=JOptionPane.showOptionDialog(this,"Máte rádi programování?",
"Otázka",JOptionPane.YES_NO_OPTION,JOptionPane.QUESTION_MESSAGE, null,
opt, opt[0]);
```

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"

Výsledek vypadá takto.



Obrázek 10.40: Upravená dvoutlačítková varianta v českém jazyce.

Detekce stisknutého tlačítka. Metoda `showOptionDialog()` vrací kód stisknutého tlačítka. Můžeme tak detekovat, kterým tlačítkem uživatel dialog zavřel.

```
if (s==JOptionPane.YES_OPTION)
 {
  //stiskl uzivatel tlacitko YES?
 }
```

Vytvoření vlastního dialogu. Vlastní dialog lze vytvořit jako instanci třídy `JDialog`. Práce s ním je podobná, jako v případě formulářů, nebudeme ji tedy uvádět. Na dialog můžeme přidávat další komponenty, používat layout managery, atd... Dialogu je možno nastavit modalitu/nemodalitu a to za použití metody `setModal(boolean b)`. Zobrazení dialogu je možné provést metodou `setVisible(boolean b)`. Následující kód vytvoří modální dialog.

```
JDialog jd=new JDialog();
jd.setTitle("Vlastní dialog");
jd.setModal(true);
jd.setVisible(true)
```

10.1.8.10 JFileChooser

Tento modální dialog slouží pro výběr souboru, je jedním z nejčastěji používaných dialogů. K dipozici je několik konstruktorů

```
JFileChooser();
JFileChooser(String directory);
```

První konstruktor vytvoří dialog zobrazující při otevření běžný adresář (current directory), druhý konstruktor dialog vytvářejí dialog zobrazující zadaný adresář.

Režimy dialogu `JFileChooser` může pracovat ve dvou režimech. Prvním je režim, ve kterém soubory můžeme otevírat, druhým je režim, ve kterém soubory můžeme ukládat. Fyzické otevření ani uložení souboru není provedeno, pouze je vrácen odkaz na vybraný soubor. Režimy jsou dány konstantami `OPEN_DIALOG` nebo `SAVE_DIALOG`, je možno nastavit metodou `setDialogType(DialogType d)`.

```
fc.setDialogType(JFileCooser.OPEN_DIALOG)
fc.setDialogType(JFileCooser.SAVE_DIALOG)
```

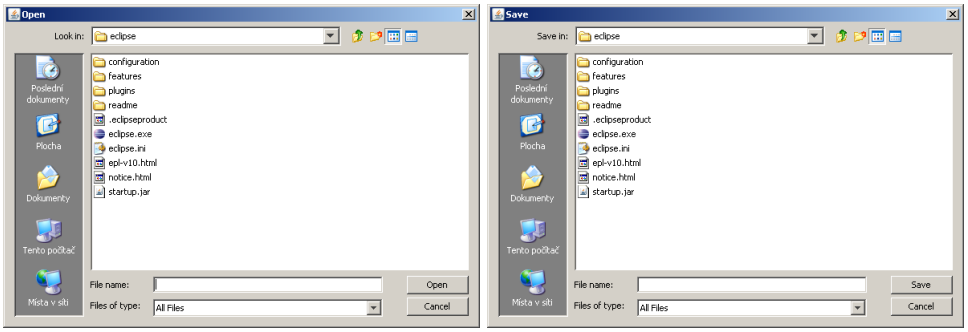
Oba dialogy se liší názvem titulku a popisem tlačítka.

Vytvoření dialogu. Pro zobrazení open dialogu je používána metoda `int showOpenDialog(JComponent c)`, pro zobrazení save dialogu je používána metoda `int showSaveDialog(JComponent c)`. Metody vracejí kód stisknutého tlačítka. Můžeme tak detekovat, zda uživatel provedl volbu nějakého souboru.

Vytvořme open dialog zobrazující při startu obsah adresáře C:\\Eclipse.

```
JFileChooser fc=new JFileChooser("C:\\eclipse");
int fc.showOpenDialog(this);
```

Výsledek vypadá takto.



Obrázek 10.41: Ukázka dialogů v režimech open a save.

Chceme -li dialogu nastavit titulek, použijeme metodu `setDialogTitle(String text)`.

```
fc.setDialogTitle("Otevření souboru");
```

Výběr souboru. Detekce vybraného souboru je možné provést po stisknutí tlačítka Open/Save za použití testu návratové hodnoty metod `showOpenDialog()` nebo `showSaveDialog()`. Pro získání jména vybraného souboru použijeme metodu `getSelectedFile(File f)`. Detekci, zda bylo okno zavřeno stiskem tlačítka Open/Save, provedeme za použití konstant `APPROVE_OPTION` a `CANCEL_OPTION`.

Konstanta	Popis
`APPROVE_OPTION`	Detekce, zda dialog uzavřen Open/Save
`CANCEL_OPTION`	Detekce, zda dialog uzavřen Cancel.

Tabulka 10.17: Detekce uzavření dialogu.

Výsledný kód může vypadat např. takto

```
if (fc.showOpenDialog(this)==fc.APPROVE_OPTION)
 {
  //nejaky kod
  System.out.println(fc.getSelectedFile().toString());
 }
```

Nastavení masky. V oknech file chooseru nemusí být nutně zobrazeny všechny typy souborů. Typy zobrazených souborů mohou být omezeny za použití příkazové masky. Můžeme tak zobrazovat pouze soubory stejného typu, s jakými pracuje naše aplikace. Pro definici filtrů slouží třída `FileFilter`.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

10.1.8.11 JPanel

Panel představuje nejjednodušší typ kontejnerové komponenty. Jeho jediným úkolem je organizovat komponenty, které jsou v něm obsaženy. Panely se používají při návrhu vzhledu okna. Za použití vhodného layout manageru umístíme na formulář panely, na tyto panely následně další komponenty. Pro `JPanel` lze použít následující konstruktory:

```
JPanel();
JPanel(LayoutManager);
```

První konstruktor vytvoří panel s výchozím layout managerem (`FlowLayout`), druhý konstruktor vytvoří panel s rozložením komponent dle zadaného layout manageru. Vzhled panelu a jeho okraje (borders) je možno nastavit prostřednictvím metody `setBorder (Border b)`. V Javě existuje několik typů borderů, jejich přehled je uveden v následující tabulce.

Border	Varianty
`BevelBorder`	`BevelBorder.LOWERED` `BevelBorder.RAISED`
`CompoundBorder`	-
`EmptyBorder`	-
`EtchedBorder`	`EtchedBorder.LOWERED` `EtchedBorder.RASISED`
`LineBorder`	`LineBorder.createBlackLineBorder()` `LineBorder.createGrayLineBorder()`
`SoftBevelBorder`	`SoftBevelBorder.LOWERED` `SoftBevelBorder.RAISED`

`TitledBorder`	`TitledBorder.DEFAULT_POSITION`
	`TitledBorder.LEFT`
	`TitledBorder.RIGHT`
	`TitledBorder.TOP`
	`TitledBorder.BOTTOM`

Tabulka 10.18: JPanel a typy borderů.

Popis jednotlivých borderů nebudeme z důvodu rozsáhlosti provádět, uvedeme pouze dvě doporučení. Pokud budeme chtít použít border bez ohraničení pouze jako podkladovou komponentu pro jiné komponenty, použijeme `EmptyBorder`. Chceme -li použít border ohraničující komponenty na `JPanelu`, použijeme `TitledBorder`. Nastavení borderu pro panel provádíme metodou `setBorder(Border b)`.

V následujícím příkladu vytvoříme dva panely, oběma nastavíme `TitledBorder` a každému z nich titulek. Komponenty na formu budou využívat GridLayout. Pro vytvoření panelů použijeme implicitní konstruktory.

```
JPanel p1=new JPanel();
JPanel p2=new JPanel();
```

Vytvoříme dvě instance třídy TitledBorder, v konstruktoru uvedeme titulky.

```
TitledBorder b1=new TitledBorder("První");
TitledBorder b2=new TitledBorder("Druhý");
```

Nastavíme polohu titulku na výchozí pozici, tj. vlevo nahoru.

```
b1.setTitlePosition(TitledBorder.DEFAULT_POSITION);
b2.setTitlePosition(TitledBorder.DEFAULT_POSITION);
```

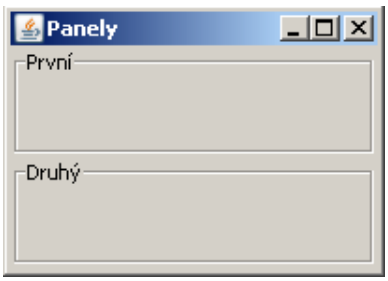
Oběma panelům nastavíme vytvořené bordery.

```
p1.setBorder(b1);
p2.setBorder(b2);
```

Výsledný kód vypadá takto.

```
public Panely(int width, int height) {
        this.setSize(width, height);
        this.setTitle("Panely");
        this.setLayout(new GridLayout(2,1));
        JPanel p1=new JPanel();
        JPanel p2=new JPanel();
        TitledBorder b1=new TitledBorder("První");
        TitledBorder b2=new TitledBorder("Druhý");
        b1.setTitlePosition(TitledBorder.DEFAULT_POSITION);
        b2.setTitlePosition(TitledBorder.DEFAULT_POSITION);
        p1.setBorder(b1);
        p2.setBorder(b2);
        this.getContentPane().add(p1);
        this.getContentPane().add(p2);
}
```

Výsledek vypadá takto.

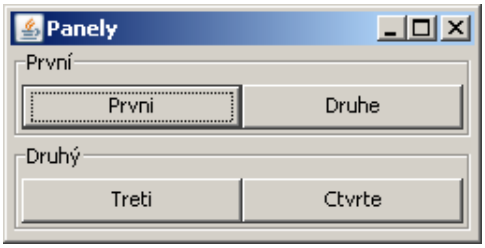


Obrázek 10.42: Vzhled TitleBorder.

Do každého z panelů můžeme následně přidat dvě tlačítka. Oběma panelům nastavíme opět `BorderLayout`.

```
p1.setLayout(new GridLayout(1,2));
p2.setLayout(new GridLayout(1,2));
```

Vytvoříme dvě tlačítka a metodou `add()` je přidáme na panely. Při práci s panely již není potřeba pracovat s grafickým kontextem komponenty prostřednictvím metody `getContentPane()`.



Obrázek 10.43: Vzhled TitleBorder se čtveřicí tlačítek.

Výsledný kód vypadá takto.

