

Přednášky z JXT

Přednášky z JXT

1. XML, parsování, Arrays, řazení	1
1.1. Základní charakteristiky	1
1.1.1. Co XML není	1
1.1.2. Ověření správnosti	2
1.1.3. Vývoj XML	2
1.1.4. Dvě hlavní oblasti použití	2
1.2. Syntaxe a prvky XML	3
1.2.1. Názvy značek v XML	4
1.2.2. Obecně platná pravidla pro značky	4
1.2.3. Atributy	5
1.2.4. Kdy použít elementy a kdy atributy	6
1.2.5. Entitní reference	7
1.2.6. Sekce CDATA	7
1.2.7. Komentáře	8
1.2.8. Zpracovávací instrukce	8
1.2.9. Deklarace XML a použitý charset	8
1.3. Ukázka výhod datově orientovaného XML dokumentu	9
1.3.1. Binární soubor v proprietárním formátu	9
1.3.2. Textový soubor	9
1.3.3. XML dokument	9
1.3.4. Jiné způsoby zápisu XML dokumentu	10
1.4. Kontrola XML dokumentu	13
1.5. Jmenné prostory (XML namespaces)	14
1.6. Parsování řetězců	16
1.6.1. Základní použití split()	17
1.6.2. Použití regulárních výrazů	18
1.7. Podpora práce s poli – třída Arrays	20
1.7.1. Možnosti třídy Arrays	21
1.8. Řazení objektů	22
1.8.1. Přirozené řazení (<i>natural ordering</i>)	23
1.8.2. Absolutní řazení (<i>total ordering</i>)	25
2. Kolekce a genericita (1)	28
2.1. Úvodní informace	28
2.2. Použití wildcard – unbounded wildcard	31
2.3. Využití polymorfismu – bounded wildcard	32
2.4. Rozhraní Collection	33
2.5. Rozhraní List	34
2.6. ArrayList	34
2.7. Třída Collections	39
3. Kolekce a genericita (2)	43
3.1. Postupný průchod kolekcí	43
3.1.1. For-Each	43
3.1.2. Iterátory	44
3.2. Automatické zapouzdřování primitivních datových typů	46
3.3. Výhodnost jednotlivých seznamů	46
3.4. Ochrana proti nekonzistenci dat	47
3.5. Množiny – rozhraní Set	48
3.5.1. Práce s vlastní třídou v množině	50
3.5.2. Použití Collections	52
3.5.3. Rozhraní SortedSet	52
3.5.4. Množinové operace a triky	53
3.6. Mapy – rozhraní Map	54
3.6.1. Třída TreeMap	56

3.7. Problémy objektů v hešovacích třídách	58
3.7.1. Metoda equals()	58
3.7.2. Metoda hashCode()	60
3.7.3. Efektivní hashCode()	60
3.8. Ideální třída pro vkládání do libovolné kolekce	65
4. Java a XML, SAX	70
4.1. Java a XML	70
4.1.1. Obecné vlastnosti parseru	70
4.1.2. Rozhraní parserů pro Javu	72
4.1.3. Základní dělení parserů	73
4.1.4. Přehled parserů	74
4.2. SAX – <i>Simple API for XML</i>	74
4.2.1. Úvodní informace	74
4.2.2. Základní postup při zpracování	74
4.2.3. Zpracování parsovaného XML dokumentu	77
4.2.4. Zpracování složitějšího XML dokumentu	83
4.2.5. Problematika různého kódování	84
4.2.6. Nastavení vlastností parseru	84
4.2.7. Validace oproti DTD nebo XSD	85
4.2.8. Práce se jmennými prostory	88
5. DOM – <i>Document Object Model</i>	91
5.1. Základní informace	91
5.2. Základní použití DOM	93
5.3. Zpracování parsovaného XML dokumentu	95
5.3.1. Výpočet celkové váhy	95
5.4. Metody rozhraní Node	96
5.4.1. Metody pro získání informace	96
5.4.2. Metoda pro pohyb nahoru (na rodiče)	97
5.4.3. Metody pro horizontální pohyb (na sourozence)	97
5.4.4. Metody pro pohyb dolů (na potomky)	97
5.4.5. Metody pro práci s atributy	97
5.5. Výpočet celkové ceny	98
5.6. Problém vkládaných elementů (odřádkování)	99
5.7. Automatické odstranění komentářů	101
5.8. Všechny objekty v paměti	102
5.9. Průchod stromem dokumentu	104
5.10. Snadné odstranění „odřádkovacích“ nodů	108
5.11. Zápis dokumentu	108
5.11.1. Ovlivnění práce transformační třídy	109
5.11.2. Ukázka zápisu do souboru	110
5.11.3. Problematika odřádkování a odsazování	112
5.12. Modifikace dokumentu	113
5.12.1. Změna hodnoty již existujícího elementu či atributu	113
5.12.2. Odstranění nodu	114
5.12.3. Vkládání nových nodů	114
5.12.4. Změna XML dokumentu a jeho zápis	115
5.13. Vytváření nového dokumentu	116
5.13.1. Klonování nodů	116
5.14. Validace nově vytvořeného nebo měněného dokumentu	118
6. Grafické uživatelské rozhraní – GUI	122
6.1. Základní informace	122
6.2. Zobrazení základního okna a přidání komponent	123
6.2.1. Používaný způsob zobrazení základního okna	124

6.2.2. Používaný způsob vložení komponent	124
6.3. Princip reakce na události	125
6.3.1. Použití vnitřních tříd	126
6.3.2. Použití anonymních vnitřních tříd	127
6.3.3. Použití anonymních vnitřních tříd a privátních metod	128
6.3.4. Rozhraní posluchače má více metod	129
6.4. Jak komponentu popíšeme	131
6.4.1. Trocha typografické teorie	131
6.4.2. Fonty v Javě	132
6.4.3. Popis komponenty zvoleným fontem	133
6.4.4. Využití HTML značkování	133
6.5. Zpřístupnění komponenty	134
6.6. Viditelnost komponenty	134
7. GUI – pokračování	136
7.1. Rozmísťování komponent	136
7.1.1. FlowLayout	136
7.1.2. GridLayout	137
7.1.3. BorderLayout	137
7.1.4. GridBagLayout	138
7.2. Komunikace pomocí Observable-Observer	139
7.3. Použití ikon	142
7.3.1. Základní informace	142
7.3.2. Problémy při natažení obrázku	143
7.3.3. Seskupení ikon do nástrojové lišty (toolbar)	144
7.3.4. Aktivita ikon a souvisejících položek v menu	146
8. Grafika	150
8.1. Jak se komponenty překreslují	150
8.2. Grafické možnosti API	151
8.2.1. Kam kreslit a jak	151
8.2.2. Souřadnicový systém	151
8.2.3. Práce s grafickým kontextem	152
8.3. Základní grafická primitiva	153
8.3.1. Porovnání starého a nového způsobu kreslení	154
8.4. Nastavení parametrů barvy a čáry	155
8.4.1. Rozhraní Paint	155
8.4.2. Rozhraní Stroke	156
8.5. Křivky jako grafická primitiva	161
8.5.1. Obecná křivka – GeneralPath	161
8.6. Afinní transformace	162
8.7. Interakce s uživatelem	163
8.8. Interakce uživatele s jednotlivými grafickými objekty	165
8.9. Práce s textem a fonty	167
8.9.1. Metrika fontu	168
8.9.2. Použití fyzických fontů	169
9. Schémové jazyky DTD a XSD	173
9.1. Význam schémových jazyků (schémat)	173
9.2. DTD – Document Type Definition	173
9.2.1. Výhody a nevýhody DTD	173
9.2.2. Spojení DTD a XML	174
9.2.3. Prvky a struktura DTD	174
9.3. W3C XML Schema – WXS nebo XSD	185
9.3.1. Základní informace	185
9.3.2. Praktické použití XSD	186

9.3.3. Začátek XSD souboru	186
9.3.4. Výběr běžně použitelných základních typů	190
9.3.5. Jaké možnosti nám dávají základní typy	190
9.3.6. Možnosti restrikcí (<i>constraining facets</i>)	192
9.3.7. Složené (komplexní) datové typy	195
9.3.8. Atributy	198
9.3.9. Atribut je součástí koncového elementu	199
9.3.10. Prázdný element s atributy	199
9.3.11. Závěrečná definice kořenového elementu	200
9.3.12. Připojení schématu k dokumentu	200
9.3.13. Validace pomocí xerces	201
9.3.14. Jmenné prostory	201
9.3.15. Použití prázdné hodnoty	204
9.3.16. Práce s okrajovými bílými znaky	205
10. JWDSP, StAX, Ant	208
10.1. JWDSP	208
10.2. Sun Java Streaming XML Parser (SJSXP)	208
10.2.1. Základní postup při zpracování	209
10.2.2. Přehled základních možností čtení	210
10.2.3. Zpracování atributů	211
10.2.4. Čtení na žádost	212
10.2.5. Zápis do XML dokumentu	215
10.3. Ant – Another Neat Tool	217
10.3.1. Základní informace	217
10.3.2. Jak Ant získat	218
10.3.3. Použití	218
10.3.4. Použití <property>	221
10.3.5. Nastavování cest a opětovné použití nastavení	224
10.3.6. Nastavení jmen souborů	225
10.3.7. Kombinování <path> a <fileset>	226
10.3.8. Možnosti <target>	226
10.3.9. Když je něco špatně	230
10.3.10. Úkoly (tasks)	230
10.3.11. Přehled a použití často používaných úkolů	232
10.3.12. XJC pro JDK 1.6	239
10.3.13. Ukázka komplexního projektu použitelného v praxi	240
10.3.14. Doporučeno k přečtení:	242
11. Java Architecture for XML Binding – JAXB	243
11.1. Základní informace	243
11.1.1. Podpora z Ant	244
11.2. Generování souborů z XSD	247
11.2.1. Princip bindingu	248
11.3. Čtení XML dokumentu	249
11.3.1. Výpočet celkové váhy	250
11.4. Čtení dokumentu včetně zpracování atributů	251
11.4.1. Výpočet celkové ceny	251
11.4.2. Všechny objekty v paměti	252
11.5. Změna hodnot a vytvoření nových elementů	252
11.6. Zápis do XML dokumentu	253
11.7. Validace	255
11.7.1. Ukázka dvojí validace	256
11.8. Příprava kompletně nového dokumentu	258
12. Java a národní prostředí	260

12.1. Kódování	260
12.1.1. Podporovaná kódování	261
12.2. Čeština v programu	261
12.2.1. Zdrojový kód není v implicitním kódování	262
12.3. České výpisy na konzoli	264
12.3.1. Vstup češtiny z konzole	266
12.4. Čeština v souborech	266
12.5. Převody mezi různými kódováními uvnitř programu	268
12.6. Třída Locale	269
12.7. Tisk dle národních zvyklostí	271
12.7.1. Základní principy <code>format()</code>	271
12.7.2. Formátování dle lokality	273
12.7.3. Formátování datumu a času	276
12.7.4. Řazení řetězců	278
12.7.5. Označování začátků a konců slov	281
13. Tabulky	283
13.1. Základní informace	283
13.2. Základní princip	283
13.2.1. Primitivní použití	284
13.3. Vylepšování funkčnosti	286
13.3.1. Rolování řádek a sloupců	286
13.3.2. Zobrazení záhlaví	287
13.4. Uživatelsky konfigurovatelné zobrazování hodnot	288
13.4.1. Datová vrstva informuje prezentační vrstvu o typech dat	288
13.4.2. Využití vlastního zobrazovače	289
13.5. Práce s jednotlivými sloupci na úrovni prezentační vrstvy	292
13.6. Práce se záhlavím	293
13.7. Změna hodnot v tabulce	294
13.7.1. Editace základních datových typů	294
13.7.2. Použití editoru na bázi <code>DefaultCellEditor</code>	296
13.7.3. Vytvoření vlastního editoru na bázi libovolné komponenty	298
13.8. Řazení v tabulce	300
13.9. Data jsou organizována jako na sobě nezávislá pole	301
13.9.1. Ukázka vzájemného provázání dat	302
13.9.2. Ukázka načítání dat ze souboru	306
13.9.3. Data jsou součástí třídy implementující <code>TableModel</code>	308

Kapitola 1. XML, parsování, Arrays, řazení

1.1. Základní charakteristiky

- rozšiřitelný značkovací jazyk (*eXtensible Markup Language*)
- pochází z oblasti zaměřené na uchování a zpracování textových dokumentů
 - jeho „otec“ je SGML (*Standard Generalized Markup Language*)
 - jeho „bratr“ je HTML (*Hypertext Markup Language*)
- standard W3C
 - velmi rozšířen
- jednoduchý otevřený formát - lze libovolně doplňovat
- popisuje data nezávisle na platformě
 - „Java poskytuje přenositelný kód, XML přenositelná data“
- volná množina značek - rozdíl s HTML - s pevnou gramatikou
 - lze programově kontrolovat správnost struktury (a částečně i obsahu)
- principiálně textový soubor (textová informace) - textové jsou značky i data
- značkování jasně popisuje účel (rozdíl od proprietárních binárních formátů)
 - značky neurčují vzhled dat, ale jejich strukturu
 - ♦ HTML určuje jak se data zobrazí
 - ♦ XML určuje, jaký mají data význam
- na velmi jednoduchou myšlenku je navázáno velké množství technologií - viz dále
- XML je jeden z nejdůležitějších formátů výměny dat strukturovaným způsobem
- jeden konkrétní příklad

```

<lekar>
  <jmeno>
    <prijmeni>Petr</prijmeni>
    <krestni>Pavel</krestni>
  </jmeno>
  <telefon>377 123 456</telefon>
</lekar>

```

1.1.1. Co XML není

- všespásitelná technologie

- programovací jazyk - nelze přeložit do spustitelného programu
- síťový protokol (asociace HTML versus HTTP)
- databáze, ačkoliv s databázemi může spolupracovat
- není vhodný pro rozsáhlé bitové sekvence - obrázky, zvuky, video
- málo vhodný pro značně (stovky MB) rozsáhlá data (značky zvyšují velikost souboru)

1.1.2. Ověření správnosti

- díky pevné struktuře lze nezávisle na budoucí aplikaci ověřovat správnost dat
 - velká výhoda - kontrola dat se přesouvá na jejich pořizovatele
 - vstupní data se zkontrolují před zasláním do aplikace
 - aplikace může mít pak mnohem jednodušší vstup - nemusí kontrolovat chybové stavy
- několik zvyšujících se úrovní ověřování správnosti
 1. jen XML - **správně strukturovaný** (*well-formed*) dokument
 - značky se nekříží, pouze jedna je kořenová, atd. (asi 7 jednoduchých pravidel - podrobně viz dále)
 2. XML a DTD - **validní dokument**
 - je použita jasně definovaná množina značek, ve správném pořadí
 3. XML schémata (XSD)
 - jako u DTD + data mají správné typy a částečně kontrolovatelné hodnoty

1.1.3. Vývoj XML

- SGML (*Standard Generalized Markup Language*)
 - značkovací jazyk pro textové dokumenty (armáda USA, letectví)
 - ♦ správa technické dokumentace rozsahů desítek tisíc stran
 - extrémně složitý
- nejúspěšnější aplikací SGML je HTML
- vývoj XML od 1996 do 1998 verze 1.0
 - v současné době verze 1.1

1.1.4. Dvě hlavní oblasti použití

- ve skutečnosti je jich mnohem více - použití XML je rychle se rozšiřující oblast
1. **dokumenty orientované na sdělení**

- knihy, WWW stránky, dokumentace - DocBook
- navazující technologie např. XSLT, XSLT-FO
- pořídí se (jednou) označovaný text a pak se automaticky transformuje do mnoha výstupních (čitelných) formátů, např. HTML, PS, PDF, RTF, plain textu

2. datově orientované dokumenty

- B2B (*bussines to bussines*) aplikace - strukturovaná data pro výměnu informací mezi aplikacemi
 - textový soubor odstiňuje:
 - ♦ rozsah čísel (int je dvoubajtové)
 - ♦ typ čísel (znaménkové int je v doplňkovém kódu)
 - ♦ způsob uložení čísel *big-* nebo *little-endian*
- konfigurační soubory a mnohé další

1.2. Syntaxe a prvky XML

- dokument XML se skládá z textového obsahu označovaného pomocí **textových značek (tagů)**
- používá se slovo „dokument“, protože to nemusí být nutně soubor (XML dokument po síti)
- značky si lze libovolně dodávat - otevřený formát - rozdíl od HTML
- značky popisují obsah nikoliv formátování
- na značky jsou mnohem přísnější pravidla než v HTML
 - dokument se hůře píše
 - dá se ale snadno správně načíst (zkontrolovat), protože je parser jednodušší
 - ♦ asi 50% kódu prohlížečů HTML řeší chybové situace, tj. problémy v datech
- **element** = počáteční a koncová značka a informace mezi nimi
- musí existovat jeden element, který je **kořenový** (vše obaluje)
- nejjednodušší XML dokument, který ale nemá žádný smysl

```
<a/>
```

- má jeden element typu `a`, který nemá obsah

- nejjednodušší XML dokument

```
<pozdrav>ahoj</pozdrav>
```

- má jeden element typu `pozdrav`, jehož obsah je `ahoj` typu „znaková data“

1.2.1. Názvy značek v XML

- obecně lze použít akcentovaná i neakcentovaná písmena, číslice a znaky „-“ (pomlčka) „_“ (podtržítka) a „.“ (tečka)

- nesmí začínat číslicí

- rozlišuje se velikost písmen <POZDRAV>, <Pozdrav>, <pozdrav> - různé

- měly by být významové - <z1>, <z12>, <z3> jsou sice správně, ale nic neříkají o obsahu

1.2.1.1. Praktické doporučení pro datově orientované dokumenty

- s ohledem na budoucí možnost automatizovaného generování programů (*data binding*) je výhodné

- vytvářet názvy značek pomocí stejných pravidel jako identifikátory v Javě, např.:

```
pozdrav, uvitaciPozdrav, pozdravNaRozloucenou
```

- v identifikátorech nepoužívat

- ♦ akcenty - teoreticky by ale neměly dělat problémy

- ♦ pomlčky a tečky - určitě budou dělat problémy

1.2.2. Obecně platná pravidla pro značky

- každá značka musí mít svoji **uzavírací značku**

- platí i pro **prázdnou značku**

```
<bezPozdravu></bezPozdravu>
```

- ♦ lze zkrátit na

```
- <bezPozdravu/>
```

```
- <bezPozdravu />
```

- dokumenty mají vždy strukturu stromu

- jen jeden element je **kořenový (element dokumentu)**

- všechny další elementy jsou elementy potomků

- mezery ve formátování jsou ignorovány

```
<lekar>
  <jmeno>
    <prijmeni>Petr</prijmeni>
    <krestni>Pavel</krestni>
  </jmeno>
  <telefon>377 123 456</telefon>
</lekar>
```

- každý potomek má nejvýše jednoho rodiče - nelze křížit značky - chyba:

```
<jmeno>
  <prijmeni>Petr</prijmeni>
  <krestni>Pavel</jmeno>
</krestni>
```

- elementy obsahují:

- jiné elementy, např.:

- ♦ <jmeno> obsahuje dva elementy a to <prijmeni> a <krestni>

- data, např.:

- ♦ <prijmeni> obsahuje řetězec (tj. data) Petr

- je možný element s kombinovaným (smíšeným) obsahem

```
<telefon>
  <predvolba> +420 </predvolba> 377 123 456
</telefon>
```

- kterému se ale snažíme vyhnout

1.2.3. Atributy

- dvojice *název-hodnota* umístěná do počáteční značky

- hodnoty musejí být zavřeny do uvozovek (pozor " nikoliv „ nebo “ nebo ”)

```
<vaha jednotka="kg">150</vaha>
```

- název atributu je „jednotka“

- hodnota atributu je „kg“

- je-li v hodnotě znak uvozovek, lze hodnotu uzavřít do apostrofů

```
<znak vzhled=''' popis="uvozovky" />
```

- atributů může mít značka víc a na jejich pořadí nezáleží

```
<vaha jednotka="kg" datumVazeni="2004-12-03"> 150 </vaha>
```

- atributy představují kompaktnější zápis

- často se snadněji zpracovávají podpůrnými technologiemi

- někdy dilema, zda použít atribut nebo obsah elementu či **vnořený element**

- tři víceméně stejné možnosti

1. <vaha jednotka="kg" hodnota="150"/>

2. `<vaha jednotka="kg">150</vaha>`

3. `<vaha>
 <jednotka>kg</jednotka>
 <hodnota>150</hodnota>
</vaha>`

- nebo (nejméně vhodně) **element se smíšeným obsahem**

```
<vaha>  
  <jednotka>kg</jednotka>  
  150  
</vaha>
```

- atributy různých elementů mohou mít stejná jména

```
<vaha jednotka="kg">45,5</vaha>
```

```
<vyska jednotka="cm">170</vyska>
```

- atribut lze vždy nahradit vnořeným elementem

1.2.4. Kdy použít elementy a kdy atributy

1.2.4.1. Atributy

- dopředu je jasná doména hodnot

- datумы a časy
- číselné údaje s omezeným rozsahem (vek="1" až "150")
- výčty čísel (dph="22" nebo "19" nebo "5")
- výčty textů (jednotka="kg" nebo "g" nebo "libra")

- informaci už nelze dále strukturovat

- musí existovat pevně daný (tj. nebude se měnit) vztah k elementu

- potřeba kompaktního zápisu - hodnota atributu je uzavřena v uvozovkách, není potřeba ji uzavírat koncovou značkou

- je nevhodné umístit do atributu obecný text

```
<citát text="Byli jsme a budem!">
```

- u dokumentů orientovaných na sdělení by měl atribut představovat pouze doplňkovou informaci

```
<citát jazyk="čeština">  
  Byli jsme a budem!  
</citát>  
<citát jazyk="angličtina">
```

```
To be or not to be?  
<poznámkaPodCarou cislo="10">  
  Shakespeare.  
</poznámkaPodCarou>  
</citát>
```

Poznámka

Prakticky se pro označení jazyka používá **jmenný prostor** (viz dále)

```
<citát xml:lang="cs">Byli jsme a budem!</citát>
```

```
<citát xml:lang="en">To be or not to be?</citát>
```

1.2.4.2. Elementy

- opakuje-li se vícekrát - atributy musejí mít rozdílná jména

```
<vahoveZmeny periodaVazeni="7 dnů">  
  <hodnota>150</hodnota>  
  <hodnota>140</hodnota>  
  <hodnota>130</hodnota>  
</vahoveZmeny>
```

- všechny další případy, které nejsou vyjmenovány u atributů

1.2.5. Entitní reference

- náhrada problematických znaků znakového obsahu elementu

- chybně `<nerovnost>3 < 5</nerovnost>`
- dobře `<nerovnost>3 < 5</nerovnost>`

- předdefinované entity

<	<
&	&
>	>
"	"
'	'

1.2.6. Sekce CDATA

- pro výpis textu „tak, jak je“ - většinou u dokumentů orientovaných na sdělení

```
<nerovnost><![CDATA[ 3 < 5 < 7 < 9]]></nerovnost>
```

- nechává všem znakům původní význam

- použití pro výpisy programů, příkazy vnořených jazyků apod.

```
<usekProgramu jazyk="Java">
  <![CDATA[
    for (int i = 0; i < 5; i++) {
      System.out.format("%d < 5\n", i);
    }
  ]]>
</usekProgramu>
```

1.2.7. Komentáře

```
<!-- komenář -->
```

- nesmí se vnořovat a nesmí být součástí značky
- chyba <vaha <!-- spisovně hmotnost --> >

1.2.8. Zpracovávací instrukce

- uzavřeny do značky <? ?>
- příklad nejčastější instrukce, která je v naprosté většině případů jako první řádka XML dokumentu

```
<?xml version="1.0" encoding="utf-8"?>
```

1.2.9. Deklarace XML a použitý charset

- de facto povinný začátek každého XML dokumentu
- pro určení charsetu zásadně používat kanonická jména - viz dříve
- XML implicitně používá znakovou sadu Unicode
- bez určení charsetu v atributu `encoding` je dokument implicitně v UTF-8

```
<?xml version="1.0" ?>
```

- nezávisle na použitém kódování lze do dokumentu vložit jakýkoliv znak
- je třeba znát jeho Unicode *code point* (www.unicode.org/charts)
- např. Ω je:
 - ♦ Ω (hexadecimálně)
 - ♦ Ω (dekadicky - nepoužívat - mate)

1.3. Ukázka výhod datově orientovaného XML dokumentu

- použití pro předávání dat mezi aplikacemi
- možnosti:
 1. binární soubor v proprietárním formátu
 2. textový soubor
 3. XML dokument

1.3.1. Binární soubor v proprietárním formátu

- nejhorší možnost, problémy s:
 - délkou řetězců
 - kódováním češtiny
 - datovými typy čísel a jejich rozsahem
 - organizací dat, atd.

1.3.2. Textový soubor

- odpadají problémy s délkou řetězců a datovými typy čísel
- zůstávají problémy kódování češtiny a hlavně význam jednotlivých dat
- lze použít jen pro jeden typ záznamu a navíc s neměnnou strukturou

```
Petr;Pavel;377 123 456
Vanda;Alexandra
mu+;+át+;hl+;ji+ö+;150;150
+ena;Ot+lie;Otyl+i;45,5;170
```

Poznámka

Data jsou záměrně v neznámém charsetu, který neumí editor zobrazit.

1.3.3. XML dokument

- je jednoznačně určen charset (kódování češtiny)
- je zcela zřejmý význam jednotlivých dat
- příklad obsahující stejná data jako předcházející textový soubor

```
<?xml version="1.0" encoding="UTF-8"?>
<obezitologie>
```



```

<personal>
  <lekar>
    <jmeno>
      <prijmeni>Petr</prijmeni>
      <krestni>Pavel</krestni>
    </jmeno>
    <telefon>377 123 456</telefon>
  </lekar>
  <sestra>
    <jmeno>
      <krestni>Vanda</krestni>
      <prijmeni>Alexandra</prijmeni>
    </jmeno>
  </sestra>
</personal>
<leceneOsoby>
  <nadvaha>
    <pacient pohlavi="muž">
      <jmeno>
        <prijmeni>Štíhlý</prijmeni>
        <krestni>Jiří</krestni>
      </jmeno>
      <vyska jednotka="cm">150</vyska>
      <vaha jednotka="kg">150</vaha>
    </pacient>
  </nadvaha>
  <podvyziva>
    <pacient pohlavi="žena">
      <jmeno>
        <krestni>Otýlie</krestni>
        <prijmeni>Otlá</prijmeni>
      </jmeno>
      <vaha>45,5</vaha>
      <vyska jednotka="cm">170</vyska>
    </pacient>
  </podvyziva>
</leceneOsoby>
</obezitologie>

```

1.3.4. Jiné způsoby zápisu XML dokumentu

- předchozí dokument splňuje všechny podmínky XML (je *well-formed*) - lze jej **validovat**
- ale z hlediska dalšího zpracování (pohlížíme na něj jako na datově orientovaný dokument) není dobrý
 - má příliš „stupňů volnosti“, např.:
 - ♦ <prijmeni> a <krestni> lze prohodit
 - ♦ <telefon> u <sestra> je nepovinný
 - ♦ atribut jednotka elementu <vaha> je nepovinný

♦ elementy <vyska> a <vaha> lze prohodit

- následují ukázky dvou různých přístupů lépe strukturovaných dokumentů

1.3.4.1. Přísně hierarchické schéma

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<obezitologie>
  <personal>
    <lekar>
      <jmeno>
        <krestni>Pavel</krestni>
        <prijmeni>Petr</prijmeni>
      </jmeno>
      <telefon>377 123 456</telefon>
    </lekar>
    <sestra>
      <jmeno>
        <krestni>Vanda</krestni>
        <prijmeni>Alexandra</prijmeni>
      </jmeno>
      <telefon>377 123 789</telefon>
    </sestra>
  </personal>

  <leceneOsoby>
    <nadvaha>
      <pacient pohlavi="muž">
        <jmeno>
          <krestni>Jiří</krestni>
          <prijmeni>Štíhlý</prijmeni>
        </jmeno>
        <vaha jednotka="kg">150</vaha>
        <vyska jednotka="cm">150</vyska>
      </pacient>
    </nadvaha>

    <podvyziva>
      <pacient pohlavi="žena">
        <jmeno>
          <krestni>Otýlie</krestni>
          <prijmeni>Otlá</prijmeni>
        </jmeno>
        <vaha jednotka="kg">45,5</vaha>
        <vyska jednotka="cm">170</vyska>
      </pacient>
    </podvyziva>
  </leceneOsoby>
</obezitologie>

```

1.3.4.2. Ploché schéma

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<obezitologie>
  <osoba role="lékař">
    <jmeno>
      <prijmeni>Petr</prijmeni>
      <krestni>Pavel</krestni>
    </jmeno>
    <telefon>377 123 456</telefon>
  </osoba>

  <osoba role="sestra">
    <jmeno>
      <krestni>Vanda</krestni>
      <prijmeni>Alexandra</prijmeni>
    </jmeno>
  </osoba>

  <osoba role="pacient - nadváha">
    <jmeno>
      <prijmeni>Štíhlý</prijmeni>
      <krestni>Jiří</krestni>
    </jmeno>
    <pohlavi hodnota="muž"/>
    <vyska jednotka="cm">150</vyska>
    <vaha jednotka="kg">150</vaha>
  </osoba>

  <osoba role="pacient - podvýživa">
    <jmeno>
      <krestni>Otýlie</krestni>
      <prijmeni>Otlá</prijmeni>
    </jmeno>
    <pohlavi/>
    <vaha>45,5</vaha>
    <vyska jednotka="cm">170</vyska>
  </osoba>
</obezitologie>
```

1.3.4.3. Porovnání obou způsobů

- nelze obecně říci, který přístup je lepší

- pokud lze očekávat v budoucím dokumentu množství organizačních změn, je lepší ploché schéma

```
<osoba role="primář"> <osoba role="vrchní sestra">
```

- ♦ nové role se přidávají snadno bez nutnosti zásahu do stávající struktury

- je-li struktura víceméně neměnná, je lepší využít co nejvíce hierarchické schéma, tj. dodat co možná nejvíce informací

- z dlouhodobých zkušeností víme, že se data mění méně (pomaleji) než aplikace:

- čím více informace dokument obsahuje, tím lépe
- nevadí, když se v jednotlivých aplikacích všechny informace pokaždé nevyužijí

1.4. Kontrola XML dokumentu

- nejnižší úroveň kontroly (validace)

- dokument je **správně strukturován** (*well-formed*)

- existuje sedm oblastí základní kontroly (všechny příklady jsou chybové):

1. musí existovat právě jeden kořenový element

```
<?xml version="1.0" encoding="UTF-8"?>
<jazykC>procedurální</jazykC>
<jazykJava>objektový</jazykJava>
```

2. každá počáteční značka má odpovídající ukončující značku

```
<?xml version="1.0" encoding="UTF-8"?>
<jazykC>procedurální
```

3. elementy se nesmějí překrývat (křížit)

```
<?xml version="1.0" encoding="UTF-8"?>
<jazykC>procedurální
  <charakteristika>univerzální
</jazykC>
</charakteristika>
```

4. hodnoty atributů musí být v uvozovkách (nebo v apostrofech)

```
<?xml version="1.0" encoding="UTF-8"?>
<jazykC vyucujici=Herout>
  procedurální
</jazykC>
```

5. element nesmí mít stejně pojmenované atributy

```
<?xml version="1.0" encoding="UTF-8"?>
<jazykC Herout="přednáší" Herout="zkouší">
  procedurální
</jazykC>
```

6. komentáře nesmí být vnořené a ani uvnitř značek

```
<?xml version="1.0" encoding="UTF-8"?>
<jazykC <!-- už dlouho --> přednáší="Herout">
```

```
    procedurální
</jazykC>
```

7. ve znakových datech nejsou znaky < a &

```
<?xml version="1.0" encoding="UTF-8"?>
<jazykC přednáší="Herout">
    operátor & vrátí adresu
</jazykC>
```

■ při průběhu kontroly se nic automaticky neopravuje!

- jednoznačnost zdrojového XML dokumentu

■ nejjednodušší kontrola - zobrazit v HTML prohlížeči, např. MSIE 6

■ prakticky jsou kontroly prováděny již hotovými parseři

■ validace pomocí parseru xerces

```
>xerces.bat soubor.xml
```

■ kde xerces.bat je soubor uložený v cestě PATH s jednořádkovým obsahem

```
@java -cp "c:\Program Files\Java\xerces\xercesImpl.jar";"c:\Program Files\Java\xerces\xmlParserAPIs.jar";"c:\Program Files\Java\xerces\xercesSamples.jar"
sax.Counter %*
```

■ pro správně formulovaný dokument vypíše:

```
>xerces obezitologie-pevne.xml
obezitologie-pevne.xml: 71 ms (27 elems, 6 attrs, 0 spaces, 358 chars)
```

1.5. Jmenné prostory (XML namespaces)

■ možnost kombinovat více sad značek v jednom dokumentu

■ každá sada značek je jednoznačně definována svojí URI adresou

- URI (*Uniform Resource Identifier*) je nadmnožina URL (*Uniform Resource Locator*)
- URL jednoznačně určuje umístění dokumentu v síti

■ proč se to dělá?

- používají se již hotové sady značek buď z DTD nebo z XSD
- dokument vytváří/doplňuje více lidí

■ pro použití elementu z určité sady značek je třeba určit identifikátor (*prefix*) této sady

- jméno prefixu je libovolné, mělo by být co nejkratší

- prefix se určuje pomocí speciálního atributu `xmlns` (XML namespaces)

```
xmlns:jmenoPrefixu="URI sady značek"
```

■ prefix se pak používá před každým názvem elementu nebo atributu

```
<?xml version="1.0" encoding="UTF-8"?>
<prj5:evidencePredmetuPRJ5
  xmlns:prj5="http://www.kiv.zcu.cz/~herout/xml/prj5-sada"
  xmlns:osobni="http://www.kiv.zcu.cz/~herout/xml/osobni-sada" >
```

```
<osobni:identifikace>
  <osobni:cislo>
    A12345
  </osobni:cislo>
  <osobni:jmeno>
    Pavel Herout
  </osobni:jmeno>
</osobni:identifikace>
```

```
<prj5:jmeno>
  Praktické poznatky z využití XML v lihovarnictví
</prj5:jmeno>
<prj5:hodnoceni prj5:bod="100"/>
</prj5:evidencePredmetuPRJ5>
```

■ URI sady značek musí být unikátní

- většinou se používá URL a odpovídající soubor nemusí existovat
- URI pouze zajišťuje jednoznačné pojmenování

■ v XML dokumentu pak lze použít dva elementy pojmenované <jmeno> rozlišené prefixem

- <osobni:jmeno>

- <prj5:jmeno>

■ je možné jeden jmenný prostor deklarovat jako implicitní

- typicky pokud jedna sada značek výrazně převažuje nad druhou
- lze pak vynechat jeho prefix

```
<?xml version="1.0" encoding="UTF-8"?>
<evidencePredmetuPRJ5
  xmlns="http://www.kiv.zcu.cz/~herout/xml/prj5-sada"
  xmlns:osobni="http://www.kiv.zcu.cz/~herout/xml/osobni-sada"
>
```

```
<osobni:identifikace>
  <osobni:cislo>
    A12345
  </osobni:cislo>
```

```
<osobni:jmeno>
  Pavel Herout
</osobni:jmeno>
</osobni:identifikace>
```

```
<jmeno>
  Praktické poznatky z využití XML v lihovarnictví
</jmeno>
<hodnoceni bodu="100"/>
</evidencePredmetuPRJ5>
```

■ pozor na to, že implicitní jmenný prostor se nevztahuje na atributy

- zde to nevádí, v budoucnu při použití XSD bude

■ jmenné prostory mají platnost pro všechny podřazené elementy

■ dále platí pravidlo, že později deklarovaný zastiňuje dříve deklarovaný

```
<?xml version="1.0" encoding="UTF-8"?>
<evidencePredmetuPRJ5
  xmlns="http://www.kiv.zcu.cz/~herout/xml/prj5-sada"
>

  <identifikace
    xmlns="http://www.kiv.zcu.cz/~herout/xml/osobni-sada"
  >

    <cislo>
      A12345
    </cislo>
    <jmeno>
      Pavel Herout
    </jmeno>
  </identifikace>

  <jmeno>
    Praktické poznatky z využití XML v lihovarnictví
  </jmeno>
  <hodnoceni bodu="100"/>
</evidencePredmetuPRJ5>
```

■ toto je prakticky využitelná možnost protože se elementy z různých sad málokdy „míchají do sebe“

■ jmenné prostory mají mnoho dalších možností - viz literatura

1.6. Parsování řetězců

■ řešíme problém, kdy ve vstupním souboru je na jedné řádce několik čísel (obecně hodnot)

- nejčastější oddělovače jsou
 - ♦ mezera nebo mezery

- ♦ středník nebo čárka – často soubory s příponou `.csv` (*Comma Separated Values*)

- ♦ dvojtečka

■ načteme celou řádku jako řetězec

- je jednodušší rozdělit řetězec na jednotlivé podřetězce (parsovat) v programu než jej postupně načítat ze souboru

■ před převodem jednotlivých číslíc je nutné načtenou řádku **parsovat**

- dříve se používala třída `java.util.StringTokenizer`
- jednodušší způsob dnes (od JDK 1.4) je použití metody `split()` třídy `String`
 - ♦ metoda rozdělí původní řetězec na jednotlivé podřetězce, které umístí do vráceného pole řetězců
 - ♦ pak se postupně převedou všechny podřetězce na čísla

1.6.1. Základní použití `split()`

- jednotlivé podřetězce jsou odděleny jen jedním unikátním dopředu známým znakem (viz nejčastější oddělovače výše)

- oddělovací znak se zapisuje jako skutečný parametr metody `split()`

Příklad 1.1. Rozdělení podřetězců oddělených mezerou

```
String radka = "123 45 6 789";
String[] podretezce = radka.split(" ");
for (int i = 0; i < podretezce.length; i++) {
    System.out.println("|" + podretezce[i] + "|");
}
```

vypíše

```
|123|
|45|
|6|
|789|
```

z výpisu je vidět, že oddělovací mezery jsou „požrány“

■ pro jiný oddělovač je situace analogická

```
String radka = "123;45;6;789";
String[] podretezce = radka.split(";");
```

- je-li v původním řetězci některý z oddělovacích znaků zdvojen, bude výsledkem prázdný řetězec

```
String radka = "123  45 6 789";
String[] podretezce = radka.split(" ");
```

vypíše

```
|123|
||
|45|
|6|
|789|
```

- to prakticky znamená, že když si nejsme jisti vícečetností oddělovačů, musíme pak ve výsledném poli řetězců testovat výskyt prázdného řetězce, např.:

```
String radka = "123 45 6 789";
String[] podretezce = radka.split(" ");
for (int i = 0; i < podretezce.length; i++) {
    if (podretezce[i].length() > 0) {
        int cislo = Integer.parseInt(podretezce[i]);
        System.out.println(cislo);
    }
}
```

vypíše

```
123
45
6
789
```

- je-li v původním řetězci některý z oddělovacích znaků na samém konci nebo je-li zdvojen na samém konci, není prázdný řetězec součástí pole řetězců

```
String radka = "123;45;6;789;;";
String[] podretezce = radka.split(";");
```

vypíše

```
|123|
|45|
|6|
|789|
```

1.6.2. Použití regulárních výrazů

- zde bude ukázáno pouze základní použití – pro sofistikované použití viz `java.util.regex.Pattern`
- jednodušší případ je, když jsou oddělovače sice různé, ale je jich konečný počet (jsou konkrétně předem známy)

- pak se všechny potenciální oddělovače vypíší do hranatých závorek
 - ♦ na jejich pořadí nezáleží
 - ♦ pokud bude nějaký oddělovač navíc, nevádí to

```
String radka = "123 45;6;789";
String[] podretezce = radka.split("[; >:]*");
```

```
for (int i = 0; i < podretezce.length; i++) {
    System.out.println("|" + podretezce[i] + "|");
}
```

vypíše (znak `>` nebyl v původním řetězci a proto nebyl použit)

```
|123|
|45|
|6|
|789|
```

Varování

Snažíme se uvádět co nejmenší počet možných oddělovačů, protože některé znaky mohou mít v regulárních výrazech jiný význam. Typickým problémovým znakem je znak `.` (tečka).

```
String radka = "soubor.txt";
String[] podretezce = radka.split(".");
```

správné použití, které vypíše

```
|soubor|
|txt|
```

nesprávné použití (bez hranatých závorek) je:

```
String radka = "soubor.txt";
String[] podretezce = radka.split(".");
```

to nevypíše nic, protože tečka má význam „jakýkoliv znak“, takže jsou všechny znaky vstupního řetězce považovány za oddělovače a „požerou“ se

- komplikovanější případ je, když dopředu neznáme konkrétní znaky, ale pouze jejich kategorie
 - v podstatě až zde by se mělo mluvit o regulárních výrazech
 - znaky stejné kategorie lze zadat jako jeden výraz
 - ♦ v dokumentaci k `java.util.regex.Pattern` je uvedeno přehledně pět různých oblastí skupin znaků (plus další možnosti)
 - ♦ pro běžné použití stačí pouze jedna oblast – *POSIX character classes*
 - Pozor na to, že fungují jen pro znakovou sadu US-ASCII (tj. ne na akcentovaná písmena – viz později!)
 - ♦ přehled možností *POSIX character classes*
 - `\p{Lower}` – malá písmena
 - `\p{Upper}` – velká písmena
 - `\p{Alpha}` – malá a velká písmena
 - `\p{Digit}` – číslice

- `\p{XDigit}` – hexadecimální číslice
- `\p{Alnum}` – malá a velká písmena a číslice
- `\p{Punct}` – interpunkce – libovolný ze znaků `!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~`
- `\p{Blank}` – mezera a tabulátor
- `\p{Space}` – bílé znaky – mezera, tabulátor, `<CR>`, `<LF>`, nová stránka
- `\p{Cntrl}` – řídicí znaky – `\x00` až `\x1F`
- `\p{Graph}` – viditelné znaky – `\p{Alnum}` a `\p{Punct}`
- `\p{Print}` – tisknutelné znaky – `\p{Graph}` a mezera
- `\p{ASCII}` – všechny ASCII znaky, tj. sedmibitové znaky

♦ Pozor: při použití ve zdrojovém kódu je nutno zdvojit úvodní zpětné lomítko!

♦ použití oddělovačů z kategorie `\p{Punct}`

```
String radka = "soubor.txt";
String[] podretezce = radka.split("\\p{Punct}");
```

♦ použití oddělovačů z kategorie `\p{Alpha}`

```
String radka = "123a45A6";
String[] podretezce = radka.split("\\p{Alpha}");
```

vypíše

```
|123|
|45|
|6|
```

Poznámka

Regulární výrazy se nepoužívají jen pro parsování řetězců, ale za stejných podmínek např. i pro vyhledávání vzorů v řetězci – viz metodu `String.matches()`.

1.7. Podpora práce s poli – třída Arrays

pro ukládání objektů do paměti máme v zásadě dvě možnosti:

■ pole, podporované třídou `Arrays`

- výhody – rychlost, primitivní datové prvky i objekty
- nevýhody – pevná velikost, malá podpora přídatnými funkcemi, menší úroveň bezpečnosti (synchronizace apod.)

■ kolekce = třídy z balíku *Collection Framework* – `java.util`

- výhody a nevýhody víceméně opačné
- používají se více než pole
- viz další přednáška

1.7.1. Možnosti třídy Arrays

■ klasická pole mají jen jednu schopnost navíc – atribut `length`

■ nemají žádné metody, např. pro seřazení pole

■ `java.util.Arrays` – poskytuje existujícímu poli určité služby

■ všechny metody jsou statické – zpracovávané pole se předává jako skutečný parametr

■ např. existuje-li pole `abc`, pak jeho seřazení vyvoláme:

1. správně `Array.sort(abc);`

2. nesprávně `abc.sort();`

■ Metody jsou (některé umí pracovat i s částí pole):

• `equals()` – porovná dvě pole

• `asList()` – převede pole na kolekci

• `toString()` – převede pole na řetězec (velmi vhodné pro jednoduchý tisk pole)

• `fill()` – naplní část nebo celé pole konstantní hodnotou

• `sort()` – seřadí část nebo celé pole vzestupně

• `binarySearch()` – v poli seřazeném metodou `sort()` vrátí index prvku, který se shoduje se zadanou hodnotou

♦ pokud v poli není prvek této hodnoty, vrací záporné číslo

♦ toto číslo nemá konstantní hodnotu, ale v absolutní hodnotě představuje index do pole, kam by mohla být zadaná hodnota vložena

♦ protože však 0 je platný index a -0 nelze použít, je záporná hodnota ještě zmenšena o 1

Příklad 1.2. Použití metod třídy Arrays

```
import java.util.*;

public class ArraysPlneniSerazeniAVyhledavani {
    final static int POCET = 5;
    static int[] pole = new int[POCET];

    public static void main(String[] args) {
        Arrays.fill(pole, 3);
        System.out.println(Arrays.toString(pole));

        for (int i = 0; i < pole.length; i++) {
            pole[i] = (POCET - i) * 2;
        }
        System.out.println(Arrays.toString(pole));

        Arrays.sort(pole);
        System.out.println(Arrays.toString(pole));

        int k = Arrays.binarySearch(pole, 6);
        if (k >= 0)
            System.out.println "[" + k + "]=" + pole[k];
    }
}
```

vypiše

```
[3, 3, 3, 3, 3]
[10, 8, 6, 4, 2]
[2, 4, 6, 8, 10]
[2]=6
```

1.8. Řazení objektů

- pro pole primitivních datových prvků jsou `sort()` a `binarySearch()` jednoduše použitelné
- totéž platí pro objekty knihovnických tříd (`String`, `Integer` apod.), které mají jen jednu hodnotu (zjednodušeně řečeno)
- obsahuje-li pole obecné objekty, nemůže být bez upřesnění známo, jakým způsobem se budou objekty navzájem porovnávat mezi sebou
- dvě možnosti:
 1. přirozené řazení (*natural ordering*)
 2. absolutní řazení (*total ordering*)

Výstraha

Všechny zde uváděné informace platí beze zbytku i pro kolekce!

1.8.1. Přirozené řazení (*natural ordering*)

- metody `sort()` a `binarySearch()` voláme stejně, jako v případě polí s primitivními datovými typy
- způsob porovnávání objektů musí být popsán ve třídě těchto objektů
 - nutno implementovat rozhraní `java.lang.Comparable<Typ>`, které má pouze jednu metodu `compareTo(Typ t)`
 - ♦ `compareTo()` vrací `int` s hodnotou 0, pokud jsou objekty stejné, zápornou, pokud je parametr menší, a kladnou, pokud je parametr větší než objekt
 - ♦ metoda `compareTo()` musí být `public`
 - rozhraní `Comparable` (a tudíž metodu `compareTo()`) implementují všechny obalové třídy primitivních datových typů i třída `String`
 - pro přirozené seřazení těchto typů nemusíme psát `compareTo()`, stačí zavolat metodu `Arrays.sort()`
- budeme-li řadit objekty libovolných tříd, implementací `Comparable` určíme, podle čeho budeme řadit

Příklad 1.3. Přirozené řazení

Příklad objektu `Osoba`, který bude mít atributy `vaha` a `vyska`. Rozhraní `Comparable` ale má pouze jednu metodu `compareTo()` – nelze ji přetížit. Musíme si vybrat jeden atribut, podle kterého bude přirozené řazení probíhat.

```
import java.util.*;

class Osoba implements Comparable<Osoba> {
    int vyska;
    double vaha;
    String popis;

    Osoba(int vyska, double vaha, String popis) {
        this.vyska = vyska;
        this.vaha = vaha;
        this.popis = popis;
    }

    public int compareTo(Osoba os) {
        int osVyska = os.vyska;
        if (this.vyska > osVyska)
            return +1;
        else if (this.vyska == osVyska)
            return 0;
        else
            return -1;
    }

    public String toString() {
        return "vy = " + vyska +
            ", va = " + vaha +
            ", " + popis;
    }
}

public class ArraysPrirozenoRazeniObecnýchObjektu {
    public static void main(String[] args) {
        Osoba[] poleOsob = new Osoba[4];
        poleOsob[0] = new Osoba(186, 82.5, "muz");
        poleOsob[1] = new Osoba(172, 63.0, "zena");
        poleOsob[2] = new Osoba(105, 26.1, "dite");
        poleOsob[3] = new Osoba(116, 80.5, "obezni trpaslik");

        Arrays.sort(poleOsob);

        for (int i = 0; i < poleOsob.length; i++)
            System.out.println "[" + i + "] " + poleOsob[i]);
    }
}
```

Vypíše:

```
[0] vy = 105, va = 26.1, dite
[1] vy = 116, va = 80.5, obezni trpaslik
[2] vy = 172, va = 63.0, žena
[3] vy = 186, va = 82.5, muz
```

■ jedna z nevýhod přirozeného řazení – při použití třídy `Osoba` si nemůžeme vybírat, zda budeme řadit podle výšky či podle váhy nebo popisu

- ve třídě `Osoba` využívá `compareTo()` atribut `vyska` – objekty této třídy lze přirozeným řazením řadit vždy jen podle výšky

■ přirozené řazení používáme nejčastěji pro pole objektů s jednou hodnotou, podle které se přirozeně řadí

- u tříd, které rozhraní `Comparable` neimplementují, nejde přirozené řazení použít

1.8.2. Absolutní řazení (*total ordering*)

■ používáme přetížené verze metod `sort()` ze třídy `Arrays`, které mají jako poslední parametr objekt třídy, která implementuje rozhraní `java.util.Comparator`

■ při řazení máme absolutní kontrolu nad procesem řazení, bez ohledu na případné přednastavení řazených objektů vyjádřené metodou `compareTo()`

`Comparator<Typ>` má dvě metody:

1. `boolean equals(Object obj)` – v naprosté většině případů neimplementujeme (dědí ji každá třída ze třídy `Object`)
2. `int compare(Typ t1, Typ t2)` – platí stejná pravidla jako pro `compareTo()` z `java.lang.Comparable`, tj. vrací `int` s hodnotou 0, pokud jsou objekty stejné, zápornou, pokud je `t1` menší než `t2`, a kladnou v opačném případě

■ metoda `compare()` musí být `public`

Poznámka

Napišeme-li metodu `compare()` nebo `compareTo()` podle zadání, bude se řadit vzestupně. Chceme-li sestupné řazení, můžeme znaménko návratové hodnoty obrátit. To ale není dobrá taktika! Pro `compareTo()` lze použít i `reverseOrder()` z `java.util.Collections`.

Příklad 1.4. Ukázka absolutního řazení

V následujícím příkladě ponecháme zcela beze změny třídu `Osoba` z předchozího příkladu (včetně její metody `compareTo()`), což nám umožní použít i přirozené řazení -- zde zakomentované).

Abychom mohli řadit jak podle výšky, podle váhy i podle popisu, napíšeme tři pomocné třídy `KomparatorOsobyPodleXyz`, jejichž anonymní objekty předáme jako druhý parametr metodě `sort()`.

Ve třídě `KomparatorOsobyPodlePopisu` můžeme bez problémů využít metodu `compareTo()` (z "konkurenčního" řazení), protože ji třída `String` dává k dispozici.

V posledním řazení seřadíme s využitím `reverseOrder()` osoby podle výšky sestupně, přičemž se (vnitřně) používá metoda `compareTo()` ze třídy `Osoba`, nikoliv metoda `compare()` třídy `KomparatorOsobyPodleVysky`.

```
import java.util.*;

class KomparatorOsobyPodleVysky implements Comparator<Osoba> {
    public int compare(Osoba o1, Osoba o2) {
        int v1 = o1.vyska;
        int v2 = o2.vyska;
        return v1 - v2;
    }
}

class KomparatorOsobyPodleVahy implements Comparator<Osoba> {
    public int compare(Osoba o1, Osoba o2) {
        return (int) (o1.vaha - o2.vaha);
    }
}

class KomparatorOsobyPodlePopisu implements Comparator<Osoba> {
    public int compare(Osoba o1, Osoba o2) {
        String s1 = o1.popis;
        String s2 = o2.popis;
        return s1.compareTo(s2);
    }
}

public class OsobaAbsolutniRazeni {
    static Osoba[] poleOsob;

    static void vypisOsoby() {
        for (int i = 0; i < poleOsob.length; i++)
            System.out.println "[" + i + "] " + poleOsob[i].toString());
    }

    public static void main(String[] args) {
        poleOsob = new Osoba[4];
        poleOsob[0] = new Osoba(186, 82.5, "muz");
        poleOsob[1] = new Osoba(172, 63.0, "zena");
        poleOsob[2] = new Osoba(105, 26.1, "dite");
        poleOsob[3] = new Osoba(116, 80.5, "obezni trpaslik");
```

```
/* System.out.println("Prirozene razeni");
Arrays.sort(poleOsob);
vypisOsoby();

*/

System.out.println("Absolutni razeni podle vahy");
Arrays.sort(poleOsob, new KomparatorOsobyPodleVahy());
vypisOsoby();

System.out.println("Absolutni razeni podle popisu");
Arrays.sort(poleOsob, new KomparatorOsobyPodlePopisu());
vypisOsoby();

System.out.println("Prirozene razeni podle vysky sestupne");
Arrays.sort(poleOsob, Collections.reverseOrder());
vypisOsoby();
}
```

vypiše:

```
Absolutni razeni podle vahy
[0] vy = 105, va = 26.1, dite
[1] vy = 172, va = 63.0, zena
[2] vy = 116, va = 80.5, obezni trpaslik
[3] vy = 186, va = 82.5, muz
Absolutni razeni podle popisu
[0] vy = 105, va = 26.1, dite
[1] vy = 186, va = 82.5, muz
[2] vy = 116, va = 80.5, obezni trpaslik
[3] vy = 172, va = 63.0, zena
Prirozene razeni podle vysky sestupne
[0] vy = 186, va = 82.5, muz
[1] vy = 172, va = 63.0, zena
[2] vy = 116, va = 80.5, obezni trpaslik
[3] vy = 105, va = 26.1, dite
```

1.8.2.1. Binární vyhledávání pomocí metod absolutního řazení

- objekty tříd vzniklé implementací rozhraní `Comparator<Typ>` se po seřazení pole dají použít i pro binární vyhledávání metodou `binarySearch()`
 - `binarySearch()` je opět přetížena, takže poslední parametr jejího volání je objekt typu `Comparator`
- pokud je v poli více prvků (objektů) se stejnou hodnotou, není řečeno, který z nich je metodou `binarySearch()` nalezen, tzn. může být nalezen první z nich, ale také libovolný další

Kapitola 2. Kolekce a genericita (1)

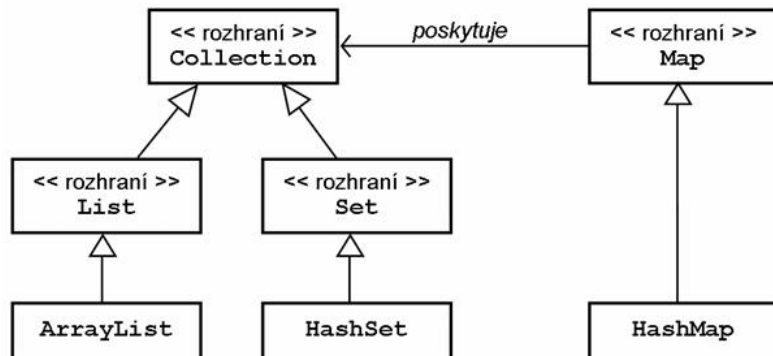
2.1. Úvodní informace

- objekty tříd z balíku `java.util` – Java Collections Framework
- slouží k uschovávání většího předem neznámého množství objektů libovolného typu
- „kolekce“ – dva nejužívanější představitelé – `ArrayList` a `HashSet` implementují rozhraní `java.util.Collection` (neplést si se třídou `java.util.Collections`)
- třetí významný představitel `HashMap` ale `java.util.Collection` neimplementuje

Poznámka

Nepoužívat „staré dědictví“ `Vector`, `Hashtable` a `Dictionary`

- výhody (prakticky převažují nad nevýhodami)
 - zjednodušují program – vše je hotovo, lepší čitelnost a přehlednost
 - výrazně zrychlují vývoj programu a umožňují jeho vyladění
 - typ a počet uschovávaných objektů není omezen – uchovávají `Object`
- nevýhody
 - nelze vkládat přímo primitivní datové typy – použijeme obalovací třídy (int uložíme jako `Integer`)
 - většinou pomalejší než pole
- celá knihovna kolekce velmi rozsáhlá a obsahuje ve své úplnosti 10 rozhraní, 5 abstraktních tříd a 13 tříd
- pokud ale nevyžadujeme žádné speciality a nezáleží nám (zpočátku) na maximální možné efektivnosti programu, potřebujeme pouze čtyři rozhraní a tři třídy



1. List – rozhraní k seznamům – uspořádaná kolekce

Seznam – `ArrayList`

- nejvíce připomíná „klasické“ pole, ovšem s proměnnou délkou
- pro přístup k jednotlivým prvkům lze používat indexy, protože prvky jsou udržovány v určitém pořadí

2. Set – rozhraní k množinám – neuspořádaná kolekce

Množina – `HashSet`

- obsahuje pouze unikátní prvky (nemá stejné prvky)
- pro přístup k jednotlivým prvkům nelze použít index, ale výhradně jen iterátor
- z hlediska efektivnosti implementace se pro přístup k prvkům množiny používá hešovací funkce – pro uživatele třídy `HashSet` naprosto skryto v implementaci (ale musí napsat metodu `hashCode()`)

3. Map – rozhraní k mapám

Mapa (asociativní pole) – `HashMap`

- uložení dvojic objektů, které jsou ve vzájemném vztahu klíč-hodnota
- pomocí klíče prvek vyhledáváme, ale zajímá nás nejen hodnota klíče, ale i hodnota prvku, se kterou je klíč svázán
- nejsou možné dva stejné klíče – při stejném klíči uschová naposledy vloženou hodnotu
- zjednodušený pohled – mapa je databáze o dvou sloupcích nebo dvojice `ArrayList`

Výstraha

- do JDK 1.5 byly všechny kolekce beztypové – do kolekce lze uložit cokoliv, ale při výběru musíme přetypovávat
- překlad beztypových kolekcí pod JDK 1.5 je možný, pouze se vypíše varovné hlášení:

Note: Some input files use unchecked or unsafe operations.