```
JButton but1=new JButton("Prvni");
JButton but2=new JButton("Druhe");
JButton but3=new JButton("Treti");
JButton but4=new JButton("Ctvrte");
p1.add(but1);
p1.add(but2);
p2.add(but3);
p2.add(but4)
```

10.1.9 Grafika

V této kapitole se seznámíme se základy grafiky v Javě nad rozhraním Swing. Naučíme se kreslit na plochu komponent, používat barvy či načíst rastrová data. Budeme pracovat s třídou Graphics, která umožňuje základní grafické operace. Pro pokročilejší kresbu lze použít třídu Graphics2D, která je také součástí Javy, její popis zde již neuvádíme.

V Javě můžeme kreslit jak 2D, tak i 3D objekty. Existuje velké množství externích tříd obsahujících nástroje provádějící řadu pokročilých grafických operací jako rendering, animaci, práci s multimédii. Java dokáže využívat knihovny OpenGL.

Rychlost grafiky v Javě. V praxi se často setkáváme s argumentem, že grafika v Javě je pomalá. Uveďme toto tvrzení na správnou míru. Tento nedostatek se projevoval zejména u prvních verzí jazyka Java, rychlost interpretovaného kódu byla až 30 nižší než u kompilovaného C++. V takovém případě bylo možno označit grafiku v Javě za poměrně pomalou. V současné době je Java asi 2x pomalejší než C++ (přibližný údaj), grafiku v Javě můžeme považovat za poměrně rychlou.

Kdy dojde k vykreslení obrazu? Při kreslení v Javě za použití třídy Graphics nemůžeme přesně určit okamžik, kdy dojde k vykreslení obrazu. Kresba je provedena v okamžiku, kdy to grafické rozhraní považovat za nezbytné. Překreslení obrazu však můžeme vyvolat "uměle" prostřednictvím metody `repaint()`.

Tento způsob je poněkud nezvyklý, vede však uživatele k tomu, že není možné na plochu okna kreslit bez systému v okamžiku, kdy se mu zrovna zachce. Je způsoben tím, že vykreslování nesmí být přerušeno žádnou jinou činností, proto JVM vyčkává na vhodný okamžik. Vykreslování probíhá hierarchicky od komponent s vyšší hierarchií sestupně ke komponentám s nižší hierarchií.

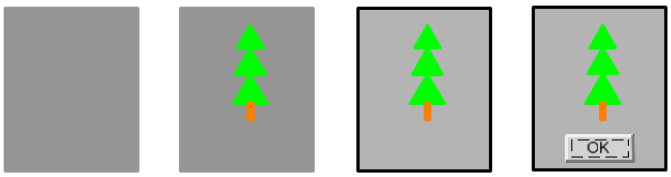
TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

Double buffering. Java pro zvýšení kvality podporuje vykreslování techniku double bufferingu. Vykreslování v takovém případě neprobíhá po přímo na komponentě, ale ve specializovaném bufferu v paměti. Po vykreslení do bufferu je obrázek "přenesen" na komponentu jako celek. Výsledkem je kvalitnější obraz než při klasickém přístupu.

Kreslící plocha má v Javě tvar obdélníka. Pro zvýšení rychlosti vykreslování se v Javě často používá trik, kdy komponenty označíme za průhledné. Grafické rozhraní pak nemusí ztrácet čas s jejich vykreslováním. Vykreslování probíhá na plátno komponenty, které se nazývá Canvas.

Metody pro kreslení ve Swingu. Jak jsme uvedli v úvodu, při překreslování je používán hierarchický model. Ten udává pořadí překreslování jednotlivých grafických kompoment na formuláři. Nejprve je vykresleno pozadí formuláře, následně vykreslení grafiky, poté okraj komponenty a jako poslední komponenty s nižší hierarchií umístěné na této komponentě.

Postup je realizován trojicí metod.



Obrázek 10.44: Postup překreslení komponenty.

1. Metoda `paintComponent()` je nejdůležitější metodou, která se používá při prekreslování komponent. Nejprve provádí překreslení pozadí, poté provádí vykreslení grafiky. Tato metoda může, a často bývá, uživatelem předefinována, umožňuje mu ovlivnit způsob vykreslování vlastní grafiky.
2. Následně metoda `paintBorder()` provede vykreslení okraje komponenty. Tato metoda by neměla být uživatelem předefinovávána.
3. Metoda `paintChildren()` provede vykreslení komponent na hierarchicky nižší úrovni. Ani tato metoda by neměla být uživatelem předefinována.

Princip překreslení plochy. Překreslení plochy komponenty je provedeno po vyvolání metody `repaint()`. Tato metoda zavolá nejprve metodu `update()`, která přes komponentu nakreslí vyplněný obdélník v barvě pozadí a tak vlastně kresbu vymaže. Následně je zavolána metoda `paintComponent()`, která provede vykreslení. Vyvoláním metody `repaint()` dojde vždy k smazání původní kresby. Metoda repaint je přetížená, uveďme některé prototypy:

```
repaint();
repaint(int time);
repaint(int time, int x, int y, int width, int heigh);
```

První metoda provede překreslení v okamžiku, kdy to systém uzná za vhodné, druhá v zadaném čase v milisekundách, třetí překreslí pouze oblast zadanou výřezem v zadaném čase v milisekundách.

10.1.9.1 Základní entity a jejich kresba

V této kapitole se seznámíme se základními grafickými entitami a metodami jejich vykreslování v GUI Javy. Každý složitější útvar lze z těchto entit složit, představují tedy základní stavební jednotku výkresu. Upozorněme, že ve třídě Graphics nemůžeme nastavit linii šířku a typ, z grafických atributů jí můžeme nastavit pouze barvu. Stručně popišme některé grafické entity:

Linie. Pro kresbu linie je používána metoda `drawLine(int x1, int y1, int x2, int y2)`. Vykreslí linii jdou z počátečního bodu $P_1 = [x_1, y_1]$ do koncového bodu $P_2 = [x_2, y_2]$.

Eliptická výseč. Kresbu eliptické výseče lze provést metodou `drawArc(int x, int y, int sirka, int vyska, int start_uhel, int konc_uhel)`. Souřadnice x, y představují levý horní roh obdélníka zadané šířky a délky, do kterého je výseč vepsána, jsou dány počáteční a koncový úhel ovlivňující "délku" oblouku.

Elipsa. Metoda `drawOval(int x, int y, int sirka, int vyska)` slouží k vykreslení elipsy vepsané do obdélníku zadané šířky a délky, jsou dány souřadnice levého horního rohu obdélníka.

Polygon. Polygon představuje uzavřenou oblast definovanou lomovými body spojenými úsečkami. Lze ho nakreslit metodou `drawPolygon()`, která je přetížená. Existují dvě varianty: `drawPolygon(int [x], int [y], int pocet_vrcholu)` a `drawPolygon(Polygon p)`. V první je polygon definován polem x-ových a y-ových souřadnic, druhý objektem typu polygon.

Polyline. Polyline představuje lomenou čáru, lze ho nakreslit metodou `drawPolyline( int [x], int [y], int pocet_vrcholu)`. Význam parametrů je stejný jako u polygonu.

Obdélník. Pro kresbu obdélníku slouží metoda `drawRect(int x, int y, int sirka, int vyska)`. Význam parametrů je stejný jako u elipsy.

3D obdélník. Metoda `draw3DRect(int x, int y, int sirka, int vyska, boolean stav)` vytvoří plastický obdélník připomínající tlačítko. Tlačítko může být zamáčklé nebo nikoliv, rozhoduje o tom booleovská proměnná `stav`.

Text. Pro umístění textu na specifické souřadnice je používána metoda `drawString( String text, int x, int y)`.

Vyplňování entit barvou. Další skupinu nástrojů tvoří metody pro vyplňování některých výše uvedených entit barvou. Útvary jsou standardně vyplňovány stejnou barvou, jakou byly nakresleny. Připomeňme metody `setColor( Color c)` a `getColor()` pro nastavení a získání barvy. Pokud chceme vyplnit útvar jinou barvou, než je nakreslen jeho obrys, postupujeme následovně. nastavíme barvu pro výplň a vyplníme útvar, změníme barvu vykreslíme obrys útvaru. Parametry metod jsou stejné jako v případě metod pro kresbu nevyplněných entit.

Vyplněný 3D obdélník. Metoda `fill3DRect(int x, int y, int sirka, int vyska, boolean stav)` nakreslí vyplněný plastický obdélník.

Vyplněná eliptická výseč. Metoda `fillArc(int x, int y, int sirka, int vyska, int start_uhel, int konc_uhel)` slouží k nakreslení vyplněné eliptické oblasti.

Vyplněná elipsa. Metoda `fillOval(int x, int y, int sirka, int vyska)` umožňuje nakreslení vyplněné elipsy.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

Vyplněný polygon. Pro kresbu vyplněného polygonu jsou používány metody `fillPolygon(int [x], int [y], int pocet_vrcholu)` nebo `fillPolygon(Polygon p)`.

Vyplněný obdélník. Metoda `drawRect(int x, int y, int sirka, int vyska)`, slouží pro kresbu vyplněného obdélníku.

10.1.9.2 Příklady

Informace, které jsme získali se nyní pokusíme aplikovat ve třech příkladech.

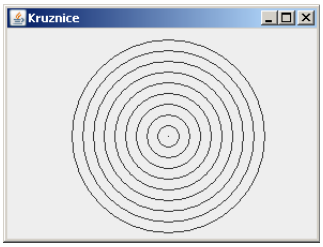
Příklad 1. První z příkladů bude generovat soustředné kružnice se středem ve středu okna s krokem 10 pixelů. Vykreslování bude provedeno v třídě, která je potomkem třídy `JComponent`. Formulář bude vytvořen ve třídě, která bude potomkem třídy `JFrame`. Vykreslování nebudeme provádět přímo na plátno formuláře, ale na komponentu `JPanel` přidanou na formulář.

```
public class Main3 extends JFrame{
    public Main3(int width, int height) {
     this.setSize(width,height);
     this.setTitle("Kruznice");
     JPanel p=new JPanel();
     p.add(new Draw());
     this.getContentPane().add(p);
     p.setLayout(new BorderLayout());
     p.add(new Kruznice(width,height));
     this.setVisible(true);

    }
   public static void main (String [] args)
   {
     new Main3(300,200);
   }
```

Všimnemě si, že na panel přidáme metodou `add()` novou instanci třídy `Kruznice`, která bude zajišťovat vykreslení. Tento postup lze *zobecnit*, instanci třídy provádějící kresbu tedy přidáváme na komponentu, na kterou se bude kreslení provádět. Implementace třídy `Kruznice` vypadá takto.

```
public class Kruznice extends JComponent{
  private int sirka, vyska;
    public Kruznice(int sirka, int vyska) {
    this.sirka=sirka;
    this.vyska=vyska;
    }
    public void paintComponent(Graphics g)
    {
     for (int i=0;(i<this.sirka/2)&&(i<this.vyska/2);i=i+10)
     {
      g.drawOval(sirka/2-i, vyska/2-i, 2*i, 2*i);
     }
    }
}
```

Výsledek vypadá takto.



Obrázek 10.45: Vykreslení soustředných kružnic.

Příklad 2. Příklad ilustruje práci s rozhraními `MouseListener`, a `MouseMotionListener`. Budeme vytvářet výběrovou množinu ve tvaru obdélníku (tj. ohradu), jejíž rozměry se budou měnit na základě polohy myši. Zamysleme se, při kterých událostech myši budeme provádět vykreslování ohrady.

Po stisknutí tlačítka na myši zjistíme aktuální pozici kurzoru, ošetřujeme handler `mousePressed()`. Tlačítko myši držíme stále stisknuté, odečítáme polohu aktuální pozice kurzoru, tj. ošetřujeme metodu `mouseDragged()`, při které provádíme vykreslování obdélníka.

Po uvolnění tlačítka na myši (metoda `mouseReleased()`) nebudeme provádět žádnou akci. Ohrada na ploše zůstane až do dalšího zadávání. Souřadnice levého horního a pravého dolního rohu ohrady budeme ukládat do proměnných typu int.

Ve třídě main deklarujeme potřebné proměnné, vytvoříme `JPanel`, který umístíme na formulář. Kreslení bude probíhat ve třídě `Draw`, jejíž instanci také vytvoříme.