Note: Recompile with `-Xlint:unchecked` for details.

- od JDK 1.5 je možné kolekce typovat, což přináší větší bezpečnost kódu a větší komfort programátora (nemusí přetypovávat)
- v API je typování značeno jako `<E>` jako „element“
- `E` je třída nebo rozhraní

zápis dříve:

```
ArrayList arnt = new ArrayList();
arnt.add(new Integer(1));
arnt.add(new Integer(2));
arnt.add("tri");
```

```
for (Iterator it = arnt.iterator(); it.hasNext(); ) {
    Integer i = (Integer) it.next();
    System.out.print(i.intValue() + ", ");
}
```

zápis v JDK 1.5:

```
ArrayList<Integer> art = new ArrayList<Integer>();
art.add(new Integer(1));
art.add(new Integer(2));
art.add("tri"); // chyba při kompilaci
for (Iterator<Integer> it = arnt.iterator(); it.hasNext(); ) {
    System.out.print(it.next().intValue() + ", ");
}
```

Poznámka

Další vylepšení z JDK 1.5 viz později

Poznámka

Kolekce všech tří skupin lze snadno používat i pro více rozměrů, kdy např. prvky seznamu `ArrayList` mohou být buď další `ArrayList`, nebo množiny nebo mapy.

Příklad 2.1.

Základní odlišnosti tří hlavních typů kolekcí. Do `ArrayList` lze uložit více stejných prvků, kdežto do `HashSet` ani do `HashMap` nikoliv. Obsahy všech kolekcí lze bez problémů tisknout.

```
import java.util.*;

public class OdlisnostiKolekci {
    public static void main(String[] args) {
        ArrayList<String> ar = new ArrayList<String>();
        ar.add("prvni");
        ar.add("druhy");
        ar.add("prvni");
        System.out.println("ArrayList: " + ar);

        HashSet<String> hs = new HashSet<String>();
        hs.add("prvni");
        hs.add("druhy");
        hs.add("prvni");
        System.out.println("HashSet: " + hs);

        HashMap<String, String> hm = new HashMap<String, String>();
        hm.put("prvni", "objekt");
        hm.put("druhy", "objekt");
        hm.put("prvni", "pivo");
        System.out.println("HashMap: " + hm);
    }
}
```

Vypíše:

```
ArrayList: [prvni, druhy, prvni]
HashSet:   [prvni, druhy]
HashMap:   {prvni=pivo, druhy=objekt}
```

2.2. Použití wildcard – unbounded wildcard

- nevíme-li dopředu, jaké typy objektů budou v kolekci uloženy, je možné použít `<?>`
 - používá se ve formálních parametrech metod
 - lze vidět v API: `boolean removeAll(Collection<?> c)`
 - nelze použít pro deklaraci: `ArrayList<?> ar = new ArrayList<?>();`
- v našich programech zásadně nepoužívat – ztrácíme výhody typování

Příklad 2.2.

```
public static void main(String[] args) {
    ArrayList<Object> ar = new ArrayList<Object>();
    ar.add("jedna");
    ar.add(new Integer(2));
    tisk(ar);
}

public static void tisk(ArrayList<?> a) {
    for (Iterator<?> it = a.iterator(); it.hasNext();) {
        Object o = it.next();
        if (o instanceof String) {
            System.out.println(o.toString());
        }
        if (o instanceof Integer) {
            System.out.println(((Integer) o).intValue());
        }
    }
}
```

2.3. Využití polymorfismu – bounded wildcard

■ do kolekcí občas vkládáme i typově příbuzné prvky, kdy využíváme polymorfismu

- pak je nutná deklarace s rozšířeným využitím `<? extends Typ>`
- používá se ve formálních parametrech metod
- v API: `boolean addAll(Collection<? extends T> c)`
- nelze použít pro deklaraci:

```
ArrayList<? extends T> ar = new ArrayList<? extends T>();
```

■ jako bázeový typ lze samozřejmě použít i rozhraní

Příklad 2.3.

```
interface T {
    public void tiskni();
}

class A implements T {
    public void tiskni() {
        System.out.println("A");
    }
}

class B extends A {
    public void tiskni() {
        System.out.println("B potomek A");
    }
}

public class TypovanaKolekcePolymorfismus {
    public static void main(String[] args) {
        ArrayList<T> ar = new ArrayList<T>();
        ar.add(new A());
        ar.add(new B());
        tisk(ar);
    }

    public static void tisk(ArrayList<? extends T> a) {
        for (Iterator<? extends T> it = a.iterator(); it.hasNext(); ) {
            it.next().tiskni();
        }
    }
}
```

2.4. Rozhraní Collection

■ základ seznamů, definuje množství metod

1. Metody pro plnění kolekce:

- `boolean add(E e)` – vložení jednoho prvku
- `boolean addAll(Collection<? extends E> c)` – vložení všech prvků, nacházejících se v jiné kolekci

2. Metody pro ubírání kolekce:

- `void clear()` – odstranění všech prvků z kolekce
- `boolean remove(E e)` – odstranění jednoho prvku
- `boolean removeAll(Collection<?> c)` – odstranění všech prvků, nacházejících se v jiné kolekci

- `boolean retainAll(Collection <?> c)` – ponechání pouze prvků, nacházejících se v jiné kolekci

3. Logické operace

- `int size()` – vrátí aktuální počet prvků kolekce
- `boolean isEmpty()` – test na prázdnou kolekci
- `boolean contains(E e)` – test, zda je daný prvek obsažen (alespoň jednou) v kolekci
- `boolean containsAll(Collection <?> c)` – test, zda jsou všechny prvky jiné kolekce obsaženy v kolekci

4. Převod kolekce na běžné pole

- `Object[] toArray()`

5. Získání přístupového objektu

- `Iterator <E> iterator()`

2.5. Rozhraní List

- přidává metody, které zavádějí možnost práce s prvky kolekce pomocí indexů:

1. Změny v kolekci:

- `void add(int index, E e)` – přidání prvku; prvky s vyšším indexem budou posunuty o jeden výše
- `E set(int index, E e)` – změna prvku na daném indexu; prvkům s vyšším indexem se indexy nemění
- `E remove(int index)` – odstranění prvku; prvky s vyšším indexem budou posunuty o jeden níže

2. Získání obsahu kolekce

- `E get(int index)` – vrátí prvek na daném indexu, ale současně jej ponechá v kolekci (rozdíl od `remove()`)
- `int indexOf(E e)` – vrátí index prvního nalezeného prvku, nebo -1, není-li prvek v kolekci
- `int lastIndexOf(E e)` – vrátí index posledního nalezeného prvku
- `List<E> subList(int startIndex, int endIndex)` – vrátí podseznam, ve kterém budou prvky od `startIndex` včetně do `endIndex-1`; Pozor, jedná se o mělkou kopii.

- rozhraní `Set` zděděné od `Collection` nepřidává žádné metody navíc

2.6. ArrayList

- nejpoužívanější typ kolekce, na který přecházíme, přestávají-li nám stačit klasická pole

- kromě své velikosti, tj. aktuálního počtu prvků vráceného metodou `size()` má i kapacitu – důležité pro efektivitu implementace

Poznámka

S řetězci se pracuje v kolekcích elegantně, protože třída `String` má ze strany kolekcí zvláštní podporu (`ar.get(i)`).

Příklad 2.4.

```
import java.util.*;

public class ArrayListMetodyZCollection {
    public static void tiskni(String jmeno, ArrayList ar) {
        int vel = ar.size();
        System.out.print(jmeno + " (" + vel + ") : ");
        for (int i = 0; i < vel; i++) {
            System.out.print "[" + i + "]=" + ar.get(i) + ", ";
        }
        System.out.println();
    }

    public static void main(String[] args) {
        ArrayList<String> ar1 = new ArrayList<String>();
        System.out.println("ar1 je prazdny: " + ar1.isEmpty());
        ar1.add("prvni");
        ar1.add("druhy");
        ar1.add("prvni");
        tiskni("ar1", ar1);

        System.out.println("\nPridavani a ubirani prvku");
        ArrayList<String> ar2 = new ArrayList<String>(ar1);
        ar2.add("treti");
        tiskni("ar2", ar2);
        ar2.remove("prvni");
        tiskni("ar2", ar2);
        ar2.removeAll(ar1);
        tiskni("ar2", ar2);
        ar2.addAll(ar1);
        tiskni("ar2", ar2);
        ar2.retainAll(ar1);
        tiskni("ar2", ar2);

        System.out.println("\nHledani prvku");
        ArrayList<String> ar3 = new ArrayList<String>(ar1);
        ar3.add("ctvrty");
        System.out.println("ar3 obsahuje 'paty': "
            + ar3.contains("paty"));
        System.out.println("ar3 obsahuje ar1: "
            + ar3.containsAll(ar1));

        System.out.println("\nPrevod na pole");
        String[] s = (String[]) ar1.toArray(new String[0]);
        System.out.println(Arrays.asList(s));
    }
}
```

Vypiše:

```
ar1 je prazdny: true
ar1 (3) : [0]=prvni, [1]=druhy, [2]=prvni,
```

```
Pridavani a ubirani prvku
ar2 (4) : [0]=prvni, [1]=druhy, [2]=prvni, [3]=treti,
ar2 (3) : [0]=druhy, [1]=prvni, [2]=treti,
ar2 (1) : [0]=treti,
ar2 (4) : [0]=treti, [1]=prvni, [2]=druhy, [3]=prvni,
ar2 (3) : [0]=prvni, [1]=druhy, [2]=prvni,
```

```
Hledani prvku
ar3 obsahuje 'paty': false
ar3 obsahuje ar1: true
```

```
Prevod na pole
[prvni, druhy, prvni]
```

Příklad 2.5.

Ve třídě `CeleCislo` je také překrytá metoda `toString()`, díky níž můžeme tisknout obsah celého seznamu najednou. Argumentem metody `tiskni()` je objekt třídy `List`. Zde s výhodou využíváme toho, že rozhraní může zastoupit třídu. To je výhodné proto, že metoda `subList()` vrací seznam `List` nikoli `ArrayList`.

Na vráceném seznamu lze ukázat ještě něco, nač je třeba dát při práci s objekty pozor – kopie seznamu je mělká kopie což prakticky znamená, že pokud v kopii změníme hodnotu objektu, změní se tato i v originálu (nebo naopak).

```
import java.util.*;

class CeleCislo {
    private int cislo;

    CeleCislo(int i) { this.cislo = i; }

    int getCislo() { return cislo; }

    void setCislo(int i) { this.cislo = i; }

    public String toString() { return (" " + cislo); }
}

public class ArrayListVlastniTridaMetodyZList {
    static void tiskni(String jmeno, List<CeleCislo> li) {
        int vel = li.size();
        System.out.print(jmeno + " (" + vel + ") : ");
        for (int i = 0; i < vel; i++) {
            System.out.print "[" + i + "]="
                + li.get(i).getCislo() + ", ";
        }
        System.out.println();
    }

    public static void main(String[] args) {
        System.out.println("Vytvoreni seznamu");
        ArrayList<CeleCislo> ar = new ArrayList<CeleCislo>();
        for (int i = 0; i < 5; i++) {
            ar.add(new CeleCislo(i + 10));
        }
        tiskni("ar", ar);
        System.out.println("Tisk celeho seznamu: " + ar);

        System.out.println("Pridavani prvku");
        ar.add(2, new CeleCislo(77));
        tiskni("ar", ar);
        System.out.println("Vytvoreni podseznamu");
        List<CeleCislo> sl = ar.subList(2, 5);
        tiskni("sl", sl);

        ar.get(3).setCislo(33);
    }
}
```

```
tiskni("ar", ar);
tiskni("sl", sl);
}
}
```

Vypiše:

```
Vytvoreni seznamu
ar (5) : [0]=10, [1]=11, [2]=12, [3]=13, [4]=14,
Tisk celeho seznamu: [10, 11, 12, 13, 14]
Pridavani prvku
ar (6) : [0]=10, [1]=11, [2]=77, [3]=12, [4]=13, [5]=14,
Vytvoreni podseznamu
sl (3) : [0]=77, [1]=12, [2]=13,
ar (6) : [0]=10, [1]=11, [2]=77, [3]=33, [4]=13, [5]=14,
sl (3) : [0]=77, [1]=33, [2]=13,
```

2.7. Třída Collections

- pro běžné pole existuje třída `Arrays` se svými statickými metodami, které slouží pro vyplnění, seřazení a vyhledávání v poli

- pro třídy, které implementují rozhraní `Collection`, existuje třída `java.util.Collections`

- poskytuje podobné metody jako `Arrays` a ještě mnohé další, většinou ale jen pro seznamy `List` nikoliv pro množiny `Set`

- všechny metody jsou opět statické

Některé metody třídy `Collections`:

- vyplnění seznamu jednou (stejnou) hodnotou

- `void fill(List<E> list, E e)`

- pro řazení lze opět použít oba dva již známé způsoby – přirozeného řazení (`compareTo()` patřící třídě řazených objektů) nebo absolutního řazení (metoda `compare()` z vnějšího komparátoru)

- `void sort(List<E> list)` – vzestupné přirozené řazení

- `void sort(List<E> list, Comparator<E> c)` – absolutní řazení podle komparátoru

- v seřazeném seznamu lze rychle vyhledávat

- `int binarySearch(List<E> list, E key)` - hledání s využitím `compareTo()`

- `int binarySearch(List<E> list, E key, Comparator<E> c)` – hledání s pomocí externího komparátoru

- vyhledávání v neseřazeném seznamu (trvá ale mnohem delší dobu)

- `int indexOf(E e)`

- nalezení prvku s minimální a maximální hodnotou v neseřazeném seznamu (opět lze použít obou typů porovnávání)

- `E max(Collection<E> coll)`
- `E max(Collection<E> coll, Comparator<E> comp)`
- `E min(Collection<E> coll)`
- `E min(Collection<E> coll, Comparator<E> comp)`

■ otočení pořadí (většinou již seřazeného) seznamu

- `void reverse(List<E> l)`

■ pokud k řazení využíváme způsob přirozeného řazení, pak lze lehce změnit vzestupné pořadí na sestupné tím, že si necháme vygenerovat komparátor měnící pořadí

- `Comparator<E> reverseOrder()`

■ "zamíchání" seznamu – výhodné když jsou v seznamu jednoznačně definované prvky (např. pexeso) a my jen potřebujeme vždy jejich jiné pořadí

- `void shuffle(List<E> list)`

Příklad 2.6.

```
public class OsobaCollections {
    public static void main(String[] args) {
        List<Osoba> sez = new ArrayList<Osoba>();
        sez.add(new Osoba(186, 82.5, "muz"));
        sez.add(new Osoba(172, 63.0, "zena"));
        sez.add(new Osoba(105, 26.1, "dite"));
        sez.add(new Osoba(116, 80.5, "obezni trpaslik"));
        System.out.println("Neserazeno: " + sez);

        Collections.sort(sez, new KomparatorOsobyPodleVahy());
        System.out.println("Absolutni razeni podle vahy: " + sez);

        Collections.reverse(sez);
        System.out.println("Podle vahy sestupne: " + sez);

        Collections.sort(sez);
        System.out.println("Prirozene razeni podle vysky: " + sez);

        Collections.shuffle(sez);
        System.out.println("Zamichano: " + sez);

        System.out.println("Nejvyssi:" + Collections.max(sez));
        System.out.println("Nejlehci:" +
            Collections.min(sez, new KomparatorOsobyPodleVahy()));

        Collections.fill(sez, new Osoba(180, 75.0, "robot"));
        System.out.println("Vyplneno: " + sez);
    }
}
```

Vypiše např.:

```
Neserazeno: [
vy = 186, va = 82.5, muz,
vy = 172, va = 63.0, zena,
vy = 105, va = 26.1, dite,
vy = 116, va = 80.5, obezni trpaslik]
Absolutni razeni podle vahy: [
vy = 105, va = 26.1, dite,
vy = 172, va = 63.0, zena,
vy = 116, va = 80.5, obezni trpaslik,
vy = 186, va = 82.5, muz]
Podle vahy sestupne: [
vy = 186, va = 82.5, muz,
vy = 116, va = 80.5, obezni trpaslik,
vy = 172, va = 63.0, zena,
vy = 105, va = 26.1, dite]
Prirozene razeni podle vysky: [
vy = 105, va = 26.1, dite,
vy = 116, va = 80.5, obezni trpaslik,
vy = 172, va = 63.0, zena,
```



```
vy = 186, va = 82.5, muz]
Zamichano: [
vy = 172, va = 63.0, zena,
vy = 186, va = 82.5, muz,
vy = 116, va = 80.5, obezni trpaslik,
vy = 105, va = 26.1, dite]
Nejvyssi:
vy = 186, va = 82.5, muz
Nejlehci:
vy = 105, va = 26.1, dite
Vyplneno: [
vy = 180, va = 75.0, robot,
vy = 180, va = 75.0, robot,
vy = 180, va = 75.0, robot,
vy = 180, va = 75.0, robot]
```

Kapitola 3. Kolekce a genericita (2)

3.1. Postupný průchod kolekcí

- možný pomocí indexů – jen pro seznamy
- nebo iterátorů – obecně pro všechny kolekce
- od JDK 1.5 jsou iterátory dvou typů
 - původní objekt třídy `Iterator`
 - „For-Each“ pomocí klíčového slova `for` (zjednodušený iterátor)
- rychlosti průchodu indexací a iterátorem jsou stejné
- použijeme-li jeden typ kolekce (např. `ArrayList`) a budeme jej procházet pomocí iterátoru, můžeme někdy v budoucnu (např. z důvodů zvýšení rychlosti) snadno změnit tento typ kolekce za jiný
- pouze se vytvoří jiná kolekce, ale vkládání a výběr prvků a průchod kolekcí zůstanou naprosto nezměněny

```
Collection<Typ> c = new ArrayList<Typ>();
```

nebo později:

```
Collection<Typ> c = new HashSet<Typ>();
```

3.1.1. For-Each

- díky typovaným kolekcím nejjednodušší a nejpoužívanější
 - je to též bezpečnější konstrukce
- musí projít celou kolekcí od prvního do posledního prvku
 - to v naprosté většině případů chceme
- lze jej použít i na běžné pole

Příklad 3.1.

```
ArrayList<Integer> ar = new ArrayList<Integer>();
ar.add(new Integer(1));
ar.add(new Integer(2));
for (Integer i: ar) {
    System.out.print(i.intValue() + ", ");
}
System.out.println("\nBezne pole");
int[] pole = {5, 6, 7, 8, 9};
for (int hod : pole) {
    System.out.print(hod + ", ");
}
```

Vypíše:

```
1, 2,
Bezne pole
5, 6, 7, 8, 9,
```

3.1.2. Iterátory

■ iterátory jsou odvozeny od rozhraní `java.util.Iterator<Typ>` (nepoužívat "starý" `Enumeration`)

■ objekty, které toto rozhraní implementují, vrací každá třída kolekci metodou `Iterator<Typ> iterator()`

■ v porovnání s `For-Each` používáme jen ve speciálních případech

- nejčastěji přesakování (vynechání) některých prvků

■ samotný iterátor umožňuje pouze tři aktivity:

- `boolean hasNext()` - zjistí, zda v kolekci existuje ještě nějaký prvek
- `Object next()` – přesune se na další prvek a vrátí jej
- `void remove()` – zruší prvek odkazovaný předchozím `next()`, NEvrací rušený prvek

Výstraha

iterátor je jednorůchodový

■ kolekce odvozené od `List` dokáží metodou `listIterator()` vrátit objekt splňující rozhraní `ListIterator<Typ>`

■ má navíc metody:

1. umožňující pohyb od konce seznamu k jeho počátku
2. změna prvku získaného předchozím `next()` nebo `previous()`
3. získání indexu pomocí iterátoru

4. přetížená metoda `listIterator(int zacIndex)` vrací iterátor fungující od uvedeného počátečního indexu

Poznámka

Použití specializovaného iterátoru je třeba zvážit, protože můžeme přijít v budoucnu o možnost zaměňovat jednotlivé typy kolekcí.

Příklad 3.2.

Na dvou způsobech tisku je vidět, jak se lze iterátor využít cyklu `for` i `while`. U cyklu `while` se nezpracovávají první dvě hrušky. Dále je vidět běžné využití překryté metody `toString()`.

```
import java.util.*;
```

```
class Hruska {
    private int cena;
    Hruska(int cena) { this.cena = cena; }
    public String toString() { return "" + cena; }
    public void tisk() { System.out.print(cena + ", "); }
}
```

```
public class IteratorZakladniPouziti {
    public static void main(String[] args) {
        ArrayList<Hruska> kosHrusek = new ArrayList<Hruska>();
        for (int i = 0; i < 10; i++) {
            kosHrusek.add(new Hruska(i + 20));
        }

        for (Iterator<Hruska> it = kosHrusek.iterator(); it.hasNext(); ) {
            System.out.print(it.next() + ", ");
        }
        System.out.println();

        Iterator<Hruska> it = kosHrusek.iterator();
        it.next();
        it.next();
        while (it.hasNext()) {
            it.next().tisk();
        }
        System.out.println();
    }
}
```

Vypíše:

```
20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
22, 23, 24, 25, 26, 27, 28, 29,
```

3.2. Automatické zapouzdřování primitivních datových typů

- též *autoboxing* a *unboxing*
- do kolekci lze ukládat jen objekty, tzn. primitivní datové typy lze vkládat pouze pomocí jejich obalovacích tříd
 - pak je otravná práce s vytvářením objektů a získáváním hodnot
- od JDK 1.5 to umí překladač zajistit automaticky, tj. provést přechod z primitivního datového typu na příslušnou obalovací třídu a naopak
 - této možnosti by se mělo využívat s rozmyslem a jen tam, kde nemůže dojít k nedorozumění
 - stále platí, že `int` a `Integer` jsou dva zcela rozdílné typy
- princip autoboxingu a unboxingu je zcela nezávislý na kolekcích

Příklad 3.3.

```
public class TypovanaKolekceBoxing {
    public static void main(String[] args) {
        ArrayList<Integer> ar = new ArrayList<Integer>();
        ar.add(1); // autoboxing
        ar.add(new Integer(2));
        for (Integer i: ar) {
            System.out.print(i + ", "); // unboxing
            System.out.print(i.intValue() + ", ");
        }

        Integer i1 = new Integer(5);
        Integer i2 = 6;
        int j1 = i1.intValue();
        int j2 = i2;
    }
}
```

3.3. Výhodnost jednotlivých seznamů

- pokud se používají jen metody z rozhraní `List`, je záměna různých typů seznamů velice jednoduchá a zvýšení (nebo též snížení) výkonu může být ohromující
- čas v benchmarkách je vypisován v milisekundách a samozřejmě závisí na typu procesoru (Athlon, 1,4 GHz) a velikosti operační paměti (512 MB)
- byly testovány dvě třídy – běžný `ArrayList`, `LinkedList`
- velikost seznamu byla 100 000 prvků

	ArrayList	LinkedList
naplnění	241	391
průchod indexací	10	552 454
průchod iterátorem	20	30
vypuštění poloviny indexací zezadu	5 477	178 107
vložení poloviny indexací	5 719	174 841
vypuštění poloviny indexací zepředu	57 723	175 743
clear a naplnění	100	360
vypuštění poloviny iterátorem	57 713	40

Jaké zajímavé závěry plynou z tabulky.

1. Průchod `ArrayListu` pomocí indexů a pomocí iterátoru je zcela srovnatelný, z čehož vyplývá, že je výhodnější používat iterátoru, protože to nám dává možnost budoucí záměny `ArrayListu` za jiný typ kolekce.
 2. Tytéž výsledky – a tedy i závěry – jsou i u vypouštění prvků pomocí indexace a iterátoru (zepředu).
 3. Jakákoliv indexace v `LinkedListu` je extrémně pomalá.
 4. Hromadné vypouštění nebo vkládání prvků do `ArrayListu` je časově velice náročné. U vypouštění prvků je `LinkedList` zhruba 1000 krát výkonnější, a je tedy velmi vhodné jej pro tento typ aplikace použít. Na druhé straně je ale problémem `LinkedListu` vkládání, které je zhruba 6krát pomalejší než u `ArrayListu`. Vkládat doprostřed lze totiž pouze indexací, která je u `LinkedListu` extrémně pomalá (vkládání pomocí iterátoru vyvolá výjimku). Záleží ale také na tom, kam se vkládá. Pokud by bylo vkládání jen na konec či na začátek, pak by bylo jistě rychlejší.
- ## 3.4. Ochrana proti nekonzistenci dat
- kolekce obsahují zabudovanou automatickou ochranu proti případu, kdy se pokusíme změnit kolekci za současného používání pohledu (iterátor, podseznam, ...)
 - ochrana způsobí vyvolání výjimky `ConcurrentModificationException`

Příklad 3.4.

Výjimku vyvolá i akce s kolekcí v jednom a též procesu. Není tedy pravda, že pro vyhození této výjimky musí být do kolekce souběžný přístup z více procesů. Použití `For-Each` tento problém eliminuje.

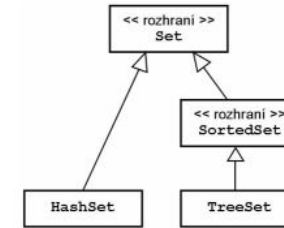
```
import java.util.*;
public class IteratorZmenaKolekce {
    public static void main(String[] args) {
        ArrayList<Integer> kont = new ArrayList<Integer>();
        for (int i = 0; i < 10; i++) {
            kont.add(new Integer(i));
        }
        Iterator<Integer> it = kont.iterator();
        System.out.println(it.next());
        kont.add(new Integer(20));
        // zde vyhodí výjimku
        System.out.println(it.next());
    }
}
```

Vypíše:

```
0
Exception in thread "main" java.util.ConcurrentModificationException
```

3.5. Množiny – rozhraní Set

- od rozhraní `java.util.Collection` je zděděno i rozhraní `java.util.Set`
 - představuje základ pro kolekce typu množiny
- množina se od seznamu liší tím, že umožňuje uschovávat pouze jeden prvek stejné hodnoty – všechny prvky množiny mají unikátní hodnotu
 - to vyžaduje, aby každý objekt vkládaný do množiny měl definovanou metodu `equals()`
- další výhodou neseřazené množiny oproti neseřazenému seznamu je větší rychlost vyhledání prvku
- nevýhodou oproti seznamu je, že množina nezaručuje, že budou vložené prvky uchovávány v nějakém definovaném pořadí
- pro kolekci se používá nejčastěji třída `java.util.HashSet`
 - lze použít i třídu `TreeSet` která implementuje `SortedSet`



- v `TreeSet` se průběžně udržují prvky seřazené (využívá stromovou strukturu)
 - již v době vložení se vkládaný prvek zařadí do odpovídajícího pořadí vzhledem ke stávajícím prvkům
 - vkládání do `TreeSet` je pomalejší než do `HashSet`
 - ♦ kupodivu pomalejší je i jakákoliv další práce s `TreeSet`, kromě vyhledávání prvku (kde však rozdíl rychlostí není významný)
- protože `Set` je stejně jako `List` zděděno od `Collection`, obsahuje stejné metody, které již byly popsány u `ArrayList`
 - cokoli, co mělo něco společného s indexy, které jsou běžné v seznamech, v množinách neexistuje
- jakýkoliv průchod kolekcí je možný pouze pomocí `For-Each` nebo iterátorů, jejichž princip je naprosto stejný jako u seznamů

Příklad 3.5.