```
public class Main extends JFrame{
    private int x1, y1, x2, y2;
    private Draw d;

    public Main(int width, int height) {
        this.setSize(width, height);
        this.setTitle("Ohrada");
        this.setLayout(new GridLayout(1,1));
        JPanel p=new JPanel();
        p.setBackground(Color.WHITE);
        p.setLayout(new BorderLayout());
        d=new Draw();
        p.add(d);
        this.getContentPane().add(p);
        this.setVisible(true);
      ...
   }
```

Pro `JPanel` zaregistrujeme posluchače, použijeme dvě rozhraní: `MouseListener` a `MouseMotionListener`. Všechny jejich metody musí být předefinovány. Nás budou zajímat metody `mousePressed()` a `mouseDragged()`. V prvním handleru inicializujeme souřadnice levého horního a pravého dolního rohu tak, aby byly totožné, ve druhém handleru měníme souřadnice pouze pravého dolního rohu. Pro získání polohy kurzoru použijeme metody `getX()` a `getY()`, které voláme pro parametry obou handlerů.

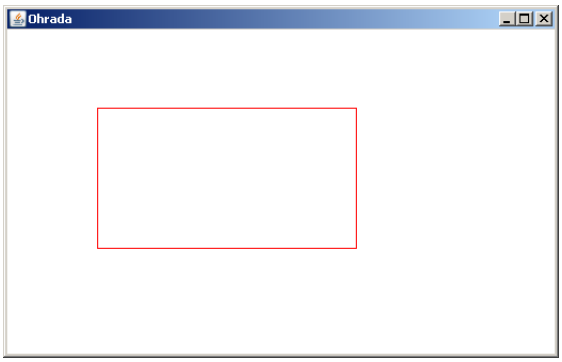
```
p.addMouseListener(new MouseListener()
```

```
    {
     public void mouseClicked(MouseEvent e) {}
     public void mouseEntered(MouseEvent e) {}
     public void mousePressed(MouseEvent e)
      {
       x1=e.getX();
       y1=e.getY();
       x2=e.getX();
       y2=e.getY();
       d.setRectangle(x1,y1, x2,y2);
      }

     public void mouseReleased(MouseEvent e){}
     public void mouseExited(MouseEvent e){}
    });

   p.addMouseMotionListener(new MouseMotionListener()
    {
      public void mouseDragged(MouseEvent e)
       {
        x2=e.getX();
        y2=e.getY();
        d.setRectangle(x1,y1, x2,y2);
        repaint();
       }

      public void mouseMoved(MouseEvent e) {}
     });
  }
```

Vykreslení budeme provádět v metodě `paintComponent()` třídy `Draw`. Hodnoty aktuálních souřadnice obou rohů jsou předávány metodou `setRectangle()`.

```
public class Draw extends JComponent{
    private int x1,y1, x2, y2;
    public Draw() {
    }
  public void paintComponent(Graphics g)
  {
   g.setColor(Color.RED);
   g.drawRect(x1,y1,x2-x1, y2-y1);
  }

  public void setRectangle(int x1, int y1,int x2, int y2)
  {
   this.x1=x1;
   this.y1=y1;
   this.x2=x2;
   this.y2=y2;
  }
}
```

Výsledek vypadá takto.



Obrázek 10.46: Vykreslení výběrové množiny.

Příklad 3. V následujícím příkladu se budeme pokoušet vytvořit program, který bude generovat úsečky s náhodnými hodnotami souřadnic koncových bodů a náhodnou hodnotou barvy. Souřadnice jednotlivých počátečních a koncových bodů budeme ukládat do dynamické datové struktury `ArrayList`. Počet úseček je znám předem, výpočet tedy proběhne v cyklu `for`. Při návrhu struktury tříd použijeme princip *kompozice* a to na základě úvahy "Má úsečka nějaké body?"

Nejprve vytvoříme třídu Bod, bude obsahovat dvě datové položky představující souřadnice x, y bodu.

```
public class Bod {
private int x,y;
    public Bod(int x, int y) {
       this.x=x;
       this.y=y;
    }
}
```

Úsečku budou tvořit počáteční a koncový bod, bude mít uloženy informace o barvě. Nové instance třídy `Usecka` budou vytvářeny za použití explicitního konstruktoru, každé úsečka si nese informace o svých krajních bodech a barvě. Aby bylo možno úsečky možno v cyklu vykreslit, ve třídě jsou implementovány metody (tj. gettery) vracející informace o počátečním, koncovém bodě a barvě úsečky.

```
public class Usecka {
  private Bod start, end;
  private Color col;

    public Usecka(int x1, int y1, int x2, int y2, Color col) {
      this.start=new Bod (x1, y1);
      this.end=new Bod (x2, y2);
      this.col=col;
    }

   public Bod getStart(){return this.start;}
   public Bod getEnd(){return this.end;}
   public Color getColor(){return this.col;}
}
```

Vlastní generování úseček provádí ve svém konstruktoru třída `Generovani`. Jako formální parametry jsou explicitnímu konstruktoru předávány následující údaje: počet úseček, šířka a výška `JPanelu`.Vygenerované úsečky jsou ukládány do ArrayListu typu `Usecky`. Pro generování náhodných čísel použijeme statickou metodu `random()`, která se nachází ve třídě `Math`. Ve třídě jsou dále implementovány metody vracející počet vygenerovaných úseček a jednu úsečku.

```
public class Generovani {

private ArrayList <Usecka> u;
    public Generovani(int kolik, int width, int height) {
      u=new ArrayList <Usecka>();
      for (int i=0; i<kolik;i++)
       {
        int x1=(int)(random()*width);
        int y1=(int)(random()*width);
        int x2=(int)(random()*width);
        int y2=(int)(random()*width);
        Color col=new  Color((int)(random()*255),
        (int)(random()*255), (int)(random()*255));
        u.add(new Usecka(x1,y1,x2,y2,col));
       }
    }
   public Usecka getUsecka(int index) {return this.u.get(index);}
   public int getPocet(){return u.size();}
}
```

Vykreslení bude probíhat v předefinované metodě `paintComponent()` třídy `Draw`, která je potomkem třídy `JComponent`. V konstruktoru třídě předáváme odkaz na instanci třídy `Generovani` obsahující ArrayList. Vykreslování probíhá po jednotlivých položkách.

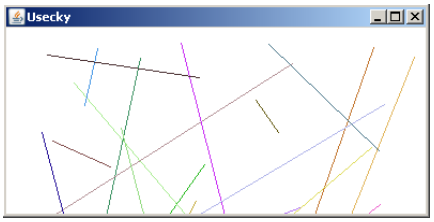
```
public class Draw extends JComponent{
Generovani gen;
    public Draw(Generovani gen) {
        this.gen=gen;
    }

    public void paintComponent(Graphics g)
    {
     for (int i=0;i<gen.getPocet();i++)
     {
      Usecka u=gen.getUsecka(i);
      g.setColor(u.getColor());
      g.drawLine(u.getStart().getX(),u.getStart().getY(),
              u.getEnd().getX(), u.getEnd().getY());
     }
    }
}
```

Ve třídě `Main`, která je potomkem třídy `JFrame` se postaráme o vytvoření potřebných formulářů. Vytvoříme novou instanci `g` třídy `Generovani` a inicializujeme ji vhodnými parametry. V dalším kroku vytvoříme novou instanci třídy `Draw`, předáme ji jako parametr odkaz na instanci g třídy, instanci třídy `Draw` přidáme na `JPanel` za použití metody `add()`. Celý panel poté přidáme na formulář.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

```
public class Main extends JFrame{
    public Main(int width, int height) {
    this.setSize(width, height);
    this.setTitle("Usecky");
    this.setLayout(new GridLayout(1,1));
    Generovani g=new Generovani(30,width, height);
    JPanel p=new JPanel();
    p.setBackground(Color.WHITE);
    p.setLayout(new BorderLayout());
    p.add(new Draw(g));
    this.getContentPane().add(p);
    p.add(new Draw(g));
    this.setVisible(true);
    }
    public static void main(String[] args) {
        new Main(400,200);
    }
}
```

Výsledek vypadá takto.



Obrázek 10.47: Vykreslení linií s náhodnými souřadnicemi i barvami.

Příklad by dále bylo možno vylepšit přidáním tlačítka generujícího po každém stisku novou posloupnost úseček. Na formulář bychom přidali tlačítko. Při umist'ování na formulář použijeme `BorderLayout`, panel bude `BorderLayout.CENTER`, tlačítko `BorderLayout.SOUTH`.

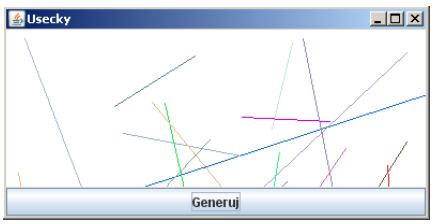
```
 JButton b =new JButton("Generuj");
    this.add(b, BorderLayout.SOUTH);
    b.addActionListener(new ActionListener()
    {
     public void actionPerformed(ActionEvent e)
     {
      Generuj(e);
     }
    });
```

Metoda `Generuj()` vypadá takto:

```
public void Generuj(ActionEvent e)
    {
     g=new Generovani(30,width, height);
     p=new JPanel();
```

```
 p.setBackground(Color.WHITE);
 p.setLayout(new BorderLayout());
 p.add(new Draw(g));
 this.getContentPane().add(p,BorderLayout.CENTER);
 p.add(new Draw(g));
 this.setVisible(true);
}
```

A výsledek.



Obrázek 10.48: Předchozí příklad doplněný tlačítkem.

10.1.9.3 Práce s obrázky

V programu často pracujeme s rastry. Java nativně pracuje s formáty JPG, PNG a GIF. Rastry mohou být připojovány jak z lokálního zdroje, tak ze vzdáleného počítače prostřednictvím URL. Metody pro práci s rastry nalezneme ve třídách `ImageIO`, `BufferedImage`, `ImageIcon`. Tato problematika je poměrně rozsáhlá, Java disponuje řadou nástrojů pro práci s rastry, omezíme se jen na velmi stručný popis základních možností, především s postupem pro načtení a uložení rastru. Práce s rastrem představuje zpravidla dva kroky: první tvoří jeho načtení, druhý jeho zobrazení.

Načtení rastru. Načtení rastru lze provést několika způsoby. Ukážeme dvě varianty, první s využitím třídy `BufferedImage`, druhou s využitím třídy `ImageIcon()`.

Načtení rastru z lokálního zdroje. Načtení nejprve provedeme z lokálně umístěného souboru. Nejprve vytvoříme instanci třídy `File` a inicializujeme ji cestou k souboru. Připomeňme, že při zápisu cesty musíme používat dvojitá lomítka, při použití jednoho lomítka by se jednalo o `ESCAPE` sekvenci.

```
File f=new File("C:\\zajic.png");
```

Vytvoříme odkaz na instanci třídy `BufferedImage`

```
BufferedImage im;
```

Rastr načteme statickou metodou třídy `ImageIO.read(File f)` a inicializujeme s ním odkaz im.

```
im=ImageIO.read(f);
```

Druhou možnost představuje využití třídy `ImageIcon`. Vytvoříme novou instanci třídy ImageIcon, v konstruktoru ji inicializujeme souborem s rastrem, konstruktor očekává na vstupu objekt typu `String`. Metodou `toString()` získáme název souboru.