Ukázka, jak pro množiny fungují běžné metody, jako je vkládání, zjištění velikosti kolekce, vyhledávání prvku, vypouštění prvku, průchod kolekcí pomocí iterátoru a vymazání obsahu kolekce.

```
public class HashSetATreeSet {

    public static void naplneniATisk(Set<String> st) {
        st.add("treti");
        st.add("prvni");
        st.add("druhy");
        // pokus o vložení stejného prvku
        if (st.add("treti") == false)
            System.out.println("'treti' podruhé nevložen");
        System.out.println(st.size() + " " + st);
        for (String s: st) {
            System.out.print(s + ", ");
        }
        if (st.contains("treti") == true) {
            System.out.println("\n'treti' je v množině");
        }
        st.remove("treti");
        System.out.println(st);
        st.clear();
    }

    public static void main(String[] args) {
        naplneniATisk(new HashSet<String>());
        naplneniATisk(new TreeSet<String>());
    }
}
```

Vypíše:

```
'treti' podruhé nevložen
3 [druhy, prvni, tretí]
druhy, prvni, tretí,
'treti' je v množině
[druhy, prvni]
podobně pro TreeSet
```

3.5.1. Práce s vlastní třídou v množině

■ předchozí příklad využíval skutečnosti, že třída `String` (stejně jako obalovací třídy) má správně naprogramované metody `compareTo()` (pro případné porovnávání), `equals()` (pro zjištění shodnosti prvků) a `hashCode()` (pro rychlý výběr prvku)

- máme-li však pracovat s objekty vlastních typů, musíme tyto tři metody přepsat – bez toho bude program chodit chybně

Příklad 3.6.

Do množiny budeme ukládat již známý typ `Hruska`. Metoda `hashCode()` je zde jednoduchá. Netypickým nastavením v `compareTo()` můžeme řadit prvky množiny sestupně. To ale většinou není dobrá implementace `compareTo()` – viz další příklad.

Pozor na typ parametru v metodě `equals()`. Ten musí být typu `Object` (nikoliv `Hruska`), aby došlo k překrytí metody `equals()` ze třídy `Object`. Použitím `Hruska` dojde pouze k přetížení a program bude pracovat chybně.

```
import java.util.*;

class Hruska implements Comparable<Hruska> {
    int cena;
    Hruska(int cena) { this.cena = cena; }

    public String toString() { return "" + cena; }

    // public boolean equals(Hruska o) { // chyba
    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (o instanceof Hruska == false)
            return false;
        boolean stejnáCena = (cena == ((Hruska) o).cena);
        return stejnáCena;
    }

    public int compareTo(Hruska h) { // sestupně řazení !!!
        int cenaPom = h.cena;
        if (cenaPom > cena)
            return (+1);
        else if (cenaPom == cena)
            return (0);
        else
            return (-1);
    }

    public int hashCode() {
        return cena;
    }
}

public class HruskyVMnožina {
    static void praceSHruskami(String typ, Set<Hruska> st) {
        for (int i = 20; i < 30; i++) {
            st.add(new Hruska(i));
        }
        st.add(new Hruska(25));

        System.out.print(typ + "-pocet: " + st.size() + " [");
        for (Hruska h: st) {
            System.out.print(h + ", ");
        }
        System.out.println("]");
    }
}
```

```

    }

    public static void main(String[] args) {
        praceSHruskami("HashSet", new HashSet<Hruska>());
        praceSHruskami("TreeSet", new TreeSet<Hruska>());
    }
}

```

Vypíše:

```

HashSet-pocet: 10 [26, 23, 28, 21, 29, 20, 25, 24, 27, 22,]
TreeSet-pocet: 10 [29, 28, 27, 26, 25, 24, 23, 22, 21, 20,]

```

3.5.2. Použití Collections

- ze všech metod tohoto rozhraní lze pro množiny použít pouze dvě (čtyři) metody pro nalezení největšího a nejmenšího prvku

- jedna dvojice metod používá přirozeného a druhá absolutního řazení

- `E max(Collection<E> coll)`
- `E max(Collection<E> coll, Comparator<E> comp)`
- `E min(Collection<E> coll)`
- `E min(Collection<E> coll, Comparator<E> comp)`

3.5.3. Rozhraní SortedSet

- `TreeSet` implementuje rozhraní `SortedSet` a uschovává prvky seřazené

- lze proto použít několik dalších metod

- `E first()` – vrací první prvek množiny
- `E last()` – vrací poslední prvek množiny
- `SortedSet<E> headSet(E horniMez)` – vrací podmnožinu prvků, které jsou uloženy před hraničním prvkem
- `SortedSet<E> tailSet(E dolniMez)` – vrací podmnožinu prvků, které jsou uloženy za hraničním prvkem, včetně tohoto prvku
- `SortedSet<E> subSet(E dolniMez, E horniMez)` – vrací podmnožinu prvků v zadaných mezích

Poznámka

První čtyři uváděné metody fungují i v případě, kdy metoda `compareTo()` vkládaných prvků vynucuje sestupné řazení. Pak má např. první prvek největší hodnotu a poslední prvek nejmenší hodnotu. Problém ale nastává, použijeme-li poslední metodu, která vrací prvky v zadaných mezích. Vynucuje-li metoda `compareTo()` sestupné řazení, pak bude vygenerována výjimka. Pro správnou činnost je nutné použít vzestupné řazení.

```

System.out.println("Pocty slov od jednotlivych pismen");
for (char ch = 'a'; ch <= 'e'; ch++) {
    String zac = new String(new char[] {ch});
    String kon = new String(new char[] {(char) (ch+1)});
    System.out.println(zac + ": " + treeset.subSet(zac, kon).size());
}

```

3.5.4. Množinové operace a triky

- vynikající např. při práci se jmény souborů atd.

- odstranění duplicit z `ArrayList`

```

public class EliminaceDuplicit {
    public static void main(String[] args) {
        Collection<String> d = new ArrayList<String>();
        d.add("prvni");
        d.add("druhy");
        d.add("prvni");
        System.out.println("Duplicitni: " + d);
        Collection<String> nd = new ArrayList<String>(
            new HashSet<String>(d));
        System.out.println("NEduplicitni: " + nd);
    }
}

```

Vypíše:

```

Duplicitni: [prvni, druhy, prvni]
NEduplicitni: [druhy, prvni]

```

- průniky, sjednocení apod.

```

Set<String> m1 = new HashSet<String>();
m1.add("1");
m1.add("2");
m1.add("3");
m1.add("4");
Set<String> m2 = new HashSet<String>();
m2.add("2");
m2.add("3");

if (m1.containsAll(m2) == true)
    System.out.println(m2 + " je podmnozinou " + m1);

m2.add("5");
Set<String> sjednoceni = new HashSet<String>(m1);
sjednoceni.addAll(m2);
System.out.println(sjednoceni + " je sjednocenim " + m1 + " a " + m2);
Set<String> prunik = new HashSet<String>(m1);
prunik.retainAll(m2);
System.out.println(prunik + " je prunikiem " + m1 + " a " + m2);

```

```

Set<String> rozdil = new HashSet<String>(m1);
rozdil.removeAll(m2);
System.out.println(rozdil+ " je rozdilem " + m1+ " a "+m2);

Set<String> symetrickyRozdil = new HashSet<String>(m1);
symetrickyRozdil.addAll(m2);
Set tmp = new HashSet(m1);
tmp.retainAll(m2);
symetrickyRozdil.removeAll(tmp);
System.out.println(symetrickyRozdil +
    " je symetrickym rozdilem " + m1 + " a " + m2);

```

Vypíše:

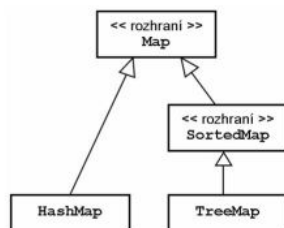
```

[3, 2] je podmnožinou [3, 2, 4, 1]
[3, 5, 2, 4, 1] je sjednocením [3, 2, 4, 1] a [3, 5, 2]
[3, 2] je pruníkem [3, 2, 4, 1] a [3, 5, 2]
[4, 1] je rozdílem [3, 2, 4, 1] a [3, 5, 2]
[5, 4, 1] je symetrickým rozdílem [3, 2, 4, 1] a [3, 5, 2]

```

3.6. Mapy – rozhraní Map

- též slovníky (*dictionary*) nebo asociativní pole (*associative array*)
- rozhraní `Map` nemá nic společného s rozhraním `Collection`, tj. mapy nejsou zaměnitelné za seznamy ani za množiny
 - z mapy ale lze získat seznam nebo množinu
- v mapě jeden prvek tvoří nedělitelná dvojice dvou objektů – klíče a hodnoty
 - pomocí klíče, který je neměnný a unikátní, se vyhledává hodnota
 - hodnota je proměnná a může být duplicitní, tj. dva různé klíče mohou mít stejnou hodnotu
- struktura tříd a rozhraní odvozených od `Map` je ve zcela stejné strategii, jako u množiny



- třída `HashMap` je (opět) nejčastěji používaná „mapová“ třída
- v `TreeMap` jsou jednotlivé prvky seřazeny podle hodnoty klíče

- `TreeMap` se používá (stejně jako `TreeSet`) méně – když potřebujeme mít prvky seřazené, nejčastěji z důvodů získání „podmapy“

- rozhraní `Map` umožňuje použít několika typů metod

1. metody známé již ze seznamů a množin

- `int size()` – vrací počet aktuálních prvků
- `void clear()` – zruší všechny prvky
- `boolean isEmpty()` – test na prázdnotu

2. vkládání a odstraňování prvků – prvkem se míní nedělitelná dvojice objektů – klíč (*key*) a hodnota (v metodách značená buď *value* nebo *entry*)

- `V put(K key, V value)` – vložení prvku
- `void putAll(Map<? extends K, ? extends V> m)` – vložení všech prvků z jiné mapy (mělká kopie)
- `V remove(K key)` – odstranění prvku podle hodnoty klíče

3. zjištění, zda je v množině buď objekt klíče nebo hodnoty

- `boolean containsKey(K key)` – obsahuje klíč
- `boolean containsValue(V value)` – obsahuje hodnotu
- `V get(K key)` – podle klíče vrátí hodnotu – nepoužívanější operace

4. z mapy vytvoří množinu nebo seznam – musíme vybrat, zda to budou klíče či hodnoty (typické použití pro iterátory)

- `Collection<V> values()` – vrací hodnoty jako `Collection` (ne `List` nebo `Set`)
- `Set<K> keySet()` – vrací množinu klíčů
- `Set<Map.Entry<K,V>> entrySet()` – vrací množinu dvojic, prvky množiny jsou typu `Map.Entry`

```

HashMap<String, Integer> hm = new HashMap<String, Integer>();
hm.put("první", 1);
hm.put("druhy", 2);
hm.put("třetí", 3);

```

```

Set<Map.Entry<String, Integer>> s = hm.entrySet();
for (Map.Entry<String, Integer> me: s) {
    System.out.print(me.getKey() + "=" + me.getValue() + ", ");
}

```

Vypíše:

```
druhy=2, první=1, třetí=3,
```

- u množiny hodnot (získaných pomocí `values()`) získáváme mělké kopie – změna hodnoty prvku se projeví i v originální mapě

Výstraha

Nelze měnit objekt hodnoty, lze měnit jen nastavení objektu představujícího hodnotu. To např. znamená, že pokud budeme mít mapu, ve které bude klíčem `String` se jménem člověka a hodnotou `Integer` s jeho váhou, nelze při ztloustnutí nahradit tento `Integer` novým při zachování původního klíče (jména). Objektu typu `Integer` nelze měnit jeho hodnotu.

- chceme-li měnit hodnoty, máme tři základní možnosti:

1. zrušit celý prvek (jméno i váhu) a vložit nový (stejně jméno, jiná váha)
2. vložit nový prvek se stejným klíčem (stejně jméno, jiná váha)
3. zavést vlastní třídu `Vaha`, jejíž datový prvek lze měnit

3.6.1. Třída `TreeMap`

- používá se tehdy, potřebujeme-li mít klíče v mapě seřazené

- to není nutné z důvodů vyhledávání nějakého klíče – to stejně dobře (tj. rychle) poslouží i třída `HashMap`

- seřazení klíčů je nezbytné v tom případě, kdy potřebujeme získat z mapy:

- největší či nejmenší klíč
- „podmapu“ v závislosti na hodnotě klíče

- `TreeMap` obsahuje všechny metody ze třídy `HashMap` a přidává k nim ještě metody:

- `K firstKey()` – vrací nejmenší (první v pořadí) klíč
- `K lastKey()` – vrací největší (poslední v pořadí) klíč

- komparátor používaný v konkrétní `TreeMap` se dá zjistit metodou `Comparator<K> comparator()`

- vrací buď objekt použitého komparátoru nebo `null` (přirozené řazení)
- není vhodné měnit očekávané vzestupné řazení na sestupné

- určitý komparátor se zadá použitím přetíženého konstruktoru `TreeMap(Comparator<K> comp)`

- použitím jen `TreeMap()`, bude použito přirozené řazení
 - ♦ je-li použito přirozené řazení, musí objekt klíče implementovat rozhraní `Comparable<K>`

- metody z `TreeMap` vracející „podmapu“ vždy jako mělkou kopii

- `SortedMap<K,V> headMap(K doKlice)` – podmapa, kde klíče jsou ostře menší než daný klíč
- `SortedMap<K,V> tailMap(K odKliceVcetne)` – podmapa, kde klíče jsou větší nebo rovny danému klíči

- `SortedMap<K,V> subMap(K odKliceVcetne, K doKlice)` – podmapa "z prostředka" stávající mapy

Příklad 3.7. Nastavení default a uživatelských hodnot

Nastavení default a uživatelských hodnot – využívá toho, že vložení stejného klíče do mapy překryje původní hodnotu

```
public class NastaveniVMape {
    private static String[] key =
        {"pozadi", "popredi", "ramecek"};
    private static String[] hodDef =
        {"bila", "cerna", "modra"};
    private static String[] hodUziv =
        {null, "modra", "cervena"};

    static Map<String, String> options(String[] hodnoty) {
        Map<String, String> m = new HashMap<String, String>();
        for (int i = 0; i < key.length; i++) {
            if (hodnoty[i] != null)
                m.put(key[i], hodnoty[i]);
        }
        return m;
    }

    public static void main(String args[]) {
        Map<String, String> defaultNastaveni = options(hodDef);
        Map<String, String> uzivatelNastaveni = options(hodUziv);
        HashMap<String, String> platneNastaveni =
            new HashMap<String, String>(defaultNastaveni);

        platneNastaveni.putAll(uzivatelNastaveni);
        System.out.println("Default:  " + defaultNastaveni);
        System.out.println("Uzivatel:  " + uzivatelNastaveni);
        System.out.println("Platne:    " + platneNastaveni);
    }
}
```

Vypiše:

```
Default:  {popredi=cerna, ramecek=modra, pozadi=bila}
Uzivatel: {popredi=modra, ramecek=cervena}
Platne:   {popredi=modra, ramecek=cervena, pozadi=bila}
```


Příklad 3.8. Zjištění frekvence slov

```
public class FrekvenceSlovPomociMapy {
    public static void main(String args[]) {
        String[] s = {"lesni", "vily", "vence", "vily", "a",
                     "psi", "z", "vily", "na", "ne", "vyli"};
        Map<String, Integer> m = new TreeMap<String, Integer>();
        Integer c;
        for (int i = 0; i < s.length; i++) {
            int cet = 0;
            if ((c = m.get(s[i])) != null)
                cet = c.intValue();
            m.put(s[i], cet + 1);
        }

        System.out.println("Nalezeno " + m.size() +
                           " rozdilnych slov");
        System.out.println(m);
    }
}
```

Vypíše:

```
Nalezeno 9 rozdilnych slov
{a=1, lesni=1, na=1, ne=1, psi=1, vence=1, vily=3, vyli=1, z=1}
```

3.7. Problémy objektů v hešovacích třídách

■ hešování již implementováno v HashSet i HashMap

- jedinou nutností je zajistit, aby objekt třídy vracel správný hešovací kód

■ první problém – hešovací kód vypočítává metoda hashCode(), kterou každý objekt dědí od třídy Object

- zdánlivě je vše v naprostém pořádku a nemusíme učinit naprosto nic a program bude přeložen správně
- správně fungovat bude ale jen pro objekty všech obalovacích tříd primitivních datových typů (Integer, Double atp.) a třídy String
 - ♦ ty jsou pro svojí jednoduchost často používány jako ukázka – problémy při přechodu na reálnou aplikaci

■ druhý problém – hešovací kód nemusí stačit, když jsou stejné hodnoty

■ ve skutečnosti musíme překrýt metody hashCode() a equals()

3.7.1. Metoda equals()

Pět pravidel obecnému kontraktu:

1. objekt se musí vždy rovnat sám sobě – reflexivnost

`x.equals(x)` je vždy `true`

těžko se poruší

2. objekty se musí rovnat křížem – symetričnost

`y.equals(x) == x.equals(y)`

porušit už lze

3. rovná-li se jeden objekt druhému a druhý třetímu, musí se rovnat i první třetímu – tranzitivita

jestliže `x.equals(y) == true` a `y.equals(z) == true` musí `x.equals(z) == true`

je reálné toto pravidlo porušit

4. jsou-li si dva objekty rovné, musí si být rovné tak dlouho, dokud u některého z nich nenastane změna – konzistentnost

upozorňuje na problém měnitelných objektů

5. žádný objekt se nesmí rovnat `null`

`x.equals(null) == false`

pravidlo v sobě zahrnuje i nepřípustnost vyvolání výjimky `NullPointerException` – místo reference na porovnávaný objekt byla metodě `equals()` předána hodnota `null`

- při porovnávání primitivních hodnot typu `float` nebo `double` je vhodné tyto hodnoty převést na celočíselný typ pomocí metod obalovacích tříd `Float.floatToIntBits()` nebo `Double.doubleToLongBits()`

- pak porovnáváme pomocí `==` typy `long` nebo `int`

- vyhneme se případným problémům s okrajovými hodnotami typu `Float.NaN` apod., nepřesnému porovnávání „velmi podobných“ čísel atd.

Příklad 3.9.

Například komparátor podle `double` atributu `vaha` by měl nejlépe vypadat takto:

```
class KomparatorOsobyPodleVahy implements
    Comparator<Osoba> {
    public int compare(Osoba o1, Osoba o2) {
        double v1 = o1.vaha;
        double v2 = o2.vaha;
        long lv1 = Double.doubleToLongBits(v1);
        long lv2 = Double.doubleToLongBits(v2);
        if (lv1 == lv2)
            return 0;
        if (v1 > v2)
            return +1;
        else
            return -1;
    }
}
```

3.7.2. Metoda `hashCode()`

Tři pravidla obecného kontraktu:

1. pro tentýž objekt musí `hashCode()` vracet vždy stejný `int`
2. rozhodla-li `equals()`, že jsou si dva objekty rovny, musí `hashCode()` vrátit stejný `int`
3. nejsou-li si objekty rovny podle `equals()`, mohou mít stejný hešovací kód – nevhodná implementace `hashCode()` – významně snižuje efektivitu programu

3.7.3. Efektivní `hashCode()`

- nestejně hešovací kódy pro nestejně objekty
- měly by mít navíc rovnoměrné rozložení

Návod:

- a. `int` pomocná proměnná `vysledek` inicializovaná číslem 17

```
int vysledek = 17;
```

- b. pro každou významnou stavovou proměnnou objektu, která byla použita pro porovnávání v metodě `equals()` vypočteme vlastní hešovací kód a uložíme jej do pomocné proměnné `pom`

Výpočet v závislosti na typu stavové proměnné:

- `boolean pom = sp ? 0 : 1;`
- `byte, char, short, int pom = (int) sp;`
- `long pom = (int) (sp ^ (sp >>> 32));`

- `float pom = Float.floatToIntBits(sp);`
- `double long l = Double.doubleToLongBits(sp);`
`pom = (int) (l ^ (l >>> 32));`
- odkaz na objekt
`pom = (sp == null) ? 0 : sp.hashCode();`
- pro pole vypočteme `pom` postupně pro každý prvek pole

Po vypočtení `pom` touto hodnotou ovlivníme proměnnou `vysledek`

```
vysledek = 37 * vysledek + pom;
```

a pokračujeme s další stavovou proměnnou.

- c. po ovlivnění `vysledek` všemi významnými stavovými proměnnými objektu je vrácen
- d. zkontrolujeme na příkladech, zda stejné objekty (z pohledu `equals()`) vracejí stejné hešovací kódy – pokud ne, je třeba zjistit proč a chybu opravit

Příklad 3.10. Ukázka použití různých přístupů k hešování

Je použita třída `Osoba`, která má základní atributy typu `boolean`, `int`, `double` a `String`. Tato třída má připravenou metodu `equals()` přesně podle doporučení, ale nemá překrytu metodu `hashCode()`.

Od třídy `Osoba` jsou odvozeny tři další třídy, které teprve metodu `hashCode()` překrývají. Všechny tři jsou funkční a pokud je použijeme pro malé objemy dat (řádu tisíců), nepoznáme při běhu programu výkonnostní rozdíl.

Testovací program vždy připraví pole objektů zvolené třídy. Pak (v měřené části) uloží všechny objekty z tohoto pole do `HashSet`. V následujícím měřeném úseku postupně v množině všechny prvky vyhledá.

Třída `NevhodnaOsoba` vrací jako hešovací kód pouze atribut `vyska`. Protože však výška může být pouze v rozsahu 170 až 200, je k dispozici pouze 31 různých hešovacích kódů. To způsobí výrazný nárůst doby běhu programu na počtu vložených prvků. Je to z toho důvodu, že jednak skutečný rozdíl mezi prvky musí rozpoznat až metoda `equals()`, ale zejména proto, že hešovací tabulka se změnila na 31 lineárně zřetězených seznamů. (= implementační detail)

Mnohem lepší je třída `PrijatelnaOsoba`, kde jednoduchou úpravou, která představuje pouze vynásobení číselných atributů třídy (`vyska * vaha`), získáme znatelný nárůst výkonnosti.

Třída `PerfektniOsoba` má metodu `hashCode()` připravenou přesně podle návodu. Při jejím použití zjistíme, že potřebný čas (zejména pro vyhledávání) narůstá s počtem prvků velmi málo.

V příkladu je použita navíc speciální třída `NemennaPerfektniOsoba`. Ta vychází z předpokladu, že hodnoty atributů se nebudou měnit a je tedy možné hešovací kód vypočítat jednou provždy v konstruktoru. Tato konstantní hodnota je pak vrácena překrytou metodou `hashCode()`. Je třeba zdůraznit, že se nemění princip výpočtu hešovacího kódu – je stále „perfektní“.

```
import java.util.*;

class Osoba {
    // zakladni stavove atributy
    protected boolean muz;
    protected int vyska;
    protected double vaha;
    protected String jmeno;
    // bitovy obraz vahy pro hashCode() a equals()
    protected long longVaha; // odvozeny atribut
    private static Random r = new Random();

    Osoba() {
        this.muz = r.nextBoolean();
        this.vyska = 170 + r.nextInt(31); // <170; 200>
        this.vaha = 50 + 50 * r.nextDouble(); // <50; 100>
        this.longVaha = Double.doubleToLongBits(this.vaha);
        byte[] b = new byte[5];
        for (int i = 0; i < 5; i++)
            b[i] = (byte) ((r.nextInt(26) + (byte) 'a'));
        jmeno = new String(b);
    }

    public String toString() {
        return jmeno + ", " + (muz ? "muz " : "zena") + ", " + vyska + ", " + vaha;
    }
}
```

```
}

public boolean equals(Object o) {
    if (o == this)
        return true;
    if (o instanceof Osoba == false)
        return false;
    Osoba os = (Osoba) o;
    boolean bMuz = this.muz == os.muz;
    boolean bVyska = this.vyska == os.vyska;
    boolean bVaha = this.longVaha == os.longVaha;
    boolean bJmeno = this.jmeno.equals(os.jmeno);
    return bMuz && bVyska && bVaha && bJmeno;
}

class NevhodnaOsoba extends Osoba {
    public int hashCode() {
        return vyska;
    }
}

class PrijatelnaOsoba extends Osoba {
    public int hashCode() {
        return (int) (vyska * vaha);
    }
}

class PerfektniOsoba extends Osoba {
    public int hashCode() {
        int vysledek = 17;
        int pom;
        pom = this.muz ? 0 : 1;
        vysledek = 37 * vysledek + pom;
        pom = this.vyska;
        vysledek = 37 * vysledek + pom;
        long l = Double.doubleToLongBits(this.vaha);
        pom = (int) (l ^ (l >>> 32));
        vysledek = 37 * vysledek + pom;
        pom = this.jmeno.hashCode();
        vysledek = 37 * vysledek + pom;
        return vysledek;
    }
}

class NemennaPerfektniOsoba extends PerfektniOsoba {
    protected int hashKod;
    NemennaPerfektniOsoba() {
        super();
        hashKod = super.hashCode();
    }

    public int hashCode() {
        return hashKod;
    }
}
```

```

    return hashKod;
}
}

public class TypyHashCode {
    static int pocet;

    public static void main(String[] args) {
        if (args[0] != null)
            pocet = Integer.parseInt(args[0]);

        Osoba[] pole = new Osoba[pocet];

        for (int i = 0; i < pocet; i++) {
            // pole[i] = new NevhodnaOsoba();
            // pole[i] = new PrijatelnaOsoba();
            pole[i] = new PerfektniOsoba();
            // pole[i] = new NemennaPerfektniOsoba();
        }

        System.out.println(pole[0].getClass().getName());
        long zac = System.nanoTime();
        HashSet<Osoba> mnOsob = new HashSet<Osoba>(pocet);
        for (int i = 0; i < pocet; i++) {
            mnOsob.add(pole[i]);
        }
        long kon = System.nanoTime();
        System.out.print("Vlozeni: " + mnOsob.size() + " (" + pocet + ") ");
        System.out.println("cas = " + (kon - zac) / 1000000);

        zac = System.nanoTime();
        int n = 0;
        for (int i = pocet - 1; i >= 0; i--)
            if (mnOsob.contains(pole[i]) == true)
                n++;

        kon = System.nanoTime();
        System.out.print("Pristup: "+n+" (" + pocet + ") ");
        System.out.println("cas = " + (kon - zac) / 1000000);
    }
}

```

	prvků	1000	10000	20000	30000
NevhodnaOsoba	vložení	20	1392	7050	17185
	přístup	10	1351	7000	16904

	prvků	1000	10000	100000	500000
PrijatelnaOsoba	vložení	10	51	1542	25066
	přístup	10	20	771	18106
PerfektniOsoba	vložení	10	40	961	8051
	přístup	10	20	130	651
NemennaPerfektniOsoba	vložení	10	40	871	7761
	přístup	0	10	80	411

- Hodnoty pro 1000 prvků v kolekci, kdy je čas přístupu k objektům libovolné z uvedených čtyř tříd prakticky neměřitelný, jsou důvodem, proč je správné přípravě metody `hashCode()` obecně věnována tak malá pozornost.
- Na příkladu `NemennaPerfektniOsoba` je vidět, že pokud nechceme měnit stav vkládaného objektu, můžeme zvýšit rychlost až o jednu třetinu.
- Na porovnání `NemennaPerfektniOsoba`, `PrijatelnaOsoba` a `PerfektniOsoba` je dobře vidět, že není nutné mít obavy z přehnané časové náročnosti výpočtu „perfektního“ hešovacího kódu. U `PerfektniOsoba` je sice výpočet zpomalen o třetinu (oproti `NemennaPerfektniOsoba`), ale zůstává stále velmi malý. To se u `PrijatelnaOsoba` (s jednodušším výpočtem) zdaleka nedá říci (čas narůstá nelineárně).