```
ImageIcon i=new ImageIcon(f.toString());
```

V dalším kroku vytvoříme odkaz na instanci třídy `Image` a inicializujeme ji prostřednictvím metody `getImage()` načteme rastr.

```
Image im=i.getImage();
```

Načtení rastru ze vzdáleného zdroje. Při načítání rastru ze vzdáleného zdroje vytvoříme novou instanci třídy URL, kterou inicializujeme odkazem na rastr.

```
url=new URL("http://www.natur.cuni.cz/gis/tomas/1.jpg");
```

Další postup je stejný jako v předchozím případě

```
im=ImageIO.read(url);
```

Zobrazení rastru. Poté, co je rastr načten, je nutné ho zobrazit. Zpravidla se používá komponenta `JPanel`. Zobrazování rastru probíhá v předefinované metodě `paintComponent()` za použití metody `drawImage()`. Metoda je přetížena, je k dispozici celkem 6 prototypů této metody. Nejjednodušší varianta vypadá takto:

```
boolean drawImage(Image img,int x,int y,
   ImageObserver observer)
```

Metoda neupravuje velikost rastru, rastr se na komponentu nemusí vejít celý, v takovém případě je zobrazena pouze levá horní část rastru. Pokud bychom chtěli rastr zobrazit celý, můžeme buď formulář zvětšit, nebo umístit `JPanel` na `JScrollPane`. Proměnná `img` představuje odkaz na načtený rastr, x, y souřadnice levého horního rohu rastru na komponentě, observer objekt, který je uvědoměn, je -li rastr načten. V našem případě bude kód pro vykreslení vypadat takto:

```
public void paintComponent(Graphics g)
{
 g.drawImage(im,0,0,this);
}
```

Druhý prototyp, který uvedeme, obsahuje parametry width/height, které umožní stanovit novou velikost rastru. Rastr tedy můžeme dodatečně zvětšit či zmenšit.

```
boolean drawImage(Image img, int x, int y, int sirka, int vyska,
ImageObserver observer).
```

Uložení rastru. Rastr můžeme uložit za použití statické metody `ImageIO.write()`. Její prototyp vypadá takto:

```
write(RenderedImage im, String format, File soubor).
```

Práce s pixely S jednotlivými pixely rastru lze manipulovat prostřednictvím metody `setRGB(int x, int y, int color)`. Metoda umožňuje nastavit pixelu zvolenou barvu, barva je vyjádřena prostřednictvím hodnoty int. Rastr se chová jako matice, x představuje řádkový index, y sloupcový index. Hodnotu barvy ve zvoleném pixelu lze získat metodou `int getRGB(int x, int y)`, metoda vrací číslo typu int. Chceme -li získat hodnoty jednotlivých barevných složek, použijeme následující kód:

```
c=getRGB(x,y);
int  red = (c & 0x00ff0000) >> 16;
int  green = (c & 0x0000ff00) >> 8;
int  blue = c & 0x000000ff;
```

Velikost rastru lze zjistit metodami `getWidht()` a `getHeight()`. Používají se, pokud chceme v cyklu procházet rastrem po pixelech.

V následujícím příkladu otevřeme rastr nacházející se ve vzdáleném zdroji. Rastr zobrazíme, všem pixelům nastavíme hodnotu barevných složek následovně $R_{nov} = R/3$, $G_{nov} = G$, $B_{nov} = B/3$. Načtení rastru provedeme v metodě `loadImage()`.

```
public void loadPicture()
 {
   try
    {
     url=new URL("http://www.natur.cuni.cz/gis/tomas/1.jpg");
     im=ImageIO.read(url);
    }
   catch (Exception e)
    {
     e.printStackTrace();
    }
 }
```

Pro získání hodnot barevných složek použijeme metodu `getRGB()`. Na základě požadavku zadání upravíme hodnoty barevných složek R a B, vytvoříme novou instanci třídy `Color` a pomocí `getRGB()` převedeme její hodnotu na číslo typu int, tuto hodnotu nastavíme pro pixel metodou `setRGB()`.

```
public void editPicture()
  {
   for (int i=0;i<im.getWidth();i++)
    {
      for (int j=0;j<im.getHeight();j++)
       {
        int c=im.getRGB(i,j);
        int  red = (c & 0x00ff0000) >> 16;
        int  green = (c & 0x0000ff00) >> 8;
        int  blue = c & 0x000000ff;
        red=red/3;
        blue=blue/3;
        Color ce=new Color(red,green,blue);
        im.setRGB(i,j,ce.getRGB());
       }
    }
 }
```

Upravený rastr následně uložíme

```
public void savePicture()
 {
  File soubor=new File("C::\\Temp\\Pokus.jpg");
  try
   {
    ImageIO.write(im, "jpg", soubor);
   }
  catch (Exception e)
  {
   e.printStackTrace();
  }
}
```

Pro zobrazení používáme metodu `drawImage()`, velikost rastru nastavíme stejnou jako velikost `JPanelu`, na které vykreslení rastru provádíme. Pro nastavení velikosti použijeme explicitní konstruktor.

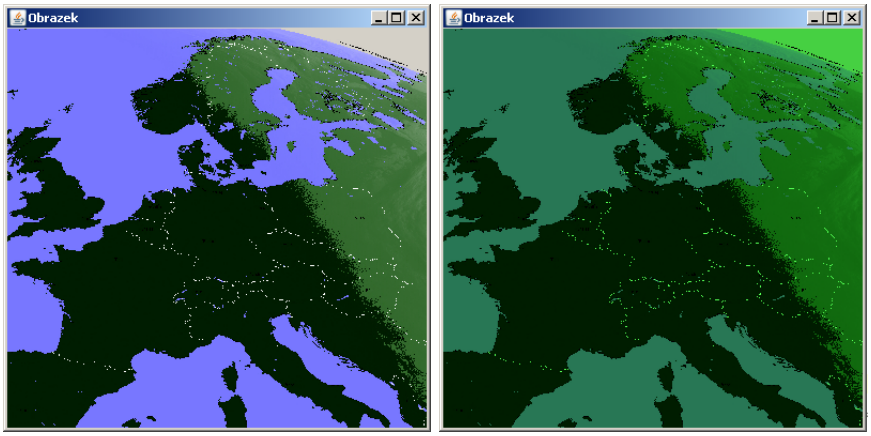
```
public class Obrazek extends JComponent{
private int width, height;
private URL url;
private BufferedImage im;
   public Obrazek(int width, int heigth)
   {
    this.width=width;
    this.height=heigth;
   }

   public void loadPicture()
   {
     try
      {
       url=new URL("http://www.natur.cuni.cz/gis/tomas/1.jpg");
       im=ImageIO.read(url);
      }
     catch (Exception e)
      {
       e.printStackTrace();
      }
   }

 public void editPicture()
 {
  for (int i=0;i<im.getWidth();i++)
   {
     for (int j=0;j<im.getHeight();j++)
      {
       int c=im.getRGB(i,j);
       int  red = (c & 0x00ff0000) >> 16;
       int  green = (c & 0x0000ff00) >> 8;
       int  blue = c & 0x000000ff;
```

```
          red=red/3;
          blue=blue/3;
          Color ce=new Color(red,green,blue);
          im.setRGB(i,j,ce.getRGB());
         }
       }
      }

     public void savePicture()
     {
      File soubor=new File("C::\\Temp\\Pokus.jpg");
     try
      {
       ImageIO.write(im, "jpg", soubor);
      }
     catch (Exception e)
     {
      e.printStackTrace();
     }
     }

     public void paintComponent(Graphics g)
     {
      g.drawImage(im,0,0, width,height, this);
     }
}
```

Nyní vytvoříme formulář s grafickým návrhem, rastr zobrazíme v `JPanelu`.

```
public class MainObrazek extends JFrame {

    public MainObrazek(int width, int height) {
     this.setSize(width, height);
     this.setTitle("Obrazek");
     JPanel p=new JPanel();
     p.setLayout(new BorderLayout());
     Obrazek o=new Obrazek(width, height);
     o.loadPicture();
     o.editPicture();
     p.add(o);
     this.getContentPane().add(p);
     this.setVisible(true);
    }
   public static void main(String [] arg)
   {
    new MainObrazek(400,400);
   }
}
```

Ukázka původního a upraveného rastru.



Obrázek 10.49: Vlevo původní, vpravo upravený rastr.

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"

Kapitola 11

Aplety

Aplet (v anglickém jazyce applet) představuje aplikaci menšího rozsahu umístěnou do WWW strany, která běží v okně prohlížeče. Aplet je spouštěn po načtení WWW strany, nemůže tedy běžet samostatně bez prohlížeče. Dříve aplety představovaly jedno z typických využití Javy, v současné době se Java používá na "serióznější" typy aplikací. Aplety jsou proto používány spíše pro oživení WWW stránek či ukázku jednoduchých algoritmů. Aplety, na rozdíl od běžných aplikací, mají některá omezení bezpečnostního charakteru:

- aplet nemůže spouštět programy na domovském serveru.
- aplet nemůže zapisovat do souborů v počítači, na kterém je spuštěn.
- aplet nemůže navazovat spojení jinam než na domovský server.

Aplety mohou být digitálně *podepsány*, v takovém případě některá z bezpečnostních omezení neplatí. Problematikou digitálně podepsaných apletů se tento materiál nezabývá. Pokud budeme aplet překládat, je nutné vzít v úvahu, že uživatel nemusí mít k dispozici poslední verzi JDK, a applet mu nemusí fungovat. Aplety lze vytvářet nad grafickým rozhraním AWT i Swing. Rozhraní AWT je podporováno všemi prohlížeči, o rozhraní Swing to neplatí.

Navzdory naším snahám bude v každém prohlížeči aplet vypadat trochu jinak. V této kapitole se budeme zabývat pouze aplety vytvořenými v grafickém rozhraní Swing. Applety mají některé zajímavé vlastnosti: mohou přehrávat multimediální soubory, mohou volat veřejné metody jiných appletů umístěných na stejné WWW straně.

11.1 Struktura appletu

Aplet může být tvořen jednou nebo více třídami, výchozí třída je odvozená od třídy javax.Swing.JApplet. Opět platí zásada, že jméno třídy musí být totožné se jménem souboru, ve kterém je třída uložena. Instance třídy JApplet představuje prázdný formulář, na něj mohou být přidávány další komponenty. Applet má podobnou strukturu jako "běžný" program, liší se na "první pohled" tím, že nemusí obsahovat metodu `main()`. Odlišností je však více, jak si časem ukážeme.

11.1.1 Životní cyklus appletu

Aplet se v průběhu svého běhu dostává do několika stavů, které jsou doprovázeny voláním některých speciálních metod. Tyto stavy jsou ovlivněny nikoliv appletem samotným, ale prohlížečem. Bývají označovány

literárním tvarem "životní cyklus" apletu. Metody jsou ve většině případů deklarovány jako public, uživatelem by měly být předefinovány.

Konstruktor. Konstruktor apletu je volán poté, co byla načtena www stránka s apletem. Konstruktory nejsou v appletech téměř používány, jejich funkce nahrazují jiné metody, např. `init()`.

metoda `init()`. Nejdůležitější metoda, je volána po vykonání konstruktoru. Jejím cílem je provedení inicializace apletu, supluje tedy funkci konstruktoru. Kód v ní by měl být chráněn pro případ vzniku výjimky, nejčastěji je používána konstrukce try-catch. Metoda je volána pro aplet pouze jedenkrát. Každý applet by měl obsahovat metodu `init()`.

metoda `start()`. Po provedení metody `init()` je applet spuštěn metodou `start()`. Metoda je vykonána vždy, když se aplet stane v okně prohlížeče viditelný. Na rozdíl od předchozí metody může být pro aplet volána opakovaně.

metoda `stop()`. Činnost apletu je metodou stop ukončena. Dojde k tomu, když aplet přestane být v okně prohlížeče viditelný. Pokud se aplet stane opět viditelný, je vykonána metoda `start()`, která aplet spustí. Metoda `start()` může být pro aplet volána opakovaně, je vlastně protějškem metody `stop()`. Používá se často v případě, kdy applet provádí výpočetně náročné činnosti, např. animace.

metoda `destroy()`. Metoda je volána při uvolněním apletu z paměti, má podobnou funkci jako destruktor. Pro aplet je volána pouze jednou.

metoda `paint(Graphics g)`. Její funkce je podobná jako v případě metody `paintComponent()`. Metoda je volána v případě, kdy je nutno překreslit obsah okna. Na objekt třídy Graphics lze kreslit běžnými metodami.

metoda `jbInit()`. Metoda, která je deklarována jako privátní, je používána při inicializaci komponent.

11.2 Tvorba appletu

V této kapitole se budeme snažit vytvořit jednoduchý aplet, který provede sečtení dvou čísel zadaných do textfieldů. Spuštění výpočtu se provede stiskem tlačítka; jedná se o modifikaci příkladu z kapitoly 1.8.5.

Vytvoříme třídu `AppletScitani`, která bude potomkem třídy `JApplet`. Datové položky třídy budou stejné.

```
public class AppletScitani extebds JApplet{
    private int prvni, druhe, soucet;
    private JLabel l1,l2,l3;
    private JTextField t1, t2;
    private JButton b;
public Form() {};
```

Všimněme si, že v konstruktoru není umístěn žádný výkonný kód. Vytvoříme metodu `init()`, která bude obsahovat výkonný kód umístěný v chráněném bloku try-catch pro případ, že by inicializace neproběhla úspěšně. Velikost appletu je nastavena na 200 x 100 pixelů.

```
public void init() {
    try
     {
      this.setSize(200,100);
      this.setLayout(new GridLayout(3,2));
      l1=new JLabel("Prvni cislo");
      l2=new JLabel("Druhe cislo");
      t1=new JTextField();
      t2=new JTextField();
      t1.setText("0");
      t2.setText("0");
      b=new JButton("Soucet");
      l3=new JLabel();
      this.getContentPane().add(l1);
      this.getContentPane().add(l2);
      this.getContentPane().add(t1);
      this.getContentPane().add(t2);
      this.getContentPane().add(b);
      this.getContentPane().add(l3);
      this.setVisible(true);
```

Za použití metody `setText()` inicializujeme výchozí hodnoty v textfieldech na 0, v opačném případě by mohlo dojít k výjimce. Práce s událostmi bude řešena stejně. Hodnoty obou sčítanců budou získány při události `FocusEvent`, vlastní sčítání proběhne po stisku tlačítka, výsledný text bude zobrazen v labelu.

```
      t1.addFocusListener(new FocusListener() {
         public void focusLost(FocusEvent e) {
                prvniZadej(e);
          }
          public void focusGained(FocusEvent e){};
        });
      t2.addFocusListener(new FocusListener() {
         public void focusLost(FocusEvent e) {
               druheZadej(e);
            }
            public void focusGained(FocusEvent e){};
        });

       b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
               soucet(e);
            }
       });
     }

    catch (Exception e)
     {
      e.printStackTrace();
     }
    }

    public void prvniZadej(FocusEvent e) {
```

```
            this.prvni=Integer.parseInt(t1.getText());
        }

        public void druheZadej(FocusEvent e) {
            this.druhe=Integer.parseInt(t2.getText());
        }

        public void soucet(ActionEvent e) {
            soucet=this.prvni+this.druhe;
            l3.setText(String.valueOf(soucet));
        }
    }
```

Všimli jsme si, že oproti "klasické" aplikaci se zdrojový kód takřka neliší. Pouze kód, který byl původně umístěn v konstruktoru, přešel do metody `init()`. V našem případě jsme nepoužili metodu `stop()`, applet neprováděl žádné výpočetně náročné operace.

11.3 Spuštění appletu

Applet neobsahuje metodu `main()`, nemůže být spouštěn klasicky z příkazového řádku. Aby aplet mohl běžet, je nutné nejprve vytvořit HTML soubor, který bude obsahovat příkaz pro spuštění appletu. Aplet je volán uvedením názvu přeložené třídy z tagu <APPLET>, musí obsahovat některé další informace, např. údaje o velikosti apletu. Pokud vytváříme aplety v některém z vývojových prostředí, jsou příslušné HTML soubory generovány automaticky, nemusíme je vytvářet "ručně". Jméno HTML souboru může být libovolné, nijak nesouvisí s apletem. V jednom souboru může být více než jeden aplet.

Následující příklad ukazuje jednoduchou WWW stránku s kódem pro spuštění apletu.

```
<HTML>
<HEAD>
<TITLE>
Sčítání dvou čísel.
</TITLE>
</HEAD>
<BODY>
Applet provádějící sčítání dvou čísel. <BR>
<APPLET
CODEBASE="."
CODE="AppletScitani.class"
NAME="Muj prvni applet"
WIDTH=200
HEIGHT=100
HSPACE=0
VSPACE=0
ALIGN=left
</APLET>
</BODY>
</HTML>
```

Popisem HTML kódu se zabývat nebudeme, uvedeme význam jednotlivých položek umístěných uvnitř tagu `<APPLET>`. Všechny položky nejsou povinné, každý applet by *měl* obsahovat položky `<CODEBASE>`, `<CODE>`, `<WIDTH>`, `<HEIGHT>`.

`CODEBASE.` Tato položka obsahuje cestu (ve formě URL), ve které se nacházejí jednotlivé přeložené třídy (popř. JAR soubory). Nemusejí být umístěny standardně, tj. ve stejném podadresáři jako HTML soubory, ale v libovolné složce. Pokud se nacházejí ve stejné složce, použijeme označení aktuálního adresáře ".", nebo tuto položku můžeme vynechat. Jednotlivé úrovně cesty jsou oddělovány znakem /.

`CODE.` Představuje jméno třídy appletu, která bude spuštěna. Standardně se jedná o třídu, která je potomkem třídy JApplet.

`NAME.` Představuje jméno apletu, lze využít při vzájemné komunikaci dvou apletů.

`WIDTH.` Šířka okna apletu v pixelech.

`HEIGHT.` Šířka okna apletu v pixelech.

`HSPACE.` Odsazení apletu v pixelech zleva.

`VSPACE.` Odsazení apletu v pixelech shora.

`ALIGN.` Zarovnání apletu vzhledem k WWW stránce. K dispozici jsou následující možnosti: `left, right, top, bottom,absmiddle`.

`PARAM.` Aplet může načítat externí parametry a jejich hodnoty. K dispozici jsou následující hodnoty: `NAME` a `VALUE.` Jejich načtení je prováděno z vlastního appletu metodou `getParametr()`. Kód v tagu `<APPLET>`.

```
<PARAM NAME="file" VALUE"applet/picture.jpg">
```

Načtením parametru získáme jeho hodnotu, kód ve vlastním appletu vypadá takto.

```
String filename=getParametr(file);
```

`ALT.` Text, který je zobrazen v případě, kdy prohlížeč nemá možnost spouštět aplety.

`ARCHIVE.` Název JAR archivu (viz dále). Tento parametr se často používá používá u apletů pracujících s GUI.

11.3.1 Java archivy

Zkratka JAR představuje Java Archiv. Představuje způsob distribuce souborů *.class, které jsou algoritmem ZIP zkomprimovány do jednoho souboru. Součástí JAR mohou být i další soubory, např. rastry či multimediální soubory, konfigurační soubory, atd... Používání archivů přináší několik výhod, velice stručně se s nimi seznámíme:

1. Zmenšení velikosti dat. Tato výhoda se projeví zejména u apletů, kdy přenášíme menší množství dat. Využitím komprese se velikost souboru sníží cca. na polovinu, důsledkem je rychlejší načtení appletu.

2. Všechny potřebné soubory jsou ”sbaleny” do jednoho archivu. Týká se to zejména případů, kdy program využívá anonymní vnitřní třídy, tj. při použití grafického rozhraní. Překladem takového programu vzniká “větší” počet přeložených souborů class.
3. Jméno Java archivu není závislé na názvu jednotlivých tříd, může být pojmenován podle našich představ.
4. Jednoduché spouštění programu. Pro spuštění Java archivu není nutno tento archiv rozbalovat, lze ho spustit jediným příkazem. Archiv s názvem Pokus lze spustit buď dvojklikem na název archivu nebo z příkazové řádky takto:

```
java -jar "Pokus.jar"
```

Popišme pro úplnost i druhou variantu spuštění programu: soubory obsažené v Java archivu nejprve rozbalíme a následně spustíme. Tento postup však není v praxi používán, je poněkud nepohodlný.
Uveďme přehled příkazů používaných pro práci s JAR soubory.

Vytvoření archivu. Pro vytvoření archivu je použit příkaz

```
java -jar cf nazev_archivu jmena_souboru
```

Název archivu musí být uveden včetně přípony jar. Jména souborů mohou být oddělena čárkami nebo lze použít masku souboru ve tvaru hvězdičkové konvence.

```
java -jar cf archiv.jar main.class,form.class
java -jar cf archiv.jar *.class
```

První příkaz vytvoří archiv ze dvou zadaných souborů, druhý ze všech souborů s příponou *.class v aktuálním podadresáři.

Rozbalení archivu. Rozbalení archivu včetně automatického vytvoření všech potřebných podadresářů lze provést příkazem

```
java -jar xf nazev_archivu
```

Výpis souborů v archivu. Soubory a adresáře obsažené v archivu lze vypsat příkazem

```
java -jar tf nazev_archivu
```

11.3.2 JAR soubory a aplety

Java aplety obsahující komponenty GUI jsou často distribuovány ve formě JAR souborů. V takovém případě je do `HTML` souboru nutno přidat sekci `ARCHIVE` s uvedením jména archivu. Položka `CODE` bude stále obsahovat spouštěcí třídu apletu.