■ jak generované hešovací kódy splňují podmínku unikátnosti pro 100 tisíc různých objektů:

- `NevhodnaOsoba` – 31 rozdílných
- `PrijatelnaOsoba` – 11147 rozdílných a 88853 shodných
- `PerfektniOsoba` – 99998 rozdílných a pouze 2 shodné

3.8. Ideální třída pro vkládání do libovolné kolekce

Píšeme-li třídu, která bude používána v kolekcích, můžeme významně ulehčit život tomu, kdo ji bude používat, respektování těchto zásad:

- `toString()` – rozumná informace o stavu objektu – naprostá samozřejmost
- `equals()` – testuje nejen rovnost objektů, ale i to, zda jsou stejného typu nebo jeden z nich není `null`
- `hashCode()` – napsaná podle „perfektního“ vzoru
- `compareTo()` – pro přirozené řazení třídy podle nejpřirozenějšího kritéria
- statické komparátory (absolutní řazení) pro všechny další použitelné řadící kritéria
- v případě práce s akcentovanými znaky je explicitně připraven národní `Collator`
- jakékoliv porovnávání na rovnost u reálných čísel je pomocí odpovídajících celých čísel

- hodnoty všech atributů kontrolovat v konstruktoru a vyhazovat asynchronní výjimky `NullPointerException` nebo `IllegalArgumentException`
- zvážit, zda neudělat objekt třídy kompletně neměnný (*immutable*), což je výhodné pro složitější práci v množinách a seznamech

Výstraha

- všechno vyjmenované jsou vlastně „pomocné“ věci
- kromě nich je třeba zajistit i vlastní funkčnost třídy

Příklad 3.11.

Ukázka bude na třídě `OsobaProKolekce`, používající dva neměnné atributy – české jméno a příjmení, a dva měnitelné atributy – výšku a váhu.

```
public class OsobaProKolekce implements
Comparable<OsobaProKolekce> {
    // zakladni stavove atributy
    // nemenitelne
    private final String krestni, prijmeni;
    // menitelne
    private int vyska;
    private double vaha;
    // odvozeny atribut - bitovy obraz vaha pro hashCode() a equals()
    protected long longVaha;
    // pomocny atribut pro ceske razeni
    private static final Collator COL =
        Collator.getInstance(new Locale("cs", "CZ"));

    public OsobaProKolekce(String krestni, String prijmeni,
                           int vyska, double vaha) {
        if (krestni == null || prijmeni == null)
            throw new NullPointerException();
        this.krestni = krestni;
        this.prijmeni = prijmeni;
        setVyska(vyska);
        setVaha(vaha);
    }

    public String getKrestni() {
        return krestni;
    }

    public String getPrijmeni() {
        return prijmeni;
    }

    public void setVyska(int vyska) {
        if (vyska <= 0)
            throw new IllegalArgumentException("vyska=" + vyska);
        this.vyska = vyska;
    }

    public void setVaha(double vaha) {
        if (vaha <= 0)
            throw new IllegalArgumentException("vaha=" + vaha);
        this.vaha = vaha;
        this.longVaha = Double.doubleToLongBits(this.vaha);
    }

    public int getVyska() {
        return vyska;
    }
}
```

```

public double getVaha() {
    return vaha;
}

public String toString() {
    return prijmeni + " " + krestni + ", " + vyska + ", " + vaha + "\n";
}

public boolean equals(Object o) {
    if (o == this)
        return true;
    if ((o instanceof OsobaProKolekce) == false)
        return false;
    OsobaProKolekce opk = (OsobaProKolekce) o;
    boolean stejneKrestni = krestni.equals(opk.krestni);
    boolean stejnePrijmeni = prijmeni.equals(opk.prijmeni);
    boolean stejnaVyska = vyska == opk.vyska;
    boolean stejnaVaha = longVaha == opk.longVaha;

    return stejneKrestni && stejnePrijmeni
        && stejnaVyska && stejnaVaha;
}

public int hashCode() {
    int vysledek = 17;
    vysledek = 37 * vysledek + krestni.hashCode();
    vysledek = 37 * vysledek + prijmeni.hashCode();
    vysledek = 37 * vysledek + vyska;
    int pom = (int) (longVaha ^ (longVaha >>> 32));
    vysledek = 37 * vysledek + pom;
    return vysledek;
}

// prirodzene razeni
public int compareTo(OsobaProKolekce opk) {
    int tmpP = COL.compare(this.prijmeni, opk.prijmeni);
    int tmpK = COL.compare(this.krestni, opk.krestni);
    return (tmpP == 0 ? tmpK : tmpP);
}

// komparatory pro absolutni razeni
public static final Comparator<OsobaProKolekce> PODLE_VYSKY =
    new Comparator<OsobaProKolekce>() {
        public int compare(OsobaProKolekce o1, OsobaProKolekce o2) {
            return o1.vyska - o2.vyska;
        }
    };

public static final Comparator<OsobaProKolekce> PODLE_VAHY =
    new Comparator<OsobaProKolekce>() {
        public int compare(OsobaProKolekce o1, OsobaProKolekce o2) {
            double v1 = o1.vaha;
            double v2 = o2.vaha;

```

```

        long lv1 = o1.longVaha;
        long lv2 = o2.longVaha;
        if (lv1 == lv2)
            return 0;
        if (v1 > v2)
            return +1;
        else
            return -1;
    }
};
}

```

Kapitola 4. Java a XML, SAX

4.1. Java a XML

- literatura – Brett McLaughlin: Java & XML, O'Reilly, 2001
- XML se používá téměř všude
 - je to jasně definovaný formát
 - ♦ existuje množství programových podpor pro jeho zpracování
- jaké akce při zpracování jsou potřeba:
 - načítání
 - manipulace s načtenými daty
 - generování nových elementů nebo oprava stávajících
 - zápis do XML dokumentu
 - transformace do jiných formátů
- nejčastější je načítání a manipulace s načtenými daty
 - nejhorší – napsat vlastní program
 - optimální – použít již hotový parser

4.1.1. Obecné vlastnosti parseru

- čte XML ze souboru (nebo obecně ze vstupního proudu)
 - provádí nízkourovňovou analýzu XML
 - provádí kontrolu správné strukturovanosti (*well formed*)
 - ♦ může provádět validaci podle DTD nebo XSD
 - extrahuje názvy elementů a atributů a jejich hodnoty
 - zpracovává všechny další pomocné informace z XML souboru
 - ♦ komentáře
 - ♦ instrukce pro zpracování
 - ♦ entity
 - ♦ CDATA sekce atd.
- přes programátorské rozhraní (API) nabízí abstraktní model XML dokumentu – infoset

- infoset je parsovaný strom XML, kde se pracuje najednou s jednotlivými částmi, tj. elementy, atributy, textovým obsahem elementů

Výstraha

Tento princip je obecný a nezáleží na použitém programovacím jazyce.

- způsobů zpracování XML dokumentu je velké množství
 - u každého způsobu je popsáno jeho rozhraní (API), kterému pak vyhovuje určitý parser
- základní dělení rozhraní
 - proudové čtení
 - práce se stromovou reprezentací dokumentu

4.1.1.1. Proudové čtení

- základní představitel SAX (*Simple API for XML*)
- též „událostmi řízené zpracování“
- princip
 - parser postupně čte XML dokument a pro každou ucelenou část vyvolá událost
 - ♦ naším úkolem je napsat obsluhy těchto událostí
- výhody
 - velká rychlost a malá paměťová náročnost
 - obecně známé a všude podporované API – nyní SAX 2
 - součástí Java Core API 1.5
- nevýhody
 - sekvenční průchod – nelze se vracet
 - zpracování velmi nízkourovňové („přes ruku“) s množstvím pomocných proměnných nebo vlastní nadstavbou
 - prakticky jen pro čtení

4.1.1.2. Práce se stromovou reprezentací dokumentu

- základní představitel DOM (*Document Object Model*) od W3C
 - celý dokument se načte najednou do stromu v paměti (XML má stromovou strukturu)
 - ♦ kterýkoli prvek XML je přístupný pomocí objektů
- výhody

- obecně známé a všude podporované API
- součástí Java Core API 1.5
- celý info set je dostupný v paměti
- pro čtení i zápis – možnost změn, generování a uložení zpět do XML dokumentu
- spolupracuje s dotazovacím jazykem pro XML XPath (*XML Path Language*)

■ nevýhody

- malá rychlost a velká paměťová náročnost
- použitelné do řádu cca desítek MB velikosti XML dokumentu
- hodně obecné a tím i pro některá použití zbytečně složité

■ kromě těchto dvou základních se objevují další rozhraní zjednodušující práci

- základní nevýhoda
 - ♦ nejsou součástí Java Core API 1.5 – nutno doinstalovat další knihovny

4.1.2. Rozhraní parserů pro Javu

4.1.2.1. JAXP – *Java API for XML Processing*

■ důležité rozhraní Java Core API

- neobsahuje nic nového, má pouze zastřešující metody
 - ♦ je tedy možné používat jednotným způsobem různé parsery, jak pro SAX, tak i pro DOM (např. pro DOM jsou to Xerces, Crimson)
 - je možné parsery měnit a od jejich drobných odlišností jsme odstínění

■ JAXP je součástí Java Core API 1.5

- viz balíky `javax.xml` a zejména `javax.xml.parsers`

■ informace <http://java.sun.com/webservices/jaxp/>

4.1.2.2. JDOM

■ speciální rozhraní jen pro Javu

- snaha o zjednodušení příliš obecného DOM – jednodušší práce s běžnými dokumenty

■ informace <http://www.jdom.org>

■ zatím (2006) není součástí Java Core API ani součástí *Java Web Services Developer Pack* (JWS DP 2.0)

[Unfortunately, due to the lack of JDOM's functionality to write a single element to SAX events, at this moment JDOM is not supported.]

4.1.3. Základní dělení parserů

1. podle způsobu zpracování XML dokumentu

Proudové čtení

■ *push parsers* – SAX

- čtení XML probíhá automaticky, generuje události, na které reagujeme

■ *pull parsers* – StAX

- čtení XML probíhá na naši žádost po částech (události generujeme my)

Práce se stromovou reprezentací dokumentu

■ DOM

- v současné době verze Level 3

■ JAXB

- speciální případ – *data binding*
- z XSD (nebo někdy z DTD) se generují Java třídy
 - ♦ v paměti pak není info set ale strom konkrétních objektů

2. podle možností validace

(základní validaci – *well formed* – provádějí všechny)

■ nevalidující

■ validující

- kontroluje proti DTD nebo XSD
- principiálně pomalejší
 - ♦ je vhodné provést validaci před zpracováním externě a pak vypnout validaci (nebo použít nevalidující parser)

■ kromě běžně očekávaných funkcí parseru, jako jsou:

- kontrola správné strukturovanosti
- validace
- příprava info setu

■ mají parsery ještě zabudován „manažer entit“

- ten má význam, pokud se XML dokument skládá z více XML souborů
 - ♦ manažer pak z těchto souborů XML dokument složí

4.1.4. Přehled parserů

- každé rozhraní musí být nakonec implementováno pomocí konkrétního parseru

- parserů je velké množství

- ◆ srovnání viz na

www.xmlsoftware.com/parsers.html

- JAXP od JDK 1.4 využívá Xerces, který není nutné instalovat zvlášť

- existuje ve verzích pro Javu a pro C++

- Xerces umožňuje použít

- DOM, SAX, jmenné prostory, validaci podle XSD i DTD, ...

4.2. SAX – Simple API for XML

4.2.1. Úvodní informace

- proudové zpracování XML dokumentu typu *push parser*

- SAX je tvořeno několika rozhraními v Java Core API popsány v balíku `org.xml.sax`

- prázdné implementace (adaptéry) jsou v balíku `org.xml.sax.helpers`

- používaný parser je v balíku `javax.xml.parsers` ale „odstíněný“ přes JAXP

4.2.2. Základní postup při zpracování

Program postupně vytvoří objekt parseru umožňující dle API SAX2 přečíst soubor `jidlo.xml`

- program nevykonává žádnou jinou činnost

- dá se použít jako verifikátor (*well formed*) – nevypíše-li chybu, je XML soubor v pořádku

- postup vytváření objektů se bude v dalších příkladech téměř stejně opakovat

4.2.2.1. Hlavní program

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import javax.xml.parsers.*;

public class VerifikatorJidlo {
    private static final String SOUBOR = "jidlo.xml";

    public static void main(String[] args) {
        try {
            SAXParserFactory spf = SAXParserFactory.newInstance();
            spf.setValidating(false);
```

```
SAXParser sp = spf.newSAXParser();
XMLReader parser = sp.getXMLReader();
parser.setErrorHandler(new ChybyZjisteneParserem());
parser.setContentHandler(new DefaultHandler());
parser.parse(SOUBOR);
System.out.println(SOUBOR + " precten bez chyb");
}
catch (Exception e) {
    e.printStackTrace();
}
}
```

Význam jednotlivých programových sekvencí

```
SAXParserFactory spf = SAXParserFactory.newInstance();
```

- nejdříve se vytvoří „obálka“ pro univerzální parser

- využívá se JAXP

- umožňuje konfiguraci z vnějších souborů atd., např. změnu defaultního parseru

- prakticky se ale využívá jen možnost nastavení validace podle XSD (nebo DTD) a zpracování jmenných prostorů (viz dále)

```
spf.setValidating(false);
```

- validace se nebude provádět

```
SAXParser sp = spf.newSAXParser();
```

- z příkazu není zřejmé, jaký skutečný parser se použije (ve skutečnosti je to Xerces)

- SAXParser splňuje požadavky SAX verze 1 (SAX1)

```
XMLReader parser = sp.getXMLReader();
```

- `org.xml.sax.XMLReader` je nadstavba nad parserem splňující požadavky SAX2 (SAX verze 2)

- umožňuje mimo jiné definovat vlastní obsluhy dle SAX2

```
parser.setErrorHandler(new ChybyZjisteneParserem());
```

- nastavení reakce na chyby

- není nutné, pokud je XML soubor v pořádku (validován před tím)

- je to ale vhodná akce

- třída `ChybyZjisteneParserem` (viz dále) se bude beze změny opakovat ve všech dalších programech

```
parser.setContentHandler(new DefaultHandler());
```

- nastavení obsluh pro čtení XML dat

- toto nastavení nedělá nic – `DefaultHandler` je prázdný adaptér

- pro skutečné zpracování XML dokumentu se vytvoří naše obslužná třída jako potomek `DefaultHandler`

4.2.2.2. Zpracování výjimek v hlavním programu

- při zpracování XML dokumentu mohou potenciálně vzniknout tři typy výjimek
- obecně je není nutné rozlišovat a používáme jednoduchou reakci

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

- v případě problémů je pro jejich lepší lokalizaci možné použít hierarchii výjimek

```
try {
    dříve uvedený kód
}
catch (SAXException e) {
    výjimky vzniklé při parsování XML dokumentu
}
catch (ParserConfigurationException e) {
    výjimky vzniklé při nastavování vlastností parseru - viz dále
}
catch (IOException e) {
    výjimky vzniklé při čtení XML dokumentu ze souboru nebo z proudu
}
```

4.2.2.3. Pomocná třída pro zpracování chyb

třída `ChybyZjisteneParserem` se bude beze změny opakovat ve všech dalších programech

```
import org.xml.sax.*;

public class ChybyZjisteneParserem
implements ErrorHandler {
    // zformatování textu hlášení
    private String textHlaseni(SAXParseException e) {
        return e.getSystemId() + "\n"
            + "radka: " + e.getLineNumber()
            + " sloupec: " + e.getColumnNumber()
            + "\n" + e.getMessage();
    }

    // obsluha varovných hlášení
    public void warning(SAXParseException e) {
        System.out.println("Varovani: " + textHlaseni(e));
    }

    // obsluha chyb
    public void error(SAXParseException e) throws SAXException {
```

```
        throw new SAXException("Chyba: " + textHlaseni(e));
    }

    // obsluha fatalních chyb
    public void fatalError(SAXParseException e) throws SAXException {
        throw new SAXException("Fatalní chyba: " + textHlaseni(e));
    }
}
```

4.2.3. Zpracování parsovaného XML dokumentu

- v balíku `org.xml.sax` jsou rozhraní, které je možné implementovat našimi třídami

- objekty těchto tříd předáme parseru

- z mnoha uvedených rozhraní jsou nezbytná:

- `ErrorHandler` – reakce na chyby
- `XMLReader` – parser vyhovující SAX2

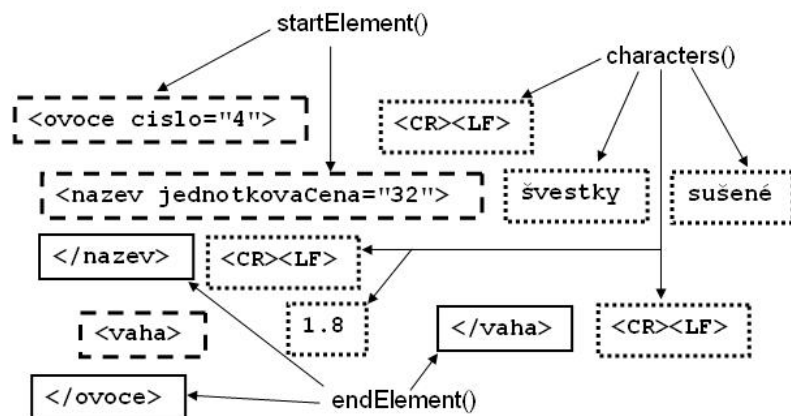
(oba viz v předchozím příkladě)

- `ContentHandler` – zpracování elementů (`DefaultHandler` z předchozího příkladu toto rozhraní implementuje)
- `Attributes` – zpracování atributů

- ve všech dalších příkladech bude zpracováván soubor `jidlo.xml` s obsahem

```
<?xml version="1.0" encoding="windows-1250"?>
<jidlo>
  <ovoce cislo="1">
    <nazev jednotkovaCena="10">jablka</nazev>
    <vaha>2.5</vaha>
  </ovoce>
  <ovoce cislo="2">
    <nazev jednotkovaCena="25">banány</nazev>
    <vaha>2</vaha>
  </ovoce>
  <ovoce cislo="3">
    <nazev jednotkovaCena="19">grapefruity</nazev>
    <vaha>0.75</vaha>
  </ovoce>
  <ovoce cislo="4">
    <nazev jednotkovaCena="32">švestky sušené</nazev>
    <vaha>1.8</vaha>
  </ovoce>
</jidlo>
```

- jak vypadá jeden element `<ovoce>` po zpracování SAX parserem



■ z rozhraní ContentHandler využíváme typicky jen metody

- startDocument() – akce na začátku dokumentu, vhodnější než konstruktor – umožňuje vícenásobné opakované čtení dokumentu
- startElement() – počáteční tag a případné atributy
- characters() – obsah elementu
- endElement() – koncový tag

Poznámka

Následující tři příklady se budou opakovat ve všech dalších technikách zpracování XML.

4.2.3.1. Výpočet celkové váhy

Program vypíše celkovou váhu nakoupeného ovoce

■ v hlavním programu se změní jen řádky:

```
VahovyHandler vh = new VahovyHandler();
parser.setContentHandler(vh);
parser.parse(SOUBOR);
System.out.println("Celkova vaha: " + vh.getCelkovaVaha());
```

■ hlavní změna bude v obslužném handleru

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class VahovyHandler extends DefaultHandler {
    private static final int VELIKOST_BUFFERU = 100;
    private static final String JMENO_ELEMENTU = "vaha";
```

```
private double celkovaVaha;
private StringBuffer hodnota = new StringBuffer(VELIKOST_BUFFERU);
private boolean uvnitrVahy;
```

```
public double getCelkovaVaha() {
    return celkovaVaha;
}
```

```
public void startDocument() {
    celkovaVaha = 0;
}
```

```
public void startElement(String uri, String localName,
    String qName, Attributes atts) {
    if (qName.equals(JMENO_ELEMENTU) == true) {
        uvnitrVahy = true;
        hodnota.setLength(0);
    }
}
```

```
public void endElement(String uri, String localName, String qName) {
    if (qName.equals(JMENO_ELEMENTU) == true) {
        uvnitrVahy = false;
        celkovaVaha += Double.parseDouble(hodnota.toString());
    }
}
```

```
public void characters(char[] ch, int start, int length) {
    if (uvnitrVahy == true) {
        hodnota.append(ch, start, length);
    }
}
```

■ význam parametrů startElement() a endElement()

- uri – URI jmenného prostoru (NS), není-li použit, pak prázdný řetězec
- localName – jméno elementu v souvislosti se jmenným prostorem, není-li NS použit, pak prázdný řetězec
- qName – úplné (kvalifikované) jméno elementu

Poznámka

po zapnutí (viz dále) `spf.setNamespaceAware(true)` ; je pro XML bez jmenných prostorů `localName` shodné s `qName`

- atts – seznam atributů (viz dále)

■ význam parametrů characters()

- `ch` – pole znaků, ve kterém jsou hodnoty všech elementů od začátku XML dokumentu
- hodnota aktuálního elementu je určena dalšími dvěma parametry
- `start` – index počátečního znaku aktuální hodnoty
- `length` – počet znaků

Výstraha

1. hodnota elementu může přijít po částech – nutné ukládat ji do `StringBufferu` a číst až v `endElement()`
2. kromě hodnot elementu obsahuje také znaky mezi elementy (typicky odřádkování), které nás nezajímají – nutné testovat, zda jsme uvnitř konkrétního elementu (po `startElement()`)

■ získání jednoho typu hodnoty z celého XML dokumentu je snadné

- pro více typů hodnot je situace složitější, ale stále přehledná

4.2.3.2. Výpočet celkové ceny

Program zpracovává atributy a vypočte celkovou cenu nákupu

```
CenovyHandler ch = new CenovyHandler();
parser.setContentHandler(ch);
parser.parse(SOUBOR);
System.out.println("Celkova cena: " + ch.getCelkovaCena());

import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class CenovyHandler extends DefaultHandler {
    private static final int VELIKOST_BUFFERU = 100;
    private double celkovaCena;
    private boolean uvnitrVahy;
    private StringBuffer hodnota = new StringBuffer(VELIKOST_BUFFERU);
    private int jednotkovaCena;
    private double vaha;

    public double getCelkovaCena() {
        return celkovaCena;
    }

    public void startDocument() {
        celkovaCena = 0;
    }

    public void startElement(String uri, String localName,
                             String qName, Attributes atts) {
        if (qName.equals("vaha") == true) {
            uvnitrVahy = true;
            hodnota.setLength(0);
        }
    }
}
```

```
else if (qName.equals("nazev") == true) {
    jednotkovaCena = Integer.parseInt(atts.getValue("jednotkovaCena"));
}

public void endElement(String uri, String localName, String qName) {
    if (qName.equals("vaha") == true) {
        uvnitrVahy = false;
        vaha = Double.parseDouble(hodnota.toString());
    }
    else if (qName.equals("ovoce") == true) {
        celkovaCena += vaha * jednotkovaCena;
    }
}

public void characters(char[] ch, int start, int length) {
    if (uvnitrVahy == true) {
        hodnota.append(ch, start, length);
    }
}
}
```

4.2.3.3. Všechny objekty v paměti

Program uloží všechny hodnoty a atributy najednou do paměti a pak je zpracovává

```
public class Ovoce {
    int cislo;
    String nazev;
    int jednotkovaCena;
    double vaha;

    public Ovoce(int cislo, String nazev,
                 int jednotkovaCena, double vaha) {
        this.cislo = cislo;
        this.nazev = nazev;
        this.jednotkovaCena = jednotkovaCena;
        this.vaha = vaha;
    }

    public String toString() {
        return "" + cislo + ". " + nazev + " - "
            + vaha + " [kg] po "
            + jednotkovaCena + " [Kc] = "
            + vaha * jednotkovaCena + " [Kc]";
    }
}

import java.util.*;

public class ZpracovaniDatVPameti {
    public static void tiskniVse(ArrayList<Ovoce> ar) {
        for (int i = 0; i < ar.size(); i++) {

```

```

        System.out.println(ar.get(i));
    }
}

public static double celkovaVaha(ArrayList<Ovoce> ar) {
    double celkovaVaha = 0;
    for (int i = 0; i < ar.size(); i++) {
        celkovaVaha += ar.get(i).vaha;
    }
    return celkovaVaha;
}

public static double celkovaCena(ArrayList<Ovoce> ar) {
    double celkovaCena = 0;
    for (int i = 0; i < ar.size(); i++) {
        Ovoce o = ar.get(i);
        celkovaCena += o.vaha * o.jednotkovaCena;
    }
    return celkovaCena;
}
}

import java.util.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class VseVPametiHandler extends DefaultHandler {
    private static final int VELIKOST_BUFFERU = 100;

    private static ArrayList<Ovoce> ar = new ArrayList<Ovoce>();
    private StringBuffer hodnota = new StringBuffer(VELIKOST_BUFFERU);
    private boolean uvnitrElementu;

    private int    cislo;
    private String nazev;
    private int    jednotkovaCena;
    private double vaha;

    public ArrayList<Ovoce> getSeznam() {
        return ar;
    }

    public void startDocument() {
        ar.clear();
    }

    public void startElement(String uri, String localName,
                             String qName, Attributes atts) {
        if (qName.equals("ovoce") == true) {
            cislo = Integer.parseInt(atts.getValue("cislo"));
        }
        else if (qName.equals("nazev") == true) {
            jednotkovaCena = Integer.parseInt(atts.getValue("jednotkovaCena"));

```

```

            hodnota.setLength(0);
            uvnitrElementu = true;
        }
        else if (qName.equals("vaha") == true) {
            hodnota.setLength(0);
            uvnitrElementu = true;
        }
    }

    public void endElement(String uri, String localName, String qName) {
        if (qName.equals("vaha") == true) {
            vaha = Double.parseDouble(hodnota.toString());
            uvnitrElementu = false;
        }
        else if (qName.equals("nazev") == true) {
            nazev = hodnota.toString();
            uvnitrElementu = false;
        }
        else if (qName.equals("ovoce") == true) {
            ar.add(new Ovoce(cislo, nazev, jednotkovaCena, vaha));
        }
    }

    public void characters(char[] ch, int start, int length) {
        if (uvnitrElementu == true) {
            hodnota.append(ch, start, length);
        }
    }
}

```

■ hlavní soubor

```

VseVPametiHandler ph = new VseVPametiHandler();
parser.setContentHandler(ph);
parser.parse(SOUBOR);
ArrayList<Ovoce> ar = ph.getSeznam();
ZpracovaniDatVPameti.tiskniVse(ar);
System.out.println("Celkova vaha = "
    + ZpracovaniDatVPameti.celkovaVaha(ar));
System.out.println("Celkova cena = "
    + ZpracovaniDatVPameti.celkovaCena(ar));

```

4.2.4. Zpracování složitějšího XML dokumentu

- pokud se v jednotlivých elementech neopakují na různých místech stejně pojmenované vnořené pod-elementy, je postup přímočarý
- v případě hodně strukturovaného dokumentu (hodně vrstev zanoření) nebo v případě velkého počtu sourozeneckých elementů není uvedený přístup příliš přehledný
- lze připravit více potomků DefaultHandler a přepínat je

- výhodou je, že každý má svůj `startElement()`, `endElement()` a `characters()`, tj. jejich kód je jednoduchý
- nevýhodou je, že je nutné vyřešit přepínání jednotlivých handlerů

4.2.5. Problematika různého kódování

- tento problém se řeší zcela automaticky, pokud má XML dokument v hlavičce uvedeno použité kódování, např.

```
<?xml version="1.0" encoding="utf-8"?>
```

- parser toto kódování zjistí a zařídí se podle něj
 - ♦ Pozor: Název kódování musí být pomocí kanonického jména – viz podrobně později. Tři nejběžnější kanonická jména:
 - utf-8
 - ISO-8859-2
 - windows-1250

- při neuvedeném nebo chybně uvedeném kódování XML dokumentu se dá problém řešit pomocí `org.xml.sax.InputSource`

- ale principiálně je to nevhodný přístup
 - ♦ je třeba zajistit, aby XML dokument byl správně označen před jeho zpracováním parserem

4.2.6. Nastavení vlastností parseru

- tuto možnost použijeme pro speciální práci s XML dokumentem

- pro naprostou většinu běžných aktivit není nutné, protože:

- parser je nakonfigurován pro běžné použití
- dvě hlavní vlastnosti – validaci a jmenné prostory – lze ovládat přímo metodami z JAXP třídy `javax.xml.parsers.SAXParserFactory`

```
setValidating(boolean validating)

setNamespaceAware(boolean awareness)
```

- pro nastavení ostatních používáme

```
void setFeature(String name, boolean value)
```

- kde `name` je jméno vlastnosti

- seznam jmen obecných vlastností (podporovaných každým parserem JAXP) lze nalézt na:

jdk1.5.0_06/docs/api/org/xml/sax/package-summary.html

Výstraha

Jména vlastností jsou ve tvaru URL, které se musí přesně opsat

- např. podpora jmenných prostorů se nastaví jako:

```
spf.setFeature("http://xml.org/sax/features/namespace", true);
```

- dává stejný výsledek jako `spf.setNamespaceAware(true);`

- kromě obecných vlastností lze pro **konkrétní** používaný parser stejným způsobem nastavit i jeho speciální vlastnosti

- tuto akci není dobré provádět bez vážného důvodu, protože pak program ztrácí na přenositelnosti (dostáváme se mimo Java Core API)

- ♦ např. pro Xerces se zapne provádění validace podle XSD schématu příkazem

```
spf.setFeature("http://apache.org/xml/features/validation/schema", true);
```

- seznam vlastností Xercesu lze nalézt na

<http://xml.apache.org/xerces2-j/features.html>

- některé vlastnosti jsou i jiného typu než „zapnout/vypnout“

- nastavují se metodou třídy `javax.xml.parsers.SAXParser`

```
void setProperty(String name, Object value)
```

- např. nastavení, že se pro validaci oproti XSD bude používat soubor `jidlo.xsd`, který není připojen do XML souboru `jidlo.xml`

```
sp.setProperty("http://java.sun.com/xml/jaxp/properties/schemaSource",
    new File("jidlo.xsd"));
```

4.2.7. Validace oproti DTD nebo XSD

- pokud budeme vytvářet složitější programy využívající XML, může se stát, že nebude možné validovat XML dokument externě

- např. XML dokumenty přicházející po síti nebo generované on-line jiným programem

- pak je možné dodat do našeho programu validaci (DTD, XSD)

Program lze využít jako externí validátor, kdy při validním XML dokumentu vypíše pouze počet jeho elementů

Použití

- pouze *well formed*

```
>java VerifikatorSAX jidlo.xml
jidlo.xml: 13 elementu 8 atributu
```

■ DTD je připojeno v souboru

```
>java VerifikatorSAX jidlo-dtd.xml dtd
jidlo-dtd.xml: 13 elementu 8 atributu
```

■ XSD je připojeno v souboru

```
>java VerifikatorSAX jidlo-xsd.xml xsd
jidlo-xsd.xml: 13 elementu 10 atributu
```

■ XSD soubor je externě připojen

```
>java VerifikatorSAX jidlo.xml xsd jidlo.xsd
jidlo.xml: 13 elementu 8 atributu
```

```
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import javax.xml.parsers.*;

public class VerifikatorSAX {

    public static void main(String[] args) {
        if (args.length < 1) {
            System.out.println("Syntaxe: "
                + "java VerifikatorSAX <soubor.xml> "
                + "[dtd | xsd] [soubor.xsd]\n");
            System.exit(1);
        }

        boolean dtd = false;
        boolean xsd = false;
        if (args.length >= 2) {
            if (args[1].toLowerCase().equals("dtd") == true) {
                dtd = true;
            }
            else if (args[1].toLowerCase().equals("xsd") == true) {
                xsd = true;
            }
        }

        try {
            SAXParserFactory spf = SAXParserFactory.newInstance();
            spf.setValidating(true);
            spf.setFeature("http://xml.org/sax/features/namespace", true);
            // spf.setNamespaceAware(true); // stejná akce
            if (dtd == false && xsd == false) {
                spf.setValidating(false);
            }
            else if (xsd == true) {
                spf.setFeature("http://apache.org/xml/features/validation/schema", ▶
```

```
true);
        }

        SAXParser sp = spf.newSAXParser();
        if (xsd == true) {
            sp.setProperty("http://java.sun.com/xml/jaxp/properties/schemaLanguage",
                "http://www.w3.org/2001/XMLSchema");
            if (args.length == 3) {
                ▶
            }
            sp.setProperty("http://java.sun.com/xml/jaxp/properties/schemaSource",
                new File(args[2]));
        }

        XMLReader parser = sp.getXMLReader();
        parser.setErrorHandler(new ChybyNalezeneParserem());
        PocetElementuHandler h = new PocetElementuHandler();
        parser.setContentHandler(h);

        parser.parse(args[0]);
        System.out.println(args[0] + ": " + h.getVysledek());
    }
    catch (Exception e){
        String s = e.getMessage();
        int i = s.indexOf(ChybyNalezeneParserem.KONEC);
        if (i > 0) {
            s = s.substring(0, i);
        }
        System.out.println(s);
    }
}

class PocetElementuHandler extends DefaultHandler {
    private int pocetElementu = 0;
    private int pocetAtributu = 0;

    public void startElement(String uri, String localName,
        String qName, Attributes atts) {
        pocetElementu++;
        pocetAtributu += atts.getLength();
    }

    public String getVysledek() {
        return "" + pocetElementu + " elementu "
            + pocetAtributu + " atributu";
    }
}

class ChybyNalezeneParserem implements ErrorHandler {
    public static final String KONEC = "konecHlaseni";

    private String textHlaseni(SAXParseException e) {
```

```

        return e.getSystemId() + "\n"
            + "radka: " + e.getLineNumber()
            + " sloupec: " + e.getColumnNumber()
            + "\n" + e.getMessage() + KONEC;
    }

    public void warning(SAXParseException e) {
        System.out.println("Varovani: " + textHlaseni(e));
    }

    public void error(SAXParseException e) throws SAXException {
        throw new SAXException("Chyba: " + textHlaseni(e));
    }

    public void fatalError(SAXParseException e) throws SAXException {
        throw new SAXException("Fatalní chyba: " + textHlaseni(e));
    }
}

```

- pomocná konstanta `KONEC` je využívána proto, aby se z textu chybového hlášení daly odříznout rušivé výpisy o lokalizaci vzniklé výjimky

Poznámka

Ukázkový program na validaci si lze prohlédnout v:

`\jwsdp-1.5\jasp\samples\sax\SAXLocalNameCount.java`

4.2.8. Práce se jmennými prostory

- je velmi jednoduchá, proti předchozím způsobům se prakticky nic nemění
- podstatné je zapnout u `SAXParserFactory` zpracování jmenných prostorů příkazem

```
spf.setNamespaceAware(true);
```

- bez tohoto nastavení je i XML dokument se jmennými prostory čten jako by v něm jmenné prostory nebyly

```

<?xml version="1.0" encoding="windows-1250"?>
<potrava:jidlo xmlns:potrava="http://www.kiv.zcu.cz/~herout/xml/jidlo-sada">

    <potrava:ovoce cislo="1">
        <potrava:nazev potrava:jednotkovaCena="10">jablka</potrava:nazev>
        <potrava:vaha>2.5</potrava:vaha>
    </potrava:ovoce>

```

- jak je známo, atributy mohou, ale nemusejí mít označení jmenného prostoru (`cislo` versus `potrava:jednotkovaCena`)

```

public void startElement(String uri, String localName,
                        String qName, Attributes atts) {
    cisloStart++;

```

```

        System.out.println(" " + cisloStart
            + ". start uri=" + uri
            + ", localName=" + localName
            + ", qName=" + qName);
        for (int i = 0; i < atts.getLength(); i++) {
            System.out.println(" " + "atts uri=" + atts.getURI(i)
                + ", localName=" + atts.getLocalName(i)
                + ", qName=" + atts.getQName(i)
                + ", type=" + atts.getType(i)
                + ", value=" + atts.getValue(i));
        }
    }
}

```

- při zapnutém `spf.setNamespaceAware(true)`; vypisuje:

```

1. start uri=http://www.kiv.zcu.cz/~herout/xml/jidlo-sada,
   localName=jidlo, qName=potrava:jidlo
2. start uri=http://www.kiv.zcu.cz/~herout/xml/jidlo-sada,
   localName=ovoce, qName=potrava:ovoce
   atts uri=, localName=cislo, qName=cislo, type=CDATA, value=1
3. start uri=http://www.kiv.zcu.cz/~herout/xml/jidlo-sada,
   localName=nazev, qName=potrava:nazev
   atts uri=http://www.kiv.zcu.cz/~herout/xml/jidlo-sada,
   localName=jednotkovaCena, qName=potrava:jednotkovaCena,
   type=CDATA, value=10
1. ch=jablka, start=162, length=6
1. end   uri=http://www.kiv.zcu.cz/~herout/xml/jidlo-sada,
   localName=nazev, qName=potrava:nazev
4. start uri=http://www.kiv.zcu.cz/~herout/xml/jidlo-sada,
   localName=vaha, qName=potrava:vaha
2. ch=2.5, start=204, length=3

```

- při vypnutém `spf.setNamespaceAware(false)`; vypisuje:

```

1. start uri=, localName=, qName=potrava:jidlo
   atts uri=, localName=, qName=xmlns:potrava, type=CDATA,
   value=http://www.kiv.zcu.cz/~herout/xml/jidlo-sada
2. start uri=, localName=, qName=potrava:ovoce
   atts uri=, localName=, qName=cislo, type=CDATA, value=1
3. start uri=, localName=, qName=potrava:nazev
   atts uri=, localName=, qName=potrava:jednotkovaCena,
   type=CDATA, value=10
1. ch=jablka, start=162, length=6
1. end   uri=, localName=, qName=potrava:nazev
4. start uri=, localName=, qName=potrava:vaha
2. ch=2.5, start=204, length=3

```

- z výpisů je vidět, že `qName` vždy obsahuje přesně to, co je v XML dokumentu uvedeno

- pro většinu případů je vhodné používat pouze `qName`

- typ atributu `getType(i)` nemá ve většině případů praktický význam

Kapitola 5. DOM – *Document Object Model*

5.1. Základní informace

- standard consorcia W3C
 - v JDK 1.6 se jedná o Level 3 Core API
- parser DOM načte celý XML dokument do paměti a vytvoří tam stromovou objektovou reprezentaci
 - objektům stromové struktury se říká *nody* (uzly)
- narozdíl od SAX je DOM vhodný i pro změnu nebo vytváření nových XML dokumentů
 - má možnosti nastavování a zápisu
- zcela ve stejné filosofii jako u SAX je DOM odstíněn přes JAXP
- v `org.w3c.dom` jsou rozhraní jednotlivých nodů
- v `javax.xml.parsers` jsou třídy zajišťující vlastní parsování
 - `DocumentBuilder`
 - `DocumentBuilderFactory`
- nejvíce nás zajímají rozhraní z `org.w3c.dom`
 - je zde popsáno celkem 14 typů nodů, běžně používaných je 5
 - ♦ `Node` – základní prvek a předek s potomky:
 - `Document` – počáteční nod (ne kořenový element !)
 - `Attr` – atributy
 - `Element` – elementy
 - `Text` – hodnoty elementů
 - další jsou např.:
 - ♦ `Comment` – komentáře
 - ♦ `CDATASection` – CDATA sekce
- z těchto nodů je vytvořen objektový stromový model

Výstraha

Atributy tvoří samostatné nody, které ale nejsou potomky příslušného elementu

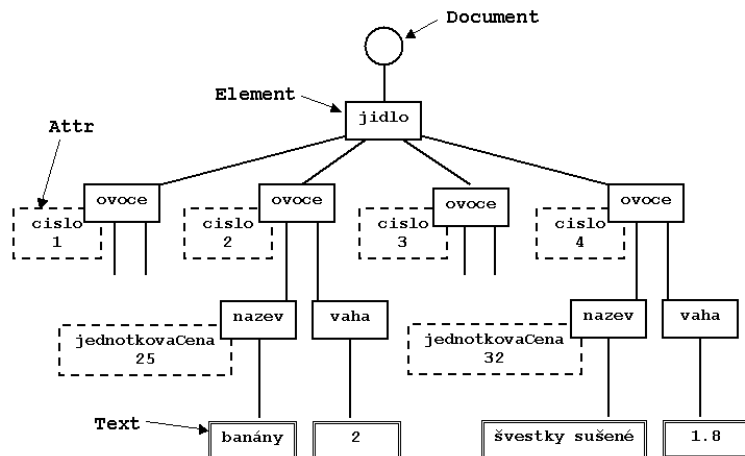
Příklad 5.1. pro jidlo.xml

```
<?xml version="1.0" encoding="windows-1250"?>
<jidlo>
  <ovoce cislo="1">
    <nazev jednotkovaCena="10">jablka</nazev>
    <vaha>2.5</vaha>
  </ovoce>

  <ovoce cislo="2">
    <nazev jednotkovaCena="25">banány</nazev>
    <vaha>2</vaha>
  </ovoce>

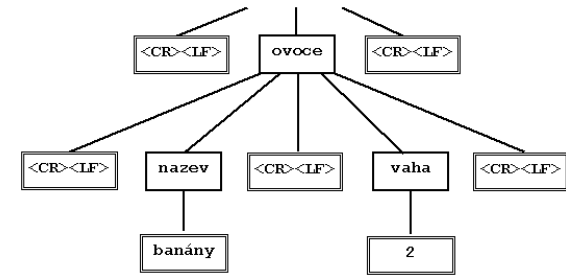
  <ovoce cislo="3">
    <nazev jednotkovaCena="19">grapefruity</nazev>
    <vaha>0.75</vaha>
  </ovoce>

  <ovoce cislo="4">
    <nazev jednotkovaCena="32">švestky sušené</nazev>
    <vaha>1.8</vaha>
  </ovoce>
</jidlo>
```



Výstraha

- ve skutečnosti (stejně jako u SAX) parser zpracovává a ukládá i konce řádků a formátovací mezery vně elementů
- ukládá je do nodu typu `Text` a je třeba s nimi při práci se stromem dokumentu počítat
- takže ve skutečnosti vypadá element `ovoce` (pro zjednodušení uveden bez atributů) takto:



- jak se při zpracování dokumentu těmto textovým nodům vyhnout viz dále v části Problém vkládaných elementů

Poznámka

- existuje i rozhraní `CharacterData`, principiálně podobné `characters()` ze SAX
 - toto rozhraní není třeba využívat
 - používá se jeho potomek `Text` (viz výše), ve kterém je již hodnota elementu
- problémy SAXu s postupným načítáním hodnoty elementu zde neexistují

Kolekce nodů

- kromě nodu `Node` a jeho potomků jsou velmi užitečné ještě kolekce uschovávající skupinu nodů
- `NodeList` – kolekce podobná `List`
 - přístup pomocí indexu, typicky je to uspořádaný seznam elementů
- `NamedNodeMap` – kolekce podobná `Map`
 - přístup pomocí jména, typicky jsou v něm uloženy jména atributů a jejich hodnoty

5.2. Základní použití DOM

Program vytvoří objekty umožňující metodou `DOM` přečíst soubor `jidlo.xml`

- program nevykonává žádnou jinou činnost
- dá se použít jako verifikátor (*well formed*) – nevypíše-li chybu, je XML soubor v pořádku
- postup vytváření objektů se bude dále téměř stejně opakovat

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;           // kvůli vyjimkam
```

```
public class VerifikatorJidloDOM {
    private static final String SOUBOR = "jidlo.xml";
```

```

public static void main(String[] args) {
    try {
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        dbf.setValidating(false);
        DocumentBuilder builder = dbf.newDocumentBuilder();
        builder.setErrorHandler(new ChybyZjisteneParserem());
        // nacteni dokumentu do pameti
        Document doc = builder.parse(SOUBOR);
        System.out.println(SOUBOR + " precten bez chyb");
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

■ vytváření objektů je díky JAXP velmi podobné SAXu

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
```

■ nejdříve se vytvoří „obálka“ pro univerzální parser

- využívá se JAXP
- umožňuje konfiguraci z vnějších souborů atd., např. změnu defaultního parseru
- prakticky se ale využívá jen možnost nastavení validace podle XSD a zpracování jmenných prostorů (viz dále)

```
dbf.setValidating(false);
```

■ validace se nebude provádět

```
DocumentBuilder builder = dbf.newDocumentBuilder();
```

■ vytvoření vlastního parseru

- není zřejmé, jaký skutečný parser se použije (ve skutečnosti je to Xerces)

```
builder.setErrorHandler(new ChybyZjisteneParserem());
```

■ nastavení reakce na chyby

- není nutné, pokud je XML soubor v pořádku (validován před tím)
- je to ale vhodná akce
 - ♦ třída `ChybyZjisteneParserem` je zcela stejná jako u SAX
 - stejné jsou i reakce na výjimky a možné rozlišení tří typů výjimek
 - proto `import org.xml.sax.*;`

Poznámka

DOM využívá vnitřně pro parsování SAX

5.3. Zpracování parsovaného XML dokumentu

- protože po parsování jsou všechna data v paměti, existuje v porovnání se SAX mnohem více možností, jak je zpracovat

Dále budou uvedeny tři programy, které mají stejnou funkčnost jako byly programy u SAX.

5.3.1. Výpočet celkové váhy

■ v hlavním programu se přidá jen řádka výpisu:

```

Document doc = builder.parse(SOUBOR);
System.out.println("Celkova vaha: " + getCelkovaVaha(doc));

```

■ dodaná metoda

```

private static double getCelkovaVaha(Document doc) {
    NodeList nl = doc.getElementsByTagName("vaha");
    double celkovaVaha = 0.0;
    for (int i = 0; i < nl.getLength(); i++) {
        Node e = nl.item(i);           // Element
        Node t = e.getFirstChild();    // Text
        String s = t.getNodeValue().trim();
        celkovaVaha += Double.parseDouble(s);
    }
    return celkovaVaha;
}

```

```
NodeList nl = doc.getElementsByTagName("vaha");
```

■ uloží do seznamu všechny elementy vaha

```
Node e = nl.item(i); Node t = e.getFirstChild();
```

■ hodnota elementu je null !!! (typická chyba)

- je třeba získat potomka, kterým je `Text` a pak teprve jeho hodnotu

```
String s = t.getNodeValue().trim();
```

■ `trim()` nás zbavuje případných okrajových bílých znaků, které by byly součástí (před a za) hodnoty elementu

```

<vaha>
  2.5
</vaha>

```

Výstraha

- je velmi nebezpečné činit implicitní předpoklady o prvním či posledním potomkovi `e.getFirstChild()`;
- v XML dokumentu mohou být kromě výše zmíněného odřádkování např. i komentáře

```
<ovoce cislo="1">
  <nazev jednotkovaCena="10">jablka</nazev>
  <vaha><!-- docela dost ->2.5</vaha>
</ovoce>
```

- bezpečný způsob získání hodnoty elementu je:

```
private static double getCelkovaVaha(Document doc) {
    NodeList nl = doc.getElementsByTagName("vaha");
    double celkovaVaha = 0.0;
    for (int i = 0; i < nl.getLength(); i++) {
        Node e = nl.item(i);           // Element
        NodeList nle = e.getChildNodes();
        String s = "";
        for (int j = 0; j < nle.getLength(); j++) {
            if (nle.item(j).getNodeType() == Node.TEXT_NODE) {
                Node t = nle.item(j);   // Text
                s += t.getNodeValue().trim();
            }
        }
        if (s.length() > 0) {
            celkovaVaha += Double.parseDouble(s);
        }
    }
    return celkovaVaha;
}
```

5.4. Metody rozhraní Node

- protože `Node` je předkem všech dalších nodů, lze používat jeho metody i v případě, že víme, že typ skutečného nodu je např. `Element`

5.4.1. Metody pro získání informace

- `short getNodeType()` – typ nodu
 - většinou se porovnává s konstantami `Node.ATTRIBUTE_NODE`, `Node.COMMENT_NODE`, `Node.ELEMENT_NODE`, `Node.TEXT_NODE` (případně s dalšími)
- `String getNodeName()` – jméno nodu, např. „vaha“

Výstraha

Pro kmenáře je jméno nodu „#comment“ a pro text je „#text“

- `String getNodeValue()` – hodnota nodu, např. „2.5“

Výstraha

Pro `Element` je hodnota `getNodeValue()` `null`

5.4.2. Metoda pro pohyb nahoru (na rodiče)

- `Node getParentNode()` – přesun na rodičovský nod

5.4.3. Metody pro horizontální pohyb (na sourozence)

- `Node getPreviousSibling()`
 - bezprostředně předchozí sourozenec
- `Node getNextSibling()`
 - bezprostředně následující sourozenec
- neexistují-li požadovaní sourozenci, pak vrací `null`

5.4.4. Metody pro pohyb dolů (na potomky)

- `boolean hasChildNodes()` – existují potomci?
- `Node getFirstChild()` – první potomek
- `Node getLastChild()` – poslední potomek
 - tyto dvě metody používat velmi opatrně a vždy otestovat, zda vrácený potomek je opravdu očekávaný potomek (viz výše problém s komentářem)
- `NodeList getChildNodes()` – list potomků

`NodeList` má dvě metody:

- `int getLength()` – počet nodů

Výstraha

nikoliv `length()` !!!

- `Node item(int index)` – nod v pořadí

5.4.5. Metody pro práci s atributy

- `boolean hasAttributes()` – existují atributy?
- `NamedNodeMap getAttributes()` – asociativní pole atributů

`NamedNodeMap` má metody:

■ `int getLength()` – počet atributů

■ `Node getItem(String name)` – vrátí atribut podle jména

■ `Node item(int index)` – vrátí atribut podle pořadí

- nepoužívat, protože pořadí atributů v XML se může libovolně měnit

Výstraha

neexistuje metoda typu `String getNamedValue(String name)` pro přímé získání hodnoty

■ typický postup je:

```
String hodnota = nm.getItem("jednotkovaCena").getNodeValue();
```

■ je-li nod přetypován na `Element`, lze použít `Attr getNode(String name)` a pak `String getValue()`

5.5. Výpočet celkové ceny

Program zpracovává atributy a vypočte celkovou cenu nákupu.

■ v hlavním programu se přidá jen řádka výpisu:

```
System.out.println("Celkova cena: " + getCelkovaCena(doc));
```

■ dodané metody

```
private static double getCelkovaCena(Document doc) {
    NodeList nl = doc.getElementsByTagName("ovoce");
    double celkovaCena = 0.0;
    for (int i = 0; i < nl.getLength(); i++) {
        Node e = nl.item(i);
        if (e.getNodeType() == Node.ELEMENT_NODE) {
            celkovaCena += getCenaOvoce(e);
        }
    }
    return celkovaCena;
}

private static int getJednotkovaCena(Node ovoce) {
    Node nazev = getNodePodleJmena(ovoce, "nazev");
    NamedNodeMap nnm = nazev.getAttributes();
    String hodnota = nnm.getItem("jednotkovaCena").getNodeValue();
    /* druhy zpusob
    Attr a = ((Element) nazev).getAttributeNode("jednotkovaCena");
    String hodnota = a.getValue();
    */
    return Integer.parseInt(hodnota);
}

private static double getCenaOvoce(Node ovoce) {
```

```
double jednotkovaCena = getJednotkovaCena(ovoce);
Node vaha = getNodePodleJmena(ovoce, "vaha");
Node text = getNodePodleJmena(vaha, "#text");
String hodnota = text.getNodeValue().trim();
return Double.parseDouble(hodnota) * jednotkovaCena;
}
```

```
private static Node getNodePodleJmena(Node rodic, String jmeno) {
    NodeList nl = rodic.getChildNodes();
    for (int i = 0; i < nl.getLength(); i++) {
        Node e = nl.item(i);
        if (e.getNodeName().equals(jmeno) == true) {
            return e;
        }
    }
    return null;
}
```

5.6. Problém vkládaných elementů (odřádkování)

■ postup použitý v předchozím příkladě v metodě `getNodePodleJmena()` se může zdát přehnaně opatrný

- v souboru `jidlo.xml` jsou přece všechny elementy potomků na jasně definovaných pozicích

♦ proč by je měl parser měnit?

■ parser pozice nodů nemění, ale (jak bylo ukázáno dříve) defaultně vkládá další textové nody, vzniklé odřádkováním mezi elementy původního XML dokumentu

- toto může být při zpracování značný problém, protože různé způsoby odřádkování a formátování mezerami víceméně nelze zachytit žádnou validací

■ vypíšeme-li si potomky prvního ovoce, dostaneme:

```
Potomci ovoce:
#text
nazev
#text
vaha
#text
```

■ těmto vkládaným textovým nodům se dá dle informace v manuálech (a v Java Core API) zabránit použitím metody `void setIgnoringElementContentWhitespace(boolean whitespace)` třídy `DocumentBuilderFactory`

tj. prakticky: `dbf.setIgnoringElementContentWhitespace(false);`

■ při použití této metody dostaneme očekávané:

```
Potomci ovoce:
navez
vaha
```

■ situace ale není tak jednoduchá

- celý princip funguje správně jen pokud lze XML dokument validovat oproti DTD (nebo XSD) – viz SAX a též dále
- bez podpory DTD nebo XSD souborů celý systém nefunguje (nehledě na zapnutou či vypnutou validaci)

Příklad 5.2. Funkční program s automatickým odstraňováním „odřádkovacích“ nodů

```
public class PotomciOvoceDOM {
    private static final String SOUBOR = "jidlo-dtd.xml";

    public static void main(String[] args) {
        try {
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            dbf.setValidating(false);
            dbf.setIgnoringElementContentWhitespace(true);
            DocumentBuilder builder = dbf.newDocumentBuilder();
            builder.setErrorHandler(new ChybyZjisteneParserem());

            Document doc = builder.parse(SOUBOR);
            System.out.println("Potomci ovoce: ");
            tiskPotomci(doc);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static void tiskPotomci(Document doc) {
        NodeList nl = doc.getElementsByTagName("ovoce");
        Node ovoce = nl.item(0);
        NodeList nlo = ovoce.getChildNodes();
        for (int i = 0; i < nlo.getLength(); i++) {
            System.out.println(nlo.item(i).getNodeName());
        }
    }
}
```

Závěr

- použití `setIgnoringElementContentWhitespace()` je závislé na validačních souborech, tj. není obecné
- doporučení – tento způsob nepoužívat a nody hledat „opatrnou“ metodou
- nebo ihned po načtení XML dokumentu nejdříve odstranit všechny „odřádkovací“ nody – viz dále

5.7. Automatické odstranění komentářů

Výstraha

Funguje jen pro JDK 1.6, respektive pro JAXP 1.4.

- velice podobné výše uvedenému způsobu odstranění „odřádkovacích“ uzlů
- rozdíl je, že odstranění komentářů není závislé na existenci schémového souboru (DTD či XSD)
 - způsob lze použít vždy
- odstraňování zapneme voláním metody `setIgnoringComments()` ze třídy `DocumentBuilderFactory`

Příklad je téměř totožný s předchozím příkladem.

Čtený soubor `jidlo-komentar2.xml` začíná takto:

```
<?xml version="1.0" encoding="windows-1250"?>
<jidlo>
  <ovoce cislo="1">
    <navez jednotkovaCena="10">jablka</navez>
    <!-- docela velka vaha -->
    <vaha>2.5</vaha>
  </ovoce>
  ...
```

Zdrojový kód programu `PotomciOvoceKomentareDOM.java` začíná takto:

```
public class PotomciOvoceKomentareDOM {
    private static final String SOUBOR = "jidlo-komentar2.xml";

    public static void main(String[] args) {
        try {
            DocumentBuilderFactory dbf =
                DocumentBuilderFactory.newInstance();
            dbf.setValidating(false);
            dbf.setIgnoringComments(true);
            DocumentBuilder builder = dbf.newDocumentBuilder();
            builder.setErrorHandler(new ChybyZjisteneParserem());

            Document doc = builder.parse(SOUBOR);
            System.out.println("Potomci ovoce: ");
            tiskPotomci(doc);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Metoda `tiskPotomci(Document doc)` je naprosto stejná, jako v předchozím příkladu.

Pro nastavení `dbf.setIgnoringComments(false)`; program vypíše:

```
Potomci ovoce:
#text
nazev
#text
#comment
#text
vaha
#text
```

Pro nastavení `dbf.setIgnoringComments(true)`; program vypíše:

```
Potomci ovoce:
#text
nazev
#text
vaha
#text
```

Poznámka

Pokud bychom tento XML dokument chtěli zpětně zapisovat do souboru, je třeba si uvědomit, že komentáře ve výsledném souboru již nebudou, protože byly z infosetu odstraněny hned při načítání.