```
CODE="AppletScitani.class"
ARCHIVE="archiv.jar"
```

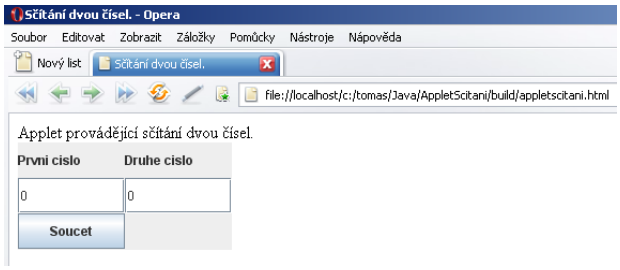
Archiv se může nacházet i v některém z podadresářů, musíme k němu tedy uvést cestu

```
CODE="AppletScitani.class"
ARCHIVE="applets/archiv.jar"
```

Pozor ! Pokud vytváříme aplet za použití GUI builderů, při návrhu metodou drag and drop jsou mnohdy používány nestandardní konstrukce či komponenty, příslušné knihovny musí být připojeny k distribuované aplikace, v opačném případě by nešla spustit. Typickým příkladem je vývojové prostředí *NetBeans*, které využívá tzv. swing layout. K naší aplikaci musíme připojit archiv `swing-layout-1.0.jar`. Zpravidla ho přibalujeme do stejné složky, ve které se nacházejí přeložené soubory.

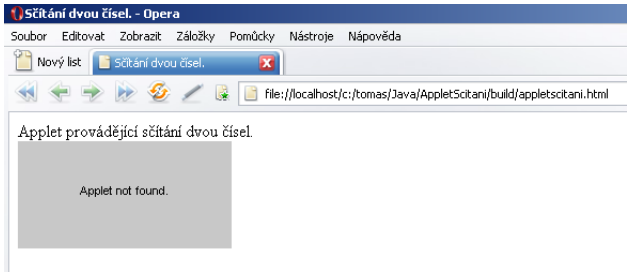
```
ARCHIVE="swing-layout-1.0.jar"
```

Spuštění appletu Applet spustíme prostřednictvím příslušného HTML souboru. Pokud jsme všechny potřebné kroky vykonali správně, výsledek může vypadat takto.



Obrázek 11.1: Ukázka apletu v prohlížeči Opera.

Pokud jsme někde udělali chybu (nejčastěji špatné zadání cesty k apletu), výsledek bude vypadat takto:



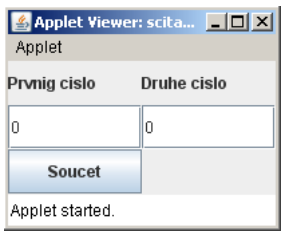
Obrázek 11.2: Problém při práci s apletem.

11.3.3 Applet viewer

Práce s appletem ve webovém prohlížeči není příliš pohodlná. Její největší nevýhodou je fakt, že případné změny ve zdrojovém kódu appletu se v prohlížeči neprojeví ani při opakovaném načtení www strany s apletem. Prohlížeč tedy musíme restartovat. Existuje aplikace, která se používá pro zobrazování appletů s názvem appletviewer.
Appletviewer se používá pro zobrazení appletu uvedeného HTML kódu stránky, veškerý HTML kód strany ignoruje. Spustíme ho následujícím příkazem

```
appletviewer nazev_html_souboru.
```

Applet viewer se tedy používá jako ladicí nástroj pro zobrazování appletů v průběhu jejich návrhu. Aplet zobrazený v applet vieweru vypadá takto:



Obrázek 11.3: Aplet provádějící součer dvou čísel.

Appletviewer můžeme ošálit. Pokud do zdrojového kódu přidáme na jeho začátek ve formě jednořádkových komentářů následující text

```
//<APPLET
//CODEBASE="."
//CODE="AppletScitani.class"
//NAME="Muj prvni applet"
//WIDTH=200
//HEIGHT=100
//HSPACE=0
//VSPACE=0
//ALIGN=left
//</APLET>
```

budeme moci appletviewererem spouštět přímo soubor AppletScitani.class.

11.3.4 Spouštění appletu jako aplikace

Často požadujeme, aby program bylo možno spustit jako aplet i aplikaci. Tento požadavek lze vyřešit velmi jednoduše, do appletu přidáme metodu `main()`. Tato metoda je appletem ignorována, zároveň však umožní spouštění apletu jako aplikace. Je vhodné připomenout, že aplet pro kreslení využívá metodu `paint()`, aplikace `paintComponent()`. Pokusíme se vytvořit aplikaci generující graf funkce sinus ve formě apletu i aplikace.

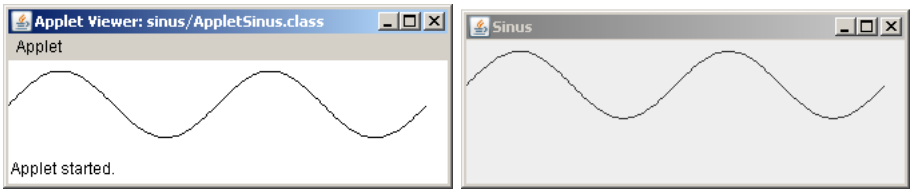
```
public class AppletSinus extends JApplet {

    public AppletSinus() {}

    public void initcomponents() {}

    public void paint(Graphics g) {
        double step=0.1, x,y,xo,yo;
        x=100/3;
        xo=x;  y=0;  yo=0;
        for (double i=0;i<4*Math.PI;i=i+step) {
            y=i*200/8;
            x=100/3-Math.sin(i)*200/8;
```

```
            g.drawLine((int)yo,(int)xo,(int)y,(int)x);
            xo=x;  yo=y;
        }
    }

    public static void main(String[] args) {
      JFrame f=new JFrame();
      f.setTitle("Sinus");
      f.setVisible(true);
      f.getContentPane().add(new Draw())
    }
}
```

Všimněme si, že v tomto případě metoda `init()` neobsahuje žádný kód. Není potřeba vytvářet žádné komponenty ani ošetřovat události. Třída `Draw` provádějící kreslení vypadá takto:

```
public class Draw extends JComponent{
public void paintComponent(Graphics g) {
        double step=0.1, x,y,xo,yo;
        x=100/3;
        xo=x;  y=0;  yo=0;
        for (double i=0;i<4*Math.PI;i=i+step) {
            y=i*200/8;
            x=100/3-Math.sin(i)*200/8;
            g.drawLine((int)yo,(int)xo,(int)y,(int)x);
            xo=x;  yo=y;
        }
    }
}
```

Applet i aplikaci si můžeme prohlédnout na následujících obrázkách.



Obrázek 11.4: Aplet generující sinusoidu. Vlevo v aplet vieweru, vpravo v prohlížeči.

Kapitola 12

Java a databáze

Podobně, jako většina vyšších jazyků, umí i Java pracovat s databázovými systémy. K tomu, abychom mohli z Javy přistupovat k databázovému serveru, ukládat a načítat data, potřebujeme JDBC (Java DataBase Connectivity). JDBC je API pro programátory v programovacím jazyku Java, které definuje jednotné rozhraní pro přístup k relačním databázovým systémům. JDBC je součástí Javy SE od JDK 1.1. Pro přístup ke konkrétnímu databázovému serveru je potřeba JDBC ovladač, který poskytuje tvůrce databázové platformy.

K přístupu k jednotlivým databázovým serverům nám tedy slouží JDBC ovladače, které jsou specifické pro každou databázovou platformu:

* **MySQL**: `com.mysql.jdbc.Driver`,

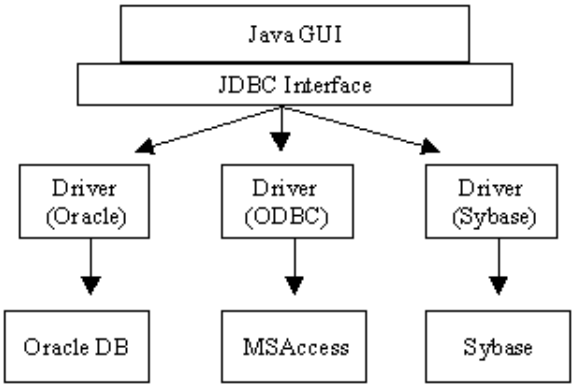
* **Oracle**: `oracle.jdbc.driver.OracleDriver`,

* **MS Access**: `sun.jdbc.odbc.JdbcOdbcDriver`,

a připojovací řetězec, který je podobně jako ovladač závislý na použité datbázové platformě:

* **MySQL**: `jdbc:mysql://server/jméno_databáze?characterEncoding=UTF-8`

* **Oracle**: `jdbc:oracle:thin:@server:port:jméno_databáze`



Obrázek 12.1: Schéma komunikace Javy s databázovým serverem.

* **MS Access**: `jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)}; DBQ="[soubor databáze]; DriverID=22; READONLY=false}`

Nyní se pokusíme vytvořit jednoduchou konzolovou aplikaci, která bude pracovat s databázovým systémem. Cílem aplikace bude správa (ukládání, mazání, atd.) bodů definovaných pravoúhlými souřadnicemi x, y. Úložištěm souřadnic nebudou datové položky třídy, ale databázový systém. MySQL[1] bude pro tuto jednoduchou aplikaci plně dostačující.

Pro komunikaci s databázovým serverem budeme potřebovat vhodný ovladač. Ovladače jsou k dispozici na webové adrese `http://dev.mysql.com/downloads/connector/`. Pro naši aplikaci je vhodný ovladač Connector/J 5.0, který je třeba stáhnout a začlenit do aplikace.

Přidání ovladače. Přidání a začlenění ovladače pro konunikace s databázovým serverem lze provést kliknutím pravého tlačítka na název projektu a položku *Properties*. V záložce *Libraries* přidáme pomocí tlačítka *Add JAR/Folder* soubor ovladače pro komunikaci MySQL s Javou. V tomto případě se jedná o soubor `mysql-connector-java-5.0.7-bin.jar`.

Potvrzením se vybraná knihovna ovladače přidá do `CLASSPATH` projektu. Bez knihovny ovladače přidané do `CLASSPATH` se nám nepodaří připojit k databázovému serveru MySQL![2]

12.1 Komunikace s databázovým systémem

Základem aplikace bude komunikace s databázovým systémem MySQL. Nadefinujeme třídu `MySQLDatabase`, která bude mít několik proměnných: `server`, `db`, `user`, `password` a `encoding`. Do těchto proměnných budeme ukládat parametry konstruktoru třídy. Pokud je MySQL nainstalováno na lokálním počítači, bude se serverem `localhost` nebo IP adresa `127.0.0.1`. Pokud MySQL na lokálním počítači nainstalováno není, je třeba zadat IP adresu nebo název vzdáleného počítače, na kterém je MySQL nainstalováno. Musíme též nadefinovat databázi, ke které se budeme připojovat, tu budeme ukládat do proměnné `db`. `User` a `password` reprezentují přihlašovací údaje a `encoding` umožňuje nastavit kódování, v kterém budeme komunikovat s databází (týká se práce s řetězci).

```
public class MySQLDatabase {

    private String server;
    private String db;
    private String user;
    private String password;
    private String encoding;


    public MySQLDatabase(String server, String db, String user, String
        password, String encoding) {
        this.server = server;
        this.db = db;
        this.user = user;
        this.password = password;
        this.encoding = encoding;
```

[1] Jde o „malou" volně stažitelnou open-source databázovou platformu. Součástí instalace je navíc velmi šikovný administrátor phpMyAdmin.

[2] Uvedený postup platí pouze pro vývojové prostředí NetBeans. V ostatních vývojových prostředích je však postup většinou velmi podobný.

```
    }
}
```

Konstruktor je velmi jednoduchý, slouží k inicializaci datových položek třídy. Pokusíme se nyní definovat metodu, pomocí které bychom se mohli připojit k MySQL. Třídy a metody pro práci s databázovými systémy jsou k dispozici v balíčku `java.sql`. Součástí tohoto balíčku je i třída `Connection`, jejíž instanci bude vracet metoda `getConnectionToMySQL()`. Tato metoda tedy naváže kontakt s databázovým serverem a vrátí kontakt v podobě objektu třídy `Connection`.

```
public Connection getConnectionToMySQL() {
        Connection conn = null;
        try {
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            conn = DriverManager.getConnection("jdbc:mysql://"+
              this.server +"/" + this.db + "?characterEncoding=" +
              this.password + "&user=" + this.user + "&password=" +
              this.password);
        } catch (SQLException ex) {
            ex.printStackTrace();
        } catch (ClassNotFoundException ex) {
            ex.printStackTrace();
        } catch (InstantiationException ex) {
            ex.printStackTrace();
        } catch (IllegalAccessException ex) {
            ex.printStackTrace();
        }
        return conn;
    }
```

Všimněme si, kolik výjimek je zapotřebí ošetřit. Výjimka `SQLException` se týká většiny tříd z balíčku `java.sql`. Výjimky `ClassNotFoundException`, `InstantiationException`, `IllegalAccessException` se týkají ovladače JDBC.

12.2 Získávání metadat o databázi

Abychom co nejlépe ošetřili ukládání bodů do databáze, vytvoříme si ve třídě `MySQLDatabase` metodu, která bude zjišťovat, zda v databázi existuje tabulka daného jména. Metoda `existTable()` bude vracet buď hodnotu `true` nebo `false` a parametrem bude proměnná `tableName` typu `String`.

```
public boolean existTable(String tableName) {
        Connection conn = null;
        ResultSet tables;
        boolean existuje = false;

        try {
            conn = getConnectionToMySQL();
            DatabaseMetaData dbm = conn.getMetaData();
            tables = dbm.getTables(null, null, tableName, null);
            if (tables.next()) {
```

```
                existuje = true;
            }
            else {
                existuje = false;
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException ex) {
                    ex.printStackTrace();
                }
                conn = null;
            }

        }
        return existuje;
    }
```

Metoda `existTable()` se nejprve pomocí metody `getConnectionToMySQL()` připojí k MySQL. Poté vytvoří objekt třídy `DatabaseMetaData` pomocí metody `getMetaData()`. Jak název napovídá třída `DatabaseMetaData` je určena pro získávání informací o databázi, ke které jsme připojeni. Metoda `getTables()` zjišťuje, zda se v databázi nalézá tabulka, daná parametrem metody. Třída `DatabaseMetaData` ale obsahuje další užitečné metody:

- `getAttributes()` - metoda vrací seznam atributů dané tabulky,
- `getPrimaryKeys()` - metoda vrací seznam atributů dané tabulky, které jsou zároveň primární klíče,
- `supportsTransactions()` - metoda zjistí, zda databázový systém podporuje transakce,
- `supportsSavepoints()` - metoda vrací hodnotu true, pokud databázový systém při neúspěšné transakci podporuje návrat do nějakého stavu (nastaveného pomocí bodu návratu - savepointu) v průběhu transakce.

Tímto máme vytvořenu třídy MySQLDatabase, která nám bude usnadňovat tvorbu třídy bodů.

12.3 Vkládání dat do databáze

Třída Point bude obsahovat dvě datové položky třídy, explicitní konstruktor a několik metod. Datovými položkami instance budou:

```
private int ID_bodu;
private MySQLDatabase db = new MySQLData-
base("localhost", "points", "root", "", "UTF-8");
```

V proměnné ID_bodu bude uložena hodnota jedinečná pro každý bod a my ji podle této proměnné v databázi bodů jednoznačně identifikujeme. Tvorbu těchto jedinečných hodnot necháme na straně databáze:

```
ID_bodu INTEGER NOT NULL AUTO_INCREMENT, PRIMARY KEY (ID_bodu)
```

Tímto se nám pro každý bod v databázi vytvoří jedinečné číslo představující identifikátor bodu. Abychom ale toto číslo mohli uložit do datové položky třídy, je třeba použít metodu třídy Statement se dvěma parametry:

```
stmt.executeUpdate("...", Statement.RETURN_GENERATED_KEYS);
rset = stmt.getGeneratedKeys();
rset.next();
this.ID_bodu = rset.getInt(1);
```

Prvním parametrem je SQL příkaz, druhým je objekt třídy Statement. Pomocí metody `getGeneratedKeys()` umožňuje návrat `ResultSetu` s automaticky vygenerovanými hodnotami.

Konstruktor třídy `Point` má tři parametry - souřadnice bodu x, y, z. Tyto souřadnice jsou přímo v těle konstruktoru ukládány do databáze:

```
public Point(double x, double y, double z) {
        // testovani, zda existuje tabulka 'body', pokud ne, jeji vytvoreni
        if (db.existTable("body")) {
            Connection conn = null;
            Statement stmt = null;
            ResultSet rset = null;
            try {
                conn = db.getConnectionToMySQL();
                stmt = conn.createStatement();
                stmt.executeUpdate("INSERT INTO body (X, Y, Z) VALUES ("
                  + x + ", " + y + ", " + z + ");",
                  Statement.RETURN_GENERATED_KEYS);
                rset = stmt.getGeneratedKeys();
                rset.next();
                this.ID_bodu = rset.getInt(1);
            } catch (SQLException ex) {
                ex.printStackTrace();
            }
            finally {
                try {
                    conn.close();
                    stmt.close();
                    rset.close();
                } catch (SQLException ex) {
                    ex.printStackTrace();
                }
            }
        }
        else {
            Connection conn = null;
            Statement stmt = null;
            ResultSet rset = null;
```

```
            try {
                conn = db.getConnectionToMySQL();
                stmt = conn.createStatement();
                stmt.executeUpdate("CREATE TABLE body (ID_bodu INTEGER
                  NOT NULL AUTO_INCREMENT, X DOUBLE PRECISION, Y DOUBLE
                  PRECISION, Z DOUBLE PRECISION, Datum_zalozeni
                  TIMESTAMP NOT NULL, Popis VARCHAR(254), PRIMARY KEY
                  (ID_bodu))");
                stmt.executeUpdate("INSERT INTO body (X, Y, Z) VALUES
                  (" + x + ", " + y + ", " + z + ");",
                  Statement.RETURN_GENERATED_KEYS);
                rset = stmt.getGeneratedKeys();
                rset.next();
                this.ID_bodu = rset.getInt(1);
            } catch (SQLException ex) {
                ex.printStackTrace();
            }
            finally {
                try {
                    conn.close();
                    stmt.close();
                    rset.close();
                } catch (SQLException ex) {
                    ex.printStackTrace();
                }
            }

        }
    }
```

Tělo konstruktoru se hned na začátku větví na dva případy. Využívá `existTable()` třídy `MySQLDatabase` ke zjištění, zda v databázi `points` existuje tabulka `body`, do níž se budou ukládat souřadnice vytvořeného bodu. Pokud tabulka existuje, je navázáno připojení k databázi a uloženo do objektu `conn`. Pomocí tohoto objektu a metody `createStatement()` je poté vytvořen objekt `stmt` třídy `Statement`. Metody této třídy umožňují komunikovat s databází pomocí SQL příkazů. Třída `Statement` obsahuje několik základních metod:

- `executeQuery()` - Metoda je určena pro SQL příkazy, jejichž výsledkem je relace (tabulka). Návratovým typem je objekt typu `ResultSet`, pomocí něhož poté můžeme přistupovat k samotným datům. Příklad:
  - `stmt.executeQuery("SELECT X, Y, Z FROM body WHERE ID_bodu = 2;");`
- `executeUpdate()` - Metoda, jejímž parametrem jsou SQL příkazy pro vkládání, aktualizaci a mazání záznamů v databázi. Tato metoda též vyhodnotí SQL příkazy pro vytváření nových tabulek či změnám v jejich struktuře. Návratovým typem metody `executeUpdate()` je integer. Pro SQL příkazy pro manipulaci s daty vrací metoda počet ovlivněných záznamů. Pro příkazy DDL (Data Definition Language) vrací 0. Příklad:
  - `stmt.executeUpdate("CREATE TABLE body (ID_bodu INTEGER);");`
  - `stmt.executeUpdate("DELETE FROM body WHERE ID_bodu = 2;");`
- `execute()` - Pokud nevíme nebo není známo, jestli SQL příkaz bude vracet relaci nebo počet ovlivněných záznamů, můžeme použít tuto metodu. Bude-li výsledkem relace, vrátí metoda hodnotu `true` a relaci je možné získat příkazem:

- `ResultSet resultSet = stmt.getResultSet();`

Když bude výsledkem počet ovlivněných záznamů, vrátí metoda `false` a počet ovlivněných záznamů bude k dispozici v proměnné `updatesCount`:

- `int updatesCount = stmt.getUpdatesCount();`

- `addBatch()` a `executeBatch()` - Tyto dvě metody jsou určeny pro dávkové zpracování. Příklad:
  - `stmt.addBatch("DELETE FROM body WHERE ID_bodu = 2;");`
  - `stmt.addBatch("DELETE FROM body WHERE ID_bodu = 3;");`
  - `stmt.addBatch("DELETE FROM body WHERE ID_bodu = 12;");`
  - `int[] results = stmt.executeBatch();`

Třídy Connection, Statement i ResultSet obsahují také metodu close(), která uzavře vytvořené objekty.

12.4 Výběr dat z databáze

Pomocí konstruktoru jsme tedy schopni souřadnice vytvářeného bodu do databáze uložit. Nyní se pokusíme uložené souřadnice naopak z databáze přečíst. K tomuto účelu si vytvoříme metodu `getCoordinates()`, která bude vracet pole typu double, v němž budou uloženy souřadnice bodu.

```
public double[] getCoordinates() {
        Connection conn = null;
        Statement stmt = null;
        ResultSet rset = null;
        double[] coordinates = new double[3];
        try {
            conn = db.getConnectionToMySQL();
            stmt = conn.createStatement();
            rset = stmt.executeQuery("SELECT X, Y, Z FROM body WHERE
                 ID_bodu = " + this.ID_bodu + ";");
            while (rset.next()) {
                coordinates[0] = rset.getDouble(1);
                coordinates[1] = rset.getDouble(2);
                coordinates[2] = rset.getDouble(3);
            }
        }
        catch (SQLException ex) {
            ex.printStackTrace();
        }
        finally {
            try {
                conn.close();
                stmt.close();
                rset.close();
            } catch (SQLException ex) {
                ex.printStackTrace();
            }
        }
        return coordinates;
    }
```

Obdobně jako u konstruktoru, i v této metodě je třeba nejprve navázat spojení s databází pomocí funkce `getConnectionToMySQL()` a vytvořit objekt třídy `Statement`. Vzhledem k tomu, že výsledkem SQL příkazu bude relace, použijeme metodu `executeQuery()`. SQL příkaz vybírá z databáze všechny body, které se rovnají `ID_bodu` (hodnotu `ID_bodu` jsme získali a uložili v konstruktoru). Výsledkem dotazu bude právě jeden záznam, protože `ID_bodu` je pro každý bod jedinečné. Tento výsledek uložíme do objektu třídy `ResultSet`.

K jednotlivým záznamům pak můžeme přistupovat pomocí metody `next()`, která vrací hodnotu `true` a posouvá aktuální čtecí pozici vždy o jeden záznam v relaci níže. Pokud již v relaci není žádný další záznam, vrátí metoda `next()` hodnotu `false`. Jednotlivé souřadnice získáme metodou třídy `ResultSet` `getDouble()`. Parametrem této metody je pořadí sloupce v relaci. K souřadnicím lze přistupovat i asociativně pomocí stejné metody:

```
coordinates[0] = rset.getDouble("X");
```

Metod pro získávání samotných dat uložených v databázových tabulkách má třída `ResultSet` celou řadu. Mezi nejpoužívanější patří:

- `getString()`,
- `getInt()`,
- `getDate()` a další.

12.5 Vymazání dat z databáze

Mazání záznamů probíhá podobným způsobem jako vkládání nových záznamů. Liší se pouze použitým SQL příkazem. Použijeme tedy podobně jako v konstruktoru metodu `executeUpdate()`.