5.8. Všechny objekty v paměti

Program uloží všechny hodnoty a atributy najednou do paměti a pak je zpracovává

■ třídy `Ovoce` a `ZpracovaniDatVPameti` jsou zcela stejné, jako v SAX

■ hlavní program:

```
Document doc = builder.parse(SOUBOR);

ArrayList<Ovoce> ar = UlozeniDoPameti.getSeznam(doc);
ZpracovaniDatVPameti.tiskniVse(ar);
System.out.println("Celkova vaha = "
    + ZpracovaniDatVPameti.celkovaVaha(ar));
System.out.println("Celkova cena = "
    + ZpracovaniDatVPameti.celkovaCena(ar));
```

■ ukládání dat do paměti:

```
import java.util.*;
import org.w3c.dom.*;

public class UlozeniDoPameti {
    private static ArrayList<Ovoce> ar = new ArrayList<Ovoce>();

    public static ArrayList<Ovoce> getSeznam(Document doc) {
        NodeList nl = doc.getElementsByTagName("ovoce");
        for (int i = 0; i < nl.getLength(); i++) {
```

```
Node e = nl.item(i);
if (e.getNodeType() == Node.ELEMENT_NODE) {
    ar.add(vytvorOvoce(e));
}
}
return ar;
}
```

```
private static Ovoce vytvorOvoce(Node ovoce) {
    String hodnota =
        ((Element)ovoce).getAttributeNode("cislo").getValue();
    int cislo = Integer.parseInt(hodnota);
    int jednotkovaCena = getJednotkovaCena(ovoce);
    String nazev = getNazev(ovoce);
    double vaha = getVahaOvoce(ovoce);
    return new Ovoce(cislo, nazev, jednotkovaCena, vaha);
}
```

```
private static int getJednotkovaCena(Node ovoce) {
    Node nazev = getNodePodleJmena(ovoce, "nazev");
    NamedNodeMap nnm = nazev.getAttributes();
    String hodnota =
        nnm.getNamedItem("jednotkovaCena").getNodeValue();
    return Integer.parseInt(hodnota);
}
```

```
private static String getNazev(Node ovoce) {
    Node nazev = getNodePodleJmena(ovoce, "nazev");
    Node text = getNodePodleJmena(nazev, "#text");
    return text.getNodeValue();
}
```

```
private static double getVahaOvoce(Node ovoce) {
    Node vaha = getNodePodleJmena(ovoce, "vaha");
    Node text = getNodePodleJmena(vaha, "#text");
    String hodnota = text.getNodeValue().trim();
    return Double.parseDouble(hodnota);
}
```

```
private static Node getNodePodleJmena(Node rodic, String jmeno) {
    NodeList nl = rodic.getChildNodes();
    for (int i = 0; i < nl.getLength(); i++) {
        Node e = nl.item(i);
        if (e.getNodeName().equals(jmeno) == true) {
            return e;
        }
    }
    return null;
}
```

5.9. Průchod stromem dokumentu

- v některých případech se mohou hodit další způsoby průchodu stromem

- typicky je to, pokud neznáme dopředu strukturu XML dokumentu

- rozhraní s příslušnými metodami jsou definovány v DOM2

- patří do balíku `org.w3c.dom.traversal`

Výstraha

dokumentaci je třeba hledat na velmi netypickém místě

```
jdk1.6.0\docs\jre\api\plugin\dom\org\w3c\dom\traversal\
```

- používají se dvě rozhraní

1. `NodeIterator` – průchod nody lineárně (jsou v seznamu)

2. `TreeWalker` – průchod nody hierarchicky

- oba dva mají možnost použít vlastní filtr, kterým se můžeme zbavit nepotřebných informací

- filtr musí splňovat rozhraní `NodeFilter`

- poslední rozhraní z tohoto balíku je `DocumentTraversal`, pomocí něhož se vytváří objekty `NodeIterator` a `TreeWalker`

```
NodeIterator ni = ((DocumentTraversal) doc).createNodeIterator(...);
```

nebo

```
TreeWalker tw = ((DocumentTraversal) doc).createTreeWalker(...);
```

- obě metody `create...()` mají stejné čtyři parametry

1. `Node root` – kořenový nod, odkud se bude procházet

- nemusí to být nutně kořenový nod XML dokumentu

- ♦ výhoda, protože můžeme pracovat s podstromem

2. `int whatToShow` – jaké typy nodů se budou zobrazovat

- využívají se konstanty z `NodeFilter` např.:

```
SHOW_ALL
```

```
SHOW_COMMENT
```

```
SHOW_ELEMENT
```

```
SHOW_TEXT
```

- konstanty se dají sčítat, např.:

```
NodeFilter.SHOW_COMMENT + NodeFilter.SHOW_TEXT
```

- ♦ zobrazí jen komentářové a textové nody

Výstraha

- konstanta `NodeFilter.SHOW_ATTRIBUTE` má význam pouze, je-li kořenovým nodem atributový nod

- ♦ prakticky je tedy nepoužitelná, protože atributy nejsou zařazeny v dokumentovém stromě

3. `NodeFilter filter` – zjemnění předchozího parametru

- z vybraných typů nodů se budou dál vybírat jen některé (viz dále)

- nepoužíváme-li filtr, má parametr hodnotu `null`

`NodeFilter` má jen jednu metodu `public short acceptNode(Node n)` – ta může vrátit tři hodnoty:

- a. `NodeFilter.FILTER_ACCEPT` – tento nod je povoleno dál zpracovávat

- b. `NodeFilter.FILTER_SKIP` – tento nod se dál nezpracovává, ale jeho případní potomci ano

- c. `NodeFilter.FILTER_REJECT` – tento nod ani jeho potomci se dál nezpracovává

4. `boolean entityReferenceExpansion` – povolení expanze entit (má smysl jen pro dokumenty využívající DTD)

- pro běžné datově orientované dokumenty nastavit na `false`

Příklad 5.3.

Program vypíše nejdříve názvy všech elementů a jejich případných hodnot a pak (stejnou výpisovou metodou) jen názvy a hodnoty elementů `nazev` a `vaha`

```
...
Document doc = builder.parse(SOUBOR);

// vypis vseh elementu a textu
NodeIterator ni = ((DocumentTraversal) doc).createNodeIterator(
    doc.getDocumentElement(),
    NodeFilter.SHOW_ELEMENT + NodeFilter.SHOW_TEXT,
    null, false);

vypisSeznam(ni);

System.out.println("Filtrovany vystup");
ni = ((DocumentTraversal) doc).createNodeIterator(
    doc.getDocumentElement(),
    NodeFilter.SHOW_ELEMENT + NodeFilter.SHOW_TEXT,
    new Filtr(), false);

vypisSeznam(ni);
}
catch (Exception e) {
    e.printStackTrace();
}
}

private static void vypisSeznam(NodeIterator ni) {
    Node n;
    while ((n = ni.nextNode()) != null) {
        if (n.getNodeType() == Node.ELEMENT_NODE) {
            System.out.print(n.getNodeName() + ": ");
        }
        else {
            System.out.println(n.getNodeValue());
        }
    }
}

class Filtr implements NodeFilter {
    public short acceptNode(Node n) {
        if (n.getNodeName().equals("ovoce") == true) {
            return NodeFilter.FILTER_SKIP;
        }
        else if (n.getNodeType() == Node.TEXT_NODE
            && n.getNodeValue().trim().length() == 0) {
            // odradkovani
            return NodeFilter.FILTER_SKIP;
        }
        return NodeFilter.FILTER_ACCEPT;
    }
}
```

- vypíše (v prvním případě vypisuje i „odrádkovací“ nody):

jidlo:

ovoce:

nazev: jablka

vaha: 2.5

...

Filtrovany vystup

jidlo: nazev: jablka

vaha: 2.5

nazev: banány

vaha: 2

nazev: grapefruity

vaha: 0.75

nazev: švestky sušené

vaha: 1.8

- užitečné je, že se po seznamu nodů můžeme pohybovat i nazpět metodou `Node previousNode()`
- `TreeWalker` nám z tohoto pohledu dává možností mnohem více, protože udržuje hierarchickou strukturu nodů

- může používat

`Node nextNode()`

`Node previousNode()`

- ♦ jako `NodeIterator`, kdy se průchod uskutečňuje postupně po jednotlivých větvích stromu

- další možnosti pohybu jsou stejné, jako má `Node`

`Node parentNode()`

`Node nextSibling()`

`Node previousSibling()`

`Node firstChild()`

`Node lastChild()`

- výhodou `TreeWalker` je ale, že tyto nody jsou už „profilované“ příslušným `NodeFilter`, takže obsahují jen to, co nás zajímá, např. elementy

5.10. Snadné odstranění „odřádkovacích“ nodů

■ tyto nody nás matou (jsou tam jaksi navíc) při průchodu DOM dokumentu

- s využitím `NodeIterator` je můžeme velmi snadno odstranit
- při případném zápisu se dají automaticky doplnit – viz dále

```
public static void odstranMezeryALF(Document doc) {
    Node n = doc.getDocumentElement();
    NodeIterator ni = ((DocumentTraversal) doc).createNodeIterator(
        doc.getDocumentElement(),
        NodeFilter.SHOW_TEXT,
        new PrazdnyText(), true);
    while ((n = ni.nextNode()) != null) {
        Node rodic = n.getParentNode();
        rodic.removeChild(n);
    }
}

private static class PrazdnyText implements NodeFilter {
    public short acceptNode(Node n) {
        if (n.getNodeValue().trim().length() == 0) {
            return NodeFilter.FILTER_ACCEPT;
        }
        else {
            return NodeFilter.FILTER_SKIP;
        }
    }
}
```

■ po volání:

```
odstranMezeryALF(doc);
```

jsou všechny odřádkovací nody z DOM odstraněny

Varování

Pokud bychom takto zpracovávali XML dokument orientovaný na sdělení, mohlo by se stát, že vymažeme i významový „odřádkovací“ nod.

5.11. Zápis dokumentu

■ značnou výhodou DOM proti SAX je možnost načtený dokument zapsat na disk (obecně uložit)

- typicky se tato akce nazývá „serializace“
- JAXP pro to poskytuje velmi účinnou podporu, protože DOM strom lze zapsat několika různými způsoby (transformovat)

Poznámka

Transformace XML dokumentů jsou další silnou zbraní XML. Využívá se stylový jazyk XSLT (*eX-tensible Stylesheet Language*) s mnoha možnostmi, které zde nebudou popisovány. Zde bude používána jen transformace XML do XML a to 1:1 (tj. stejná struktura obou XML dokumentů).

■ pro serializaci se používají třídy JAXP z balíku `javax.xml.transform`

- `TransformerFactory` a `Transformer` ve stejném duchu, jako při vytváření SAX či DOM parseru
- dále potřebujeme třídu `DOMSource` z balíku `xml.transform.dom` a třídu `StreamResult` z balíku `javax.xml.transform.stream`

■ konkrétní program pro zápis (odstíněný pomocí JAXP) je Xalan

- další často používaný je Saxon

5.11.1. Ovlivnění práce transformační třídy

■ metodu `setOutputProperty()` třídy `Transformer` se dvěma formálními parametry

- prvním je jméno nastavované transformační vlastnosti a druhým hodnota této vlastnosti

■ skutečným prvním parametrem jsou konstanty třídy `javax.xml.transform.OutputKeys` z nichž jsou významové

- `OMIT_XML_DECLARATION`

je-li druhým skutečným parametrem řetězec `"yes"`, pak nebude ve výsledném XML dokumentu uvedena jeho hlavička

```
<jidlo>
  <ovoce cislo="1">
```

druhou možností je řetězec `"no"`, a toto nastavení způsobí, že výsledný XML dokument bude obsahovat hlavičku

```
<?xml version="1.0" encoding="windows-1250" standalone="no"?>
<jidlo>
  <ovoce cislo="1">
```

Implicitní nastavení této vlastnosti je:

```
setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "no");
```

- `ENCODING`

druhým skutečným parametrem je řetězec udávající požadované kódování pro výstupní XML dokument.

implicitní nastavení této vlastnosti je převzato z hlavičky vstupního XML dokumentu; nebylo-li tam uvedeno, případně pokud DOM vznikl jinak než čtením XML dokumentu, je implicitní nastavení:

```
setOutputProperty(OutputKeys.ENCODING, "UTF-8");
```

Výstraha

Pro JDK 1.6 toto nastavení funguje při transformaci, ale bohužel se neprojeví zápisem do hlavičky vzniklého XML dokumentu; řešení viz dále.

- INDENT

druhým skutečným parametrem je řetězec "yes" nebo "no" - viz dále

- METHOD

druhým skutečným parametrem je řetězec "xml" nebo "text"

- ♦ pro "xml" se provede očekávaná transformace 1:1

- ♦ pro "text" použijeme chceme-li z XML dokumentu získat pouze hodnoty elementů

```
setOutputProperty(OutputKeys.METHOD, "text");
```

dostaneme výstupní soubor:

```
jablka
2.5

banány
2

grapefruity
0.75

švestky sušené
1.8
```

5.11.2. Ukázka zápisu do souboru

Program přečte soubor `jidlo.xml` a uloží jej do souboru `jidlo-utf-8.xml`, tj. změni mu kódování.

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

public class TransformaceJidloDOM1_6 {
    private static final String SOUBOR = "jidlo.xml";
    private static final String VYSTUPNI_SOUBOR = "jidlo-UTF-8.xml";
```

```
public static void main(String[] args) {
    try {
        DocumentBuilderFactory dbf =
            DocumentBuilderFactory.newInstance();
        dbf.setValidating(false);

        DocumentBuilder builder = dbf.newDocumentBuilder();
        builder.setErrorHandler(new ChybyZjisteneParserem());
        Document doc = builder.parse(SOUBOR);

        String kodovani = "UTF-8";

        TransformerFactory tf = TransformerFactory.newInstance();
        Transformer zapisovac = tf.newTransformer();
        zapisovac.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION,
            "yes");
        zapisovac.setOutputProperty(OutputKeys.METHOD, "xml");
        DOMResult DOMbezHlavicky = new DOMResult();
        zapisovac.transform(new DOMSource(doc),
            DOMbezHlavicky);

        zapisovac.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION,
            "no");
        zapisovac.setOutputProperty(OutputKeys.ENCODING, kodovani);
        // nefunguje
        zapisovac.setOutputProperty(OutputKeys.STANDALONE, "yes");
        // nastaveni standalone="yes"
        Document doc2 = (Document) DOMbezHlavicky.getNode();
        doc2.setXmlStandalone(true);

        zapisovac.transform(new DOMSource(doc2),
            new StreamResult(
                new File(VYSTUPNI_SOUBOR)));
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

první transformace s nastavením

```
OutputKeys.OMIT_XML_DECLARATION, "yes"
```

způsobí, že dočasný DOM nebude mít hlavičku

nad tímto DOM se provede druhá transformace, při které se změni výstupní kódování a také nastaví požadavek na vytvoření hlavičky, do které již bude zvolené výstupní kódování správně zapsáno

ale v hlavičce se použije implicitní `standalone="no"`, které nelze přepsat pomocí očekávaného:

```
OutputKeys.OUTPUT_KEYS.STANDALONE, "yes"
```

a musí se proto použít trik, ve kterém se nastaví standalone pomocí metody `setXmlStandalone(true)`;

bez tohoto komplikovaného postupu dvojitá transformace dostaneme výstupní XML soubor v požadovaném kódování, ale v hlavičce bude uvedeno kódování původního vstupního XML dokumentu, což je samozřejmě hrubá chyba

5.11.3. Problematika odřádkování a odsazování

■ v XML dokumentu nezáleží na odřádkování odsazení elementů

- následující dva XML dokumenty nesou stejnou informaci
 - ♦ druhý příklad je jasně přehlednější

```
<jidlo><ovoce cislo="4"><nazev
jednotkovaCena="32">švestky</nazev><vaha>1.8</vaha></ovoce></jidlo>
```

a

```
<jidlo>
  <ovoce cislo="4">
    <nazev jednotkovaCena="32">švestky</nazev>
    <vaha>1.8</vaha>
  </ovoce>
</jidlo>
```

■ vytváříme-li nový XML dokument, je zajištění správného odřádkování a odsazení nepřijemná práce (viz dále)

- tuto činnost lze ale automaticky zajistit nastavením parametrů výstupu

```
zapisovac.setOutputProperty(OutputKeys.INDENT, "yes");
```

které zajistí odřádkování elementů a i jejich odsazení

■ hloubku odsazení (zde dvě mezery) lze nastavit voláním:

```
zapisovac.setOutputProperty("{http://xml.apache.org/xalan}indent-amount",
"2");
```

nebo lépe (přenositelněji)

```
zapisovac.setOutputProperty("{http://xml.apache.org/xslt}indent-amount",
"2");
```

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
```

```
public class OdsazovaniVystupuDOM1_6 {
    private static final String SOUBOR = "jidlo-neodsazene2.xml";
    private static final String VYSTUPNI_SOUBOR = "jidlo-odsazene.xml";
```

```
    public static void main(String[] args) {
        try {
            DocumentBuilderFactory dbf =
                DocumentBuilderFactory.newInstance();
            dbf.setValidating(false);
            DocumentBuilder builder = dbf.newDocumentBuilder();
            builder.setErrorHandler(new ChybyZjisteneParserem());
            Document doc = builder.parse(SOUBOR);

            TransformerFactory tf = TransformerFactory.newInstance();
            Transformer zapisovac = tf.newTransformer();
            zapisovac.setOutputProperty(OutputKeys.INDENT, "yes");
            zapisovac.setOutputProperty(
                "{http://xml.apache.org/xalan}indent-amount",
                "2");
            zapisovac.setOutputProperty(
                "{http://xml.apache.org/xslt}indent-amount",
                "2");

            zapisovac.transform(new DOMSource(doc),
                                new StreamResult(
                                    new File(VYSTUPNI_SOUBOR)));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

5.12. Modifikace dokumentu

■ další výhodou DOM proti SAX je možnost načtený dokument měnit (zmenšovat, zvětšovat, opravovat)

Poznámka

Samozřejmě se nemusíme starat např. o správný zápis entit typu & nebo <

5.12.1. Změna hodnoty již existujícího elementu či atributu

■ metoda třídy `Node`

```
void setNodeValue(String nodeValue)
```

- kterou je ale třeba používat jen na správné typy nodů

• lepší metody:

- ♦ ze třídy `Element`

```
void setAttribute(String name, String value)
```

◆ ze třídy `Attr`

```
void setValue(String value)
```

◆ ze třídy `Text`

```
Text replaceWholeText(String content)
```

5.12.2. Odstranění nodu

■ v případě nodů `Element` nebo `Text` musíme být v rodičovském nodu `Element` a pak použijeme:

```
Node removeChild(Node oldChild)
```

■ pro odstranění atributu musíme být v nodu `Element`, ke kterému odstraňovaný atribut patří

```
void removeAttribute(String name)
```

5.12.3. Vkládání nových nodů

■ nejobtížnější akce

■ nejprve je nutné nový nod vytvořit jednou z metod třídy `org.w3c.dom.Document`

- `Element createElement(String tagName)`

- `Text createTextNode(String data)`

- `Attr createAttribute(String name)`

Poznámka

Atributy je lepší vytvářet na hotových elementech metodou `void setAttribute(String name, String value)`

■ pak už lze nody pracovat

- `Node insertBefore(Node newChild, Node refChild)` – vkládat

- `Node replaceChild(Node newChild, Node oldChild)` – zaměňovat

- `Node appendChild(Node newChild)` — přidá nod jako poslední do seznamu dětí

Poznámka

■ chceme-li mít výstupní XML soubor s úhledně odsazenými elementy, je nutné před a za přidávaný element přidat ještě nod `Text` s odřádkováním a odsazením

- pro odřádkování se použije jen `"\n"`

- ◆ `"\r"` způsobí nadbytečný zápis textu ``

5.12.4. Změna XML dokumentu a jeho zápis

```
...
TransformerFactory tf = TransformerFactory.newInstance();
Transformer zapisovac = tf.newTransformer();
zapisovac.setOutputProperty(OutputKeys.ENCODING,
    "windows-1250");

zapisovac.transform(new DOMSource(doc),
    new StreamResult(
        new File(VYSTUPNI_SOUBOR)));
...
```

```
private static void zmenDokument(Document doc) {
    Node jidlo = doc.getDocumentElement();

    // odstraneni <ovoce cislo="1">
    NodeList nl = doc.getElementsByTagName("ovoce");
    Node ovoce1 = nl.item(0);
    jidlo.removeChild(ovoce1);

    // odstraneni atributu cislo="2"
    Element ovoce2 = (Element) nl.item(0);
    ovoce2.removeAttribute("cislo");

    // zmena jednotkovaCena="25" na "15"
    NodeList nlNazev = doc.getElementsByTagName("nazev");
    Element nazev = (Element) nlNazev.item(0);
    nazev.setAttribute("jednotkovaCena", "15");

    // pridani atributu <vaha jednotka="kg">2</vaha>
    NodeList nlVaha = doc.getElementsByTagName("vaha");
    Element vaha = (Element) nlVaha.item(0);
    vaha.setAttribute("jednotka", "kg");

    // pridani elementu <vzhled>cerstvy & vonavy</vzhled>
    Element vzhled = doc.createElement("vzhled");
    Text text = doc.createTextNode("cerstvy & vonavy");
    vzhled.appendChild(text);
    ovoce2.appendChild(vzhled);

    // odradkovani za elementem </vzhled>
    Text textCRLF = doc.createTextNode("\n  ");
    ovoce2.appendChild(textCRLF);
}
```

vytvoří

```
<?xml version="1.0" encoding="windows-1250" standalone="no"?><jidlo>

<ovoce>
  <nazev jednotkovaCena="15">banány</nazev>
  <vaha jednotka="kg">2</vaha>
```

```

<vzhled>čerstvý &amp; voňavý</vzhled>
</ovoce>
<ovoce cislo="3">
  <nazev jednotkovaCena="19">grapefruity</nazev>
  <vaha>0.75</vaha>
</ovoce>
<ovoce cislo="4">
  <nazev jednotkovaCena="32">švestky sušené</nazev>
  <vaha>1.8</vaha>
</ovoce>
</jidlo>

```

5.13. Vytváření nového dokumentu

- chceme-li v paměti vytvořit zcela nový dokument, potřebujeme instanci třídy, která splňuje rozhraní `org.w3c.dom.DOMImplementation`

- tu můžeme získat použitím známých příkazů JAXP

```

DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = dbf.newDocumentBuilder();
DOMImplementation impl = builder.getDOMImplementation();

```

- rozhraní `DOMImplementation` pak poskytuje metodu `Document createDocument(String namespaceURI, String rootQName, DocumentType doctype)`

- ♦ parametry `namespaceURI` a `doctype` jsou většinou `null`

- od této chvíle máme objekt třídy `Document`, se kterým můžeme už pracovat běžným způsobem

5.13.1. Klonování nodů

- vytváříme-li XML dokument, ve kterém se nody opakují pouze se změněnými hodnotami atributů či elementů, může být vhodné připravit si jeden vzorový nod

- ostatní nody pak vytvářet klonováním vzorového nodu

- práce se změnou hodnot atributů a elementů je sice stále velká (viz dříve), ale v případě složitějších nodů bude zřejmě menší, než opětovné kompletní vytváření nodu

- ze třídy `Node` použijeme metodu

```
Node cloneNode(boolean hlubokaKopie)
```

- je-li `hlubokaKopie true`, vytvoří se kopie nodu se všemi případnými potomky, což většinou chceme

Program vytvoří `jidlo-nove.xml` se dvěma stejnými elementy `<ovoce>` přičemž ten druhý vzniknul klonováním. V příkladu je také vidět, že je automaticky zajištěno odřádkování a odsazení.

```

public class NoveJidloDOM {
  private static final String VYSTUPNI_SOUBOR =
    "jidlo-nove.xml";

```

```

public static void main(String[] args) {
  try {
    Document zdroj = vytvorDocument();

    TransformerFactory tf = TransformerFactory.newInstance();
    Transformer zapisovac = tf.newTransformer();
    zapisovac.setOutputProperty(OutputKeys.INDENT, "yes");
    zapisovac.setOutputProperty(
      "{http://xml.apache.org/xslt}indent-amount",
      "2");
    zapisovac.setOutputProperty(OutputKeys.ENCODING,
      "windows-1250");

    zapisovac.transform(new DOMSource(zdroj),
      new StreamResult(
        new File(VYSTUPNI_SOUBOR)));
  } catch (Exception e) {
    e.printStackTrace();
  }
}

```

```

private static Document vytvorDocument() throws Exception {
  DocumentBuilderFactory dbf =
    DocumentBuilderFactory.newInstance();
  dbf.setValidating(false);
  DocumentBuilder builder = dbf.newDocumentBuilder();
  DOMImplementation impl = builder.getDOMImplementation();
  Document doc = impl.createDocument(null, "jidlo", null);
  Node jidlo = doc.getDocumentElement();

```

```

  Element ovoce = doc.createElement("ovoce");
  ovoce.setAttribute("cislo", "10");
  Element nazev = doc.createElement("nazev");
  nazev.setAttribute("jednotkovaCena", "30");
  Text textNazev = doc.createTextNode("švestky");
  nazev.appendChild(textNazev);
  Element vaha = doc.createElement("vaha");
  Text textVaha = doc.createTextNode("1.5");
  vaha.appendChild(textVaha);

```

```

  ovoce.appendChild(nazev);
  ovoce.appendChild(vaha);

```

```

  Element ovoceKlon = (Element) ovoce.cloneNode(true);
  jidlo.appendChild(ovoce);
  jidlo.appendChild(ovoceKlon);
  return doc;
}

```

vytvoří:

```
<?xml version="1.0" encoding="windows-1250" standalone="no"?>
<jidlo>
  <ovoce cislo="10">
    <nazev jednotkovaCena="30">švestky</nazev>
    <vaha>1.5</vaha>
  </ovoce>
  <ovoce cislo="10">
    <nazev jednotkovaCena="30">švestky</nazev>
    <vaha>1.5</vaha>
  </ovoce>
</jidlo>
```

5.14. Validace nově vytvořeného nebo měněného dokumentu

- pokud XML dokument v paměti vytváříme nebo programově měníme, může být dobré si výsledek zvalidovat také programově

- jinak je nutné zapsat výsledek do XML souboru a ten externě zvalidovat

- pro validaci se používají třídy z balíku `javax.xml.validation`

- před tím je třeba nastavit objekt třídy `DocumentBuilderFactory` metodou `setNamespaceAware()`

- je-li její skutečný parametr `false` (což je též implicitní nastavení tohoto objektu), dostaneme „nesmyslné“ chybové hlášení:

```
Chyba validace:
Chyba: System ID: null
radka: -1 sloupec: -1
cvc-elt.1: Cannot find the declaration of element 'jidlo'.
```

- pro správný průběh validace je tedy nutné použít volání `setNamespaceAware(true)`;

Program validuje načtený XML dokument.

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import javax.xml.*;
import javax.xml.validation.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

public class ValidaceJidloDOM1_6 {
    private static final String SOUBOR = "jidlo.xml";
    // private static final String SOUBOR = "jidlo-chyba-validace.xml";
    private static final String VALIDACNI_SOUBOR = "jidlo.xsd";
```

```
public static void main(String[] args) throws Exception {
    try {
        DocumentBuilderFactory dbf =
            DocumentBuilderFactory.newInstance();
        dbf.setNamespaceAware(true);
        // dbf.setNamespaceAware(false);
        DocumentBuilder builder = dbf.newDocumentBuilder();
        Document zdroj = builder.parse(SOUBOR);

        SchemaFactory sf = SchemaFactory.newInstance(
            XMLConstants.W3C_XML_SCHEMA_NS_URI);
        Schema sch = sf.newSchema(new File(VALIDACNI_SOUBOR));
        Validator val = sch.newValidator();
        val.setErrorHandler(new ValidaceChybyZjisteneParserem());
        val.validate(new DOMSource(zdroj));
        System.out.println("Validace probehla uspesne");
    }
    catch (SAXException e) {
        System.out.println("Chyba validace:");
        // e.printStackTrace();

        String s = e.getMessage();
        int i = s.indexOf(ValidaceChybyZjisteneParserem.KONEC);
        if (i > 0) {
            s = s.substring(0, i);
        }
        System.out.println(s);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

V případě bezchybného DOM (zde soubor `jidlo.xml`) vypíše:

```
Validace probehla uspesne
```

V případě výskytu chyby (zde nesmyslná hodnota "ABC1" atributu `cislo` v souboru `jidlo-chyba-validace.xml`) vypíše:

```
Chyba validace:
Chyba: System ID: null
radka: -1 sloupec: -1
cvc-datatype-valid.1.2.1: 'ABC1' is not a valid value for 'integer'.
```

- v případě validování nově vytvořeného DOM postup s nastavením `setNamespaceAware(true)`; nefunguje

- náhradní řešení v metodě `konverze()` transformuje DOM na objekt typu `Reader` (ve skutečnosti je to XML dokument zapsaný v jednom řetězci)

- pak stačí jen upravit skutečný parametr metody `validate()`

```

import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import javax.xml.*;
import javax.xml.validation.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

public class ValidaceNoveJidloDOM1_6 {
    private static final String VALIDACNI_SOUBOR = "jidlo.xsd";

    public static void main(String[] args) throws Exception {
        try {
            Document zdroj = vytvorDocument();
            Reader r = konverze(zdroj);

            SchemaFactory sf = SchemaFactory.newInstance(
                XMLConstants.W3C_XML_SCHEMA_NS_URI);
            Schema sch = sf.newSchema(new File(VALIDACNI_SOUBOR));
            Validator val = sch.newValidator();
            val.setErrorHandler(new ValidaceChybyNalezeneParserem());
            val.validate(new StreamSource(r));
            System.out.println("Validace probehla uspesne");
        }
        catch (SAXException e) {
            System.out.println("Chyba validace:");
            // e.printStackTrace();

            String s = e.getMessage();
            int i = s.indexOf(ValidaceChybyNalezeneParserem.KONEC);
            if (i > 0) {
                s = s.substring(0, i);
            }
            System.out.println(s);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static Reader konverze(Document zdroj) {
        try {
            TransformerFactory tf = TransformerFactory.newInstance();
            Transformer zapisovac = tf.newTransformer();
            zapisovac.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION,
                "yes");
            StringWriter sw = new StringWriter();
            StreamResult stres = new StreamResult(sw);
            zapisovac.transform(new DOMSource(zdroj),
                stres);

```

```

        return new StringReader(sw.toString());
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}

private static Document vytvorDocument() throws Exception {
    ...