```
public void destroy() {
        Connection conn = null;
        Statement stmt = null;
        int smazan = 0;
        try {
            conn = db.getConnectionToMySQL();
            stmt = conn.createStatement();
            smazan = stmt.executeUpdate("DELETE FROM body WHERE ID_bodu
                   = " + this.ID_bodu + ";");
        }
        catch (SQLException ex) {
            ex.printStackTrace();
        }
        finally {
            try {
                conn.close();
                stmt.close();
            } catch (SQLException ex) {
                ex.printStackTrace();
            }
        }
        if (smazan == 1) {
```

```
            System.out.println("Bod byl uspesne smazan");
        }
        else {
            System.out.println("Vymaz se nezdaril");
        }

    }
```

Všimněme si, že pokud se podaří záznam z databáze vymazat, vrátí metoda `executeUpdate()` hodnotu 1. Můžeme tedy na konzoli vypsat, zda se vymazání povedlo či nikoli.

12.6 Aktualizace dat v databázi

Když jsme v konstruktoru vytvářeli tabulku s body, vytvořili jsme v ní i sloupec Popis, ale doposud jsme do něj žádné údaje nevložili. Napíšeme metodu, pomocí níž budeme schopni k vytvořenému bodu přidat komentář.

```
public void addComment(String comment) {
        Connection conn = null;
        Statement stmt = null;
        int aktualizovan = 0;
        try {
            conn = db.getConnectionToMySQL();
            stmt = conn.createStatement();
            aktualizovan = stmt.executeUpdate("UPDATE body SET Popis =
               '" + comment + "' WHERE ID_bodu = " + this.ID_bodu + ";");
        }
        catch (SQLException ex) {
            ex.printStackTrace();
        }
        finally {
            try {
                conn.close();
                stmt.close();
            } catch (SQLException ex) {
                ex.printStackTrace();
            }
        }
        if (aktualizovan == 1) {
            System.out.println("Popis byl uspesne pridan");
        }
        else {
            System.out.println("Popis se nezdarilo vlozit");
        }

    }
```

Jak vidíme, metoda pro přidání popisu funguje na stejném principu jako předchozí metoda. Jediná odlišnost je v použitém SQL příkazu.

12.7 Další příklady

Povedlo se nám vytvořit jednoduchou aplikaci, pomocí níž můžeme vytvářet body pouhým definováním jejich souřadnic. Aplikaci lze dále rozšiřovat a vylepšovat. Metodu pro přidání komentáře k bodu již máme hotovou, pokusíme se uložený komentář vytisknout na konzoli. Použijeme k tomu metodu `getComment()`.

```
public String getComment() {
        Connection conn = null;
        Statement stmt = null;
        ResultSet rset = null;
        String popis = null;
        try {
            conn = db.getConnectionToMySQL();
            stmt = conn.createStatement();
            stmt.execute("SET NAMES 'cp852'");
            rset = stmt.executeQuery("SELECT Popis FROM body WHERE
                   ID_bodu = " + this.ID_bodu + ";");
            while (rset.next()) {
                popis = rset.getString(1);
            }
        }
        catch (SQLException ex) {
            ex.printStackTrace();
        }
        finally {
            try {
                conn.close();
                stmt.close();
            } catch (SQLException ex) {
                ex.printStackTrace();
            }
        }
        return popis;
    }
```

Do databázové tabulky též ukládáme informaci o tom, kdy byl bod vytvořen. Metoda `getCreationDate()` dokáže tuto informaci z databáze přečíst a vypsat.

```
public String getCreationDate() {
        Connection conn = null;
        Statement stmt = null;
        ResultSet rset = null;
        String datum = null;
        Timestamp timeStampTemp = null;
        try {
            conn = db.getConnectionToMySQL();
            stmt = conn.createStatement();
            rset = stmt.executeQuery("SELECT Datum_zalozeni FROM body
                     WHERE ID_bodu = " + this.ID_bodu + ";");
            while (rset.next()) {
                timeStampTemp = rset.getTimestamp(1);
```

```
            }
        }
        catch (SQLException ex) {
            ex.printStackTrace();
        }
        finally {
            try {
                conn.close();
                stmt.close();
                rset.close();
            } catch (SQLException ex) {
                ex.printStackTrace();
            }
        }

        return timeStampTemp.toString();
    }
```

12.8 Závěr

Na závěr si na jednoduchém příkladu ukážeme, jak by mohla vypadat práce s vytvořenou třídou `Point`. Komunikace s databází probíhá na pozadí, uživatel se o ni nemusí starat.

```
Point bod2 = new Point(142.5, 1217.45, 1274.45);
double[] souradnice = bod2.getCoordinates();
for (int i = 0; i < souradnice.length; i++) {
    System.out.print(souradnice[i]);
    System.out.print("  ");
}
System.out.println();
System.out.println(bod2.getCreationDate());
bod2.addComment("Nějaký komentář");
System.out.println(bod2.getComment());
bod2.destroy();
```

Na jednoduché konzolové aplikace jsme si tedy ukázali, že práce s databázovým systémem je v Javě jednoduchá a srozumitelná. Další informace o Java ve spojení s databázovými systémy jsou dostupné například na webových stránkách firmy Sun (http://java.sun.com/products/jdbc/learning.html).

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"
CZECH SPACE OFFICE

Index

3D obdélník, 148

Abstraktní třídy, 66
Aktualizace dat v databázi, 178
Anonymní vnitřní třídy, 85
aplet, 161
Applet viewer, 167
Aritmetické operátory, 26
ArrayList, 90
AWT, 95

background, 113
Bajtový kód, 10
Barvy, 112
Blok, 31
BorderLayout, 118
break, 40

Celočíselné datové typy, 20
CLASSPATH, 16
content pane, 104
continue, 40
cyklus, 37
cyklus appletu, 161

Délka pole, 32
Dědičnost, 61
Deklarace metody, 43
Detekcezměn v tabulce, 138
Dialogová okna, 138
do while, 39
dokumentační komentáře, 16
Double buffering, 147

Elipsa, 148
Eliptická výseč, 148
Error, 70
Exception, 70
Explicitní konstruktor, 53
Explicitní konverze, 24

finální metody, 65
finální třídy, 65
FlowLayout, 117

FocusLost, 127
Fonty, 114
for, 38
foreground, 113
Formální parametry, 43
formátovaný výstup, 29
formátovaný vstup, 30
Formuláře, 99

Garbage collector, 11
gettery, 57
Grafika, 146
GridLayout, 118

Handler, 108

if-else, 34
Implementace rozhraní, 78
Implementace více rozhraní, 81
Implicitní konstruktor, 53
Implicitní konverze, 24
informační komponenty, 98
Inicializace pole, 32
Inkrementace a dekrementace, 27
Interaktivní komponenty, 98

JAR, 165
Java archívy, 165
Javadoc, 17
JCheckBox, 121
JComboBox, 128
JDBC, 170
JDK, 9
Jednoduchý příkaz, 31
JFileChooser, 142
JLabel, 120
JList, 129
JPanel, 144
JRadioButton, 123
JRE, 9
JTable, 134
JTextField, 125
JVM, 10

Kombinování výjimek, 74
Komunikace s databázovým serverem, 171
konstanta řetězcová, 22
Konstruktor, 52
Kontejnerové komponenty, 97
Kontejnery, 88
Konverze základních datových typů, 25
Kopírování pole, 33
kvalifikace třídy, 18

Layout managery, 105
Linie, 148
List, 89
Listener, 107
Logický datový typ, 22
Lokální proměnné, 45
look and feel, 101

Map, 92
Matematické metody, 28
metodoa chráněných bloků, 72
Metody, 42, 43
Metody třídy, 56

Návěští, 36
Návratový typ, 44
Načtení rastru z lokálního zdroje, 155
Načtení rastru ze vzdáleného zdroje, 156
Nastavení souborové masky, 143
NEGATIVE INFINITY, 23

ošetření události, 108
Obdélník, 148
objekty jako parametry, 58
Odkaz this, 54
odvozená třída, 62
Operátor ?, 35

Předávání hodnotou, 43
Předávání odkazem, 44
předefinování metody, 64
překreslení komponenty, 147
přetížení metody, 64
Přetěžování metod, 47
Přetypování, 24
Přetypování předka a potomka, 67
Přiřazovací příkaz, 24
packages, 18
PATH, 15
podmínka úplná, 34
podmínka neúplná, 34
Pole, 31
Pole objektů, 57
poloha komponenty, 114
Polygon, 148
Polyline, 148
Polymorfismus, 67
POSITIVE INFINITY, 23
primitivní datové typy, 11
Proměnná rozhraní, 80
Proměnné třídy, 55
propagace výjimky, 71

RAD, 9
Reálné datové typy, 22
rekurze, 46
Relační operátory, 27
rodičovská třída, 61
Rozhraní a dědičnost, 82
Rušení objektů, 52
RuntimeException, 70

Scrollování seznamem, 133
Set, 91
settery, 57
ShowMessageDialog, 139
showOptionDialog, 140
Skutečné parametry, 43
Složený příkaz, 31
Smíšené konverze, 25
Souřadnicový systém komponent, 104
Spuštění appletu, 164
statické proměnné, 11
Swing, 95
switch, 36

Tělo třídy, 48
Třída, 47
tečková notace, 50
Text, 148
Throwable, 70
Top-level komponenty, 97
TreeMap, 93
TreeSet, 92

Událostmi řízené programování, 96

Vícerozměrné pole, 33
Výběr dat z databáze, 176
velikost komponenty, 114
Vkládání dat do databáze, 173
vlastní výjimky, 76
Vnitřní třída, 82
Vnitřní třída a rozhraní, 84
Vymazání dat z databáze, 177
Vytvoření objektu, 50

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"

Vyvolání výjimky, 74

while, 37

Získávání metadat o databázi, 172
Základní komponenty, 97
Zaměření komponenty, 101
Znakový typ, 21
Zobrazení rastru, 156

TENTO PROJEKT JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY
„ESF rovné příležitosti pro všechny"

Literatura

[1] Eckel B. : Myslíme v jazyce Java, Grada Publishing, 2000

[2] Herout P.: Java, grafické uživatelské prostředí a čeština, Kopp, 2001

[3] Herout P.: Učebnice jazyka Java, Java, Kopp, 2001

[4] Chapman S. J.: Začínáme programovat v jazyce Java, Computer Press, 2001

[5] Pecinovský R.: Myslíme objektově v jazyku Java, Grada publishing, 2006

[6] Virius M.: Java pro zelenáče, Neocortex, 2001

[7] Creating a GUI with JFC/Swing, http://java.sun.com/docs/books/tutorial/uiswing/index.html