```


Kapitola 6. Grafické uživatelské rozhraní – GUI

6.1. Základní informace

■ slouží pro zobrazovací vrstvu (*presentation layer*)

■ v Core API dva základní typy:

1. AWT (*Abstract Windowing Toolkit*) – starší, jednoduché, jen primitivní komponenty, rychlé (přímo na API OS) – vzhled podle OS, podpora v WWW prohlížečích (MSIE), `java.awt.Button`
2. Swing (kódové jméno) – novější, velmi komplexní (přes 30 různých komponent), všechny typy komponent, pomalé (všechny komponenty se vykreslují samy) – vzhled nezávislý na OS, do MSIE plugin nebo odkaz v HTML, `javax.swing.JButton`

- jeho „spodní“ vrstva je AWT (`java.awt.Component`)
- souběžně využívá AWT (událostní model, fonty, barvy, layouty)

■ několik konkurenčních GUI od třetích stran – např. SWT od IBM (používáno v Eclipse)

■ GUI prošlo značným vývojem:

- JDK 1.0 – AWT se nedokonalou obsluhou událostí
- JDK 1.1 – AWT se současnou obsluhou událostí, Swing v přípravné fázi jako přidavné balíky
- JDK 1.2 – Swing součástí Core API
- JDK 1.3 – Swing velký rozvoj knihoven a zrychlení, AWT zcela zavrženo (přesto stále žije ;-)
- JDK 1.4 – rozšíření knihoven (15 balíků) a mírná vylepšení
- JDK 1.5 – rozšíření knihoven (+3 balíky), přidání mnoha funkcí ke stávajícím třídám, zlepšení rychlosti

■ vztah JFC (*Java Foundation Classes*) a Swing:

- JFC komplexní knihovna pro GUI, má části:
 - ♦ Swing – nejvýznamnější část
 - ♦ *Pluggable Look and Feel Support* – různý vzhled komponent
 - ♦ Java 2D API – 2D grafika
 - ♦ *Accessibility API* – podpůrné technologie (např. Brailův display)
 - ♦ *Drag and Drop Support* – pro spolupráci s jinými aplikacemi

■ základní učebnice *The Swing Tutorial*, dokumentace v Core API

■ nezbytné dovednosti ve Swing:

- zobrazení základního okna a přidání komponent

- princip reakce na události
- rozmístění komponent na ploše

6.2. Zobrazení základního okna a přidání komponent

■ výklad na příkladu tlačítka (`javax.swing.JButton`) – atomické

■ zatím nemá žádnou funkčnost

■ základní okénko (*top-level container*) – instance třídy `JFrame` – kontejnerové okénko

```
import javax.swing.*;
```

```
public class ZakladniDovednostiMain {
    public static void main(String[] args) {
        JFrame oknoF = new JFrame("ZakladniDovednostiMain");
        JButton tlacitkoBT = new JButton("Ahoj");
        oknoF.getContentPane().add(tlacitkoBT);
        oknoF.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        oknoF.setSize(250, 70);
        oknoF.setVisible(true);
    }
}
```



■ je vhodné zavést jednotný způsob pojmenovávání typů objektů, např. `tlacitkoBT`

■ komponenty se přidávají do mezilehlého (*intermediate*) kontejneru, nazývaného *content pane*

Poznámka

Od JDK 1.5 lze použít místo:

```
oknoF.getContentPane().add(tlacitkoBT);
```

pouze jednodušší:

```
oknoF.add(tlacitkoBT);
```

■ tento způsob zobrazení okna se používá pouze pro nejprimitivnější školní příklady

- neumožňuje dělení na aplikační a zobrazovací vrstvu

6.2.1. Používaný způsob zobrazení základního okna

■ zobrazovací třída dědí od JFrame

■ pro uložení více komponent používá JPanel

```
import java.awt.*; // FlowLayout
import javax.swing.*;

class ZobrazovaciTrida extends JFrame {
    ZobrazovaciTrida() {
        this.setTitle("ZakladniDovednostiDveTridy");
        JButton tlacitkoBT = new JButton("Ahoj");

        JPanel vnitrekPN = new JPanel();
        vnitrekPN.setLayout(new FlowLayout());
        vnitrekPN.add(tlacitkoBT);
        vnitrekPN.add(new JButton("Nazdar"));

        this.getContentPane().add(vnitrekPN);

        // this.setLocation(100, 200);
        this.setLocationRelativeTo(null); // střed obrazovky
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(280, 70);
        this.setVisible(true);
    }
}

public class Hlavni {
    public static void main(String[] args) {
        new ZobrazovaciTrida();
    }
}
```



6.2.2. Používaný způsob vložení komponent

■ komponent je většinou mnoho a různě rozmístěných v JFrame

- to by pak značně znepřehledňovalo konstruktor třídy
 - ◆ kód pro vložení komponent vyčleníme do samostatné metody

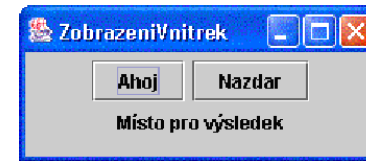
```
public class ZobrazeniVnitrek extends JFrame {
    JButton ahojBT;
    JButton nazdarBT;
    JLabel popisLB;
```

```
ZobrazeniVnitrek() {
    this.setTitle(getClass().getName());
    this.getContentPane().add(vytvorVnitrek());
    this.setLocationRelativeTo(null);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setSize(270, 100);
    this.setVisible(true);
}
```

```
Container vytvorVnitrek() {
    ahojBT = new JButton("Ahoj");
    nazdarBT = new JButton("Nazdar");
    popisLB = new JLabel("Místo pro výsledek");
```

```
JPanel vnitrekPN = new JPanel();
vnitrekPN.setLayout(new FlowLayout());
vnitrekPN.add(ahojBT);
vnitrekPN.add(nazdarBT);
vnitrekPN.add(popisLB);
```

```
return vnitrekPN;
}
```



6.3. Princip reakce na události

- promyšlený koncept událostí založený na návrhovém vzoru vydavatel-odběratel
 - jedna komponenta (zde tlačítko jako vydavatel) vytváří události a všechny ostatní komponenty (zde label jako odběratel) je zcela adresně zachycují
- vydavatel vysílá informaci o události jen odběratelům, které si sám zaregistroval (dále posluchač – *listener*)
 - registruje si posluchače pomocí své metody `addXYZListener()`
 - zaregistrovaným posluchačem může být pouze objekt, který má schopnost být posluchačem – musí implementovat rozhraní `XYZListener()`
 - ◆ to má většinou několik metod, z nichž každá reaguje na specifický typ události
 - ◆ zaslání zpráva pak způsobí vyvolání konkrétní metody
- události jsou potomci třídy `java.awt.AWTEvent` (nedůležité)

- všechny základní události jsou z AWT, Swing přidává speciální události vztahované ke komponentám
 - JDK1.5 rozlišuje celkem 41 typů událostí, z toho 18 z AWT
 - způsob reakce na událost je zcela stejný pro všechny možné události
- pro obsluhu jakékoliv události stačí pouze z dokumentace k Java Core API zjistit, jak se jmenuje rozhraní typu `Listener` od příslušného objektu
 - v popisu tohoto rozhraní se zjistí, jak se jmenují jeho metody
- tento jednoduchý princip je možné naprogramovat mnoha odlišnými způsoby (pro různé situace jsou vhodné různé postupy)
- v zásadě existují tři základní způsoby využívající:
 1. vnitřní třídy – nejpřehlednější
 2. anonymní vnitřní třídy – úsporný, ale málo čitelný
 3. anonymní vnitřní třídy a privátní metody – praktický
- platí obecná zásada – pro každou událost připravíme samostatnou obsluhu, tzn. kolik je zdrojů, tolik je obslužných tříd
 - nemícháme do sebe podobné věci
 - budeme-li chtít v budoucnu přidat další zdroj událostí, nemusíme žádným způsobem modifikovat kód stávajících obsluh
 - další výhodou je naprosto jasné oddělení činností – výhodné při pozdější změně funkčnosti některého zdroje
- další ukázky budou rozvíjet program `ZobrazeniVnitrek.java`

6.3.1. Použití vnitřních tříd

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

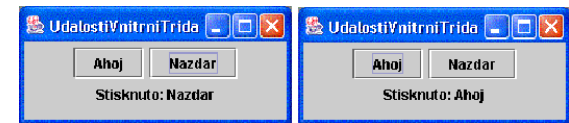
public class UdalostiVnitřníTřída extends JFrame {
    JButton ahojBT;
    JButton nazdarBT;
    JLabel popisLB;

    UdalostiVnitřníTřída() {
        this.setTitle(getClass().getName());
        this.getContentPane().add(vytvorVnitrek());
        obsluhyUdalosti();
        this.setLocationRelativeTo(null);
        ...
    }
    Container vytvorVnitrek() { ... }
```

```
private void obsluhyUdalosti() {
    ahojBT.addActionListener(new ALahojBT());
    nazdarBT.addActionListener(new ALnazdarBT());
}

private class ALahojBT implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        popisLB.setText("Stisknuto: " + ahojBT.getText());
    }
}

private class ALnazdarBT implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        popisLB.setText("Stisknuto: " + nazdarBT.getText());
    }
}
```



6.3.2. Použití anonymních vnitřních tříd

- vhodné, je-li v obsluze událostí pouze jediná a navíc krátká metoda
 - není nutné vymýšlet nové jméno pro vnitřní třídu
- v případě dlouhých obsluh nepřehledné

import ...

```
public class UdalostiAnonymniVnitřníTřída extends JFrame {
    JButton ahojBT;
    JButton nazdarBT;
    JLabel popisLB;

    UdalostiAnonymniVnitřníTřída() {
        this.setTitle(getClass().getName());
        this.getContentPane().add(vytvorVnitrek());
        obsluhyUdalosti();
        ...
    }

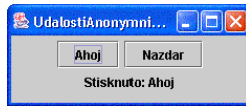
    Container vytvorVnitrek() { ... }

    private void obsluhyUdalosti() {
        ahojBT.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                popisLB.setText("Stisknuto: " + ahojBT.getText());
            }
        });
    }
}
```

```

nazdarBT.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        popisLB.setText("Stisknuto: " + nazdarBT.getText());
    }
});
}

```



6.3.3. Použití anonymních vnitřních tříd a privátních metod

- odstraňuje nevýhodu nepřehlednosti anonymních tříd
- v anonymní vnitřní třídě pouze zavoláme jednu metodu – jednu řádku kódu
 - veškerá funkčnost je pak přenesena do těla této metody
 - ♦ její pojmenování jasně určuje, k jaké konkrétní komponentě se váže

```

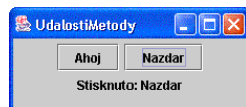
public class UdalostiMetody extends JFrame {
    ...
    private void obsluhyUdalosti() {
        ahojBT.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                ahojBT_actionPerformed(e);
            }
        });

        nazdarBT.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                nazdarBT_actionPerformed(e);
            }
        });

        void ahojBT_actionPerformed(ActionEvent e) {
            popisLB.setText("Stisknuto: " + ahojBT.getText());
        }

        void nazdarBT_actionPerformed(ActionEvent e) {
            popisLB.setText("Stisknuto: " + nazdarBT.getText());
        }
    }
}

```



6.3.4. Rozhraní posluchače má více metod

- existuje více typů událostí než jen `ActionEvent` a tím i více typů rozhraní
 - pro zaregistrování posluchače pomocí metody `addActionListener()` stačilo, aby implementoval jedinou metodu `actionPerformed()`
 - některá rozhraní mají i více metod a při implementování rozhraní je nutné implementovat **všechny** jeho metody
- problém bude vysvětlen na rozhraní `MouseListener`, které má celkem pět metod (Pozor – není to `FocusListener`)
 - cílem všech následujících programů bude, aby při najetí kurzoru myši nad tlačítko se zvýraznil jeho nápis a po opuštění byl opět původní
 - ♦ pro tuto akci navíc využijeme vnější třídu, takže jakékoliv tlačítko pak bude mít tuto funkčnost
 - ♦ rozmístění komponent a reakce na stisk tlačítek jsou zcela stejné jako u `UdalostiVnitriTrida`

6.3.4.1. Implementace prázdných metod rozhraní

- tři nepoužité metody musí být uvedeny (tj. jsou implementovány), ale mají prázdné tělo

```

class MujButton extends JButton implements MouseListener {
    Font normalniF, vyraznyF;

    MujButton(String popis) {
        super(popis);
        normalniF = this.getFont();
        vyraznyF = new Font(normalniF.getName(), Font.BOLD, normalniF.getSize()*2);
        this.addMouseListener(this);
    }

    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mouseClicked(MouseEvent e) {}

    public void mouseEntered(MouseEvent e) {
        this.setFont(vyraznyF);
    }

    public void mouseExited(MouseEvent e) {
        this.setFont(normalniF);
    }
}

public class UdalostiVnejsi extends JFrame {
    MujButton ahojBT;
    MujButton nazdarBT;
    JLabel popisLB;
}

```



6.3.4.2. Použití adaptéru

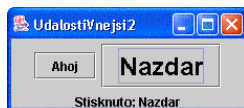
- adaptéry jsou třídy, které implementují příslušné rozhraní, ale všechny jejich metody jsou prázdné
 - existují pro všechny listenery, které mají více než jednu metodu
- pro implementování rozhraní je nutné pouze zdědit třídu adaptéru a překrýt jen tu metodu, kterou budeme skutečně potřebovat
 - všechny ostatní zdědíme jako prázdné
- tento příklad není typický

```
class MujButtonAdapter extends JButton {
    Font normalniF, vyraznyF;

    MujButtonAdapter(String popis) {
        super(popis);
        normalniF = this.getFont();
        vyraznyF = new Font(normalniF.getName(), Font.BOLD, normalniF.getSize()*2);
        this.addMouseListener(new Obsluha());
    }

    private class Obsluha extends MouseAdapter {
        public void mouseEntered(MouseEvent e) {
            MujButtonAdapter.this.setFont(vyraznyF);
        }
        public void mouseExited(MouseEvent e) {
            MujButtonAdapter.this.setFont(normalniF);
        }
    }
}
```

```
public class UdalostiVnejsi2 extends JFrame {
    MujButtonAdapter ahojBT;
    MujButtonAdapter nazdarBT;
    JLabel popisLB;
```



6.3.4.3. Adaptér a předchozí způsoby reakce na události

- adaptéry lze využít i ve všech třech dříve uvedených způsobech
- vnitřní třída

```
JButton ahojBT;
ahojBT.addMouseListener(new MLahojBT());
...
private class MLahojBT extends MouseAdapter {
    public void mouseEntered(MouseEvent e) {
        ahojBT.setFont(vyraznyF);
    }
    public void mouseExited(MouseEvent e) {
        ahojBT.setFont(normalniF);
    }
}
```

■ anonymní vnitřní třída

```
ahojBT.addMouseListener(new MouseAdapter() {
    public void mouseEntered(MouseEvent e) {
        ahojBT.setFont(vyraznyF);
    }
    public void mouseExited(MouseEvent e) {
        ahojBT.setFont(normalniF);
    }
});
```

■ metody z anonymní vnitřní třídy

```
ahojBT.addMouseListener(new MouseAdapter() {
    public void mouseEntered(MouseEvent e) {
        ahojBT_mouseEntered(e);
    }
    public void mouseExited(MouseEvent e) {
        ahojBT_mouseExited(e);
    }
});

void ahojBT_mouseEntered(MouseEvent e) {
    ahojBT.setFont(vyraznyF);
}
void ahojBT_mouseExited(MouseEvent e) {
    ahojBT.setFont(normalniF);
}
```

6.4. Jak komponentu popíšeme

6.4.1. Trocha typografické teorie

- v GUI Javy si můžeme tvar vypisovaného znaku vybrat jednoduše z (implicitně) dvanácti možností
- používá se pojem **font**, který představuje souhrn tvarů jednotlivých znaků – má mnoho významů, které je (občas) nutné rozlišovat

- z typografického hlediska je nejvyšší organizační jednotkou tzv. *rodina písem* (rodina fontů) (*font family*)

- ♦ rodina obsahuje několik – často čtyři – **řezy písem** (*font face*)

– jeden řez je vždy **základní** (*regular, plain, roman*) a zbývající jsou **vyznačovací**

– tři běžné vyznačovací řezy jsou:

- **kurzíva** (italika, *italic, cursive, oblique*)
- **tučný** (někdy též polotučný, *bold, demibold*)
- **tučná kurzíva** (*bold italic, bold oblique*)

- rodin písem je mnoho druhů, ale dají se rozřadit podle dvou základních a ihned viditelných kritérií do čtyř velkých podskupin

- první kritérium je, zda má font patky (francouzsky *serif*) či nikoliv

Patkové Bezpatkové

- druhé kritérium, je to, zda má každý znak stejnou šířku, jako znaky ostatní – pak se jedná o neproporcionální písmo (*monospaced*)

- ♦ opakem jsou písma proporcionální, kde má každý znak jinou šířku

neproporcionální proporcionální

- tato dvě kritéria se dají kombinovat

	<i>patkové</i>	<i>bezpatkové</i>
proporcionální	Times New Roman	Arial
<i>neproporcionální</i>	Courier New	Lucida Console

6.4.2. Fonty v Javě

- všechny dosud zmiňované fonty jsou **fyzické fonty**

- použijeme-li je, dostáváme se do problémů s přenositelností programů na jinou platformu
 - ♦ tam tyto fonty nemusí být nainstalovány
- proto se od JDK 1.1 používá systém tzv. symbolických jmen (**virtuální fonty** nebo **logické fonty**)
 - ♦ místo konkrétních jmen fontů (Arial nebo Helvetica) jsou používána symbolická jména
 - program je pak nezávislý na platformě
 - ♦ konečné přiřazení symbolického jména konkrétnímu fyzickému fontu je pak provedeno pomocí jejich mapování uvedeném v konfiguračním souboru JRE

– jeho jméno se nepochopitelně mění s každou verzí JDK

- JDK1.4 - `jre\lib\font.properties.CP1250`
- JDK1.5 - `jre\lib\fontconfig.properties.src`

– hledat uvedený soubor však není většinou nutné, protože JRE na konkrétní platformě by již mělo být správně nakonfigurováno

- Java umožňuje použít pět symbolických jmen rodin fontů

symbolické jméno	Linux	Windows
<code>Serif</code>	Times Roman	Times New Roman
<code>SansSerif</code>	Helvetica	Arial
<code>MonoSpaced</code>	Courier	Courier New
<code>Dialog</code>	Helvetica	Arial
<code>DialogInput</code>	Courier	Courier New

- u uvedených rodin symbolických fontů můžeme použít vždy libovolný ze všech čtyř řezů, tj. *plain, italic, bold a bolditalic*

- symbolická jména se uvádí ve zdrojovém souboru všude tam, kde by se uváděla fyzická jména, např.:

```
Font f = new Font("SansSerif", Font.ITALIC, 10);
```

- bude-li pak program spuštěn pod Windows, bude font `f` představován fyzickým fontem Arial, kdežto pod Linuxem to bude font Helvetica

6.4.3. Popis komponenty zvoleným fontem

- implicitně font `Dialog` v řezu *plain* ve velikosti 12 bodů
- pro změnu nutno vytvořit novou instanci `Font` a nastavit ji jako aktuální font pro konkrétní komponentu
 - jméno fontu je zásadně symbolické jméno
 - řez fontu konstantami `Font.PLAIN`, `Font.ITALIC`, `Font.BOLD`
 - pro tučnou kurzívu musíme použít `Font.BOLD + Font.ITALIC`
 - velikost fontu se zadává jako celé číslo
- změna fontu jedné komponenty neovlivní fonty jiných komponent

6.4.4. Využití HTML značkování

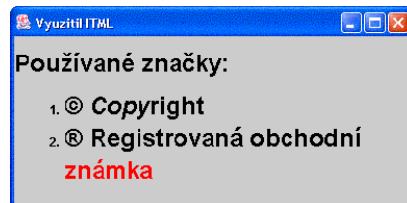
- popis komponenty je možný jen v jedné řádce a jedním fontem
 - toto omezení lze obejít použitím HTML značek
 - ♦ fungují spolehlivě pro:

- změnu velikosti a barvy
- nastavení některých řezů
- odřádkování <p>i

- výčty , ,

♦ jiné kombinace je nutno vyzkoušet

```
JLabel lab = new JLabel();
lab.setText("<html><big>Používané značky:"
+ "<ol>"
+ "<li>\u00A9 <i>Copy</i><b>right</b></li>"
+ "<li>\u00AE Registrovaná obchodní<br>"
+ "<font color=#FF0000>známka</font></li>"
+ "</ol>"
+ "</big></html>");
this.getContentPane().add(lab);
```



6.5. Zpřístupnění komponenty

■ komponenta není aktivní, tj. není pomocí ní možné vygenerovat událost, ale je stále viditelná

- metoda `void setEnabled(boolean b)`
- ♦ zda je právě přístupná – `boolean isEnabled()`

```
public void actionPerformed(ActionEvent e) {
    pristBT.setEnabled(! pristBT.isEnabled());

    skryjBT.setText(skryjBT.getText().equals("Skryj") ? "Obnov" : "Skryj");
}
```



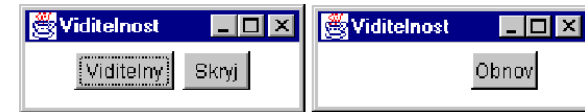
6.6. Viditelnost komponenty

■ zobrazenou komponentu lze skrýt a skrytou komponentu zviditelnit

- `void setVisible(boolean b)` a `boolean isVisible()`

```
public void actionPerformed(ActionEvent e) {
    pristBT.setVisible(! pristBT.isVisible());

    skryjBT.setText(skryjBT.getText().equals("Skryj") ? "Obnov" : "Skryj");
}
```



Kapitola 7. GUI – pokračování

7.1. Rozmísťování komponent

- za umístění jednotlivých komponent v kontejnerové komponentě zodpovídá *layout manager*

- objekt třídy `java.awt.LayoutManager` nebo častěji některé ze zděděných tříd

- každý kontejner má svůj vlastní implicitní *layout manager*, ale lze jej snadno vyměnit za jiný

- typické použití jiného manageru je např.:

```
objektKontejneru.setLayout(new FlowLayout());
```

- každý *layout manager* má jiné možnosti, k dispozici jsou např.:

- dva běžné – `FlowLayout` a `GridLayout`

- speciální – `BorderLayout`

- velmi sofistikovaný – `GridBagLayout`

- téměř nikdy nevoláme jejich metody – stačí manager vytvořit pomocí konstruktoru a použít metodu `add()` pro umístění komponent

- v krajním případě lze použití *layout managerů* vypnout a rozmísťovat komponenty na jejich absolutní pozice

- každý kontejner má svůj implicitní *layout manager*:

```
JPanel – FlowLayout
```

```
JFrame, JDialog – BorderLayout
```

- vyhovuje-li implicitní *layout manager*, nemusíme dělat vůbec nic

7.1.1. FlowLayout

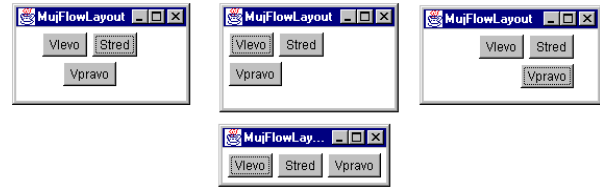
- vkládané komponenty rozmísťuje v pořadí vložení do řádky a nemění jejich velikosti implicitně centrovane s pětibodovou mezerou mezi nimi

- nevejdou-li se na řádku, pokračuje na další řádce

- ♦ výška řádky je odvozena od výšky nejvyšší komponenty

- při použití `pack()` dostaneme jen jednu řádku komponent

- požadujeme-li více řádek, musíme použít `setSize()` u kontejneru



7.1.2. GridLayout

- vkládá komponenty v pořadí vložení do předdefinované mřížky

- na další řádku přechází, když vyplní všechny sloupce na stávající řádce

- horizontální i vertikální velikosti komponent nastavuje podle velikosti největší komponenty v řádku/sloupci pro všechny řádky a sloupce jednotně

- vyplní celou plochu kontejneru



7.1.3. BorderLayout

- může umístit maximálně pět komponent

- umístění nezáleží na pořadí vložení – je přesně specifikováno „světovou stranou“

- ♦ lze použít konstanty `NORTH`, `SOUTH`, `WEST`, `EAST`, `CENTER`

- vkládané komponenty zvětšuje podle následujícího algoritmu:

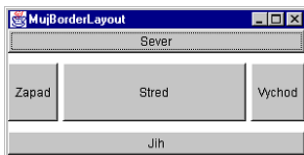
- severní a jižní – roztáhne na maximální šířku, ale ponechá výšku
- západní a východní – roztáhne do maximální výšky, ale ponechá jejich šířku
- středová – vyplní všechny zbylý prostor

- není-li v některé „světové straně“ umístěna komponenta, ostatní se do tohoto směru roztáhnou

- je matoucí různé použití metody `add()` – existují až čtyři funkční možnosti

```
this.add(new Button("Sever"), BorderLayout.NORTH);  
this.add(new Button("Jih"), "South");  
this.add(BorderLayout.WEST, new Button("Zapad"));  
this.add("East", new Button("Vychod"));
```

- zásadně používat jen první z nich



7.1.4. GridBagLayout

- nejvíce sofistikovaný (hodně možností), ale prakticky nejužívanější, protože dává většinou nejlepší výsledky
- spolupracují třídy `GridBagLayout` a `GridBagConstraints`
- využívá strukturu mřížky, každá komponenta může obsadit více řádek a/nebo sloupců
- každé komponentě lze individuálně stanovit pozici, roztažitelnost, velikost, okraje a umístění
- dodržuje-li se v kódu pořádek, je použití přímočaré, ale zdlouhavé

```
public class MujGridBagLayout extends JFrame {
    private GridBagLayout gbl;
    private GridBagConstraints gbc;
    private JPanel vnitrekPN;
    private static final int NON=GridBagConstraints.NONE;
    . . .
```

```
MujGridBagLayout() {
    this.setTitle(getClass().getName());
    this.getContentPane().add(vytvorVnitrek());
    this.setLocationRelativeTo(null);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.pack();
    this.setVisible(true);
}
```

```
private void nastav(Component c, int x, int y,
                    int s, int v,
                    double rs, double rv,
                    int vyp, int k) {
```

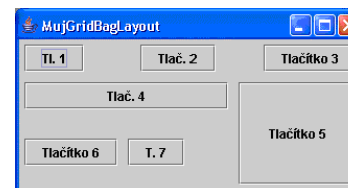
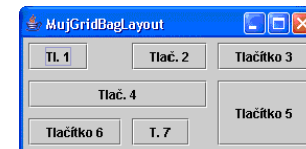
```
    gbc.gridx = x;
    gbc.gridy = y;
    gbc.gridwidth = s;
    gbc.gridheight = v;
    gbc.weightx = rs;
    gbc.weighty = rv;
    gbc.fill = vyp;
    gbc.anchor = k;
    gbl.setConstraints(c, gbc);
    vnitrekPN.add(c);
}
```

```
Container vytvorVnitrek() {
    JButton btn;
```

```
vnitrekPN = new JPanel();
gbl = new GridBagLayout();
vnitrekPN.setLayout(gbl);
gbc = new GridBagConstraints();
gbc.insets = new Insets(5, 5, 5, 5);
```

```
btn = new JButton("Tl. 1");
nastav(btn, 0, 0, 1, 1, 0.0, 0.0, NON, WES);
btn = new JButton("Tlač. 2");
nastav(btn, 1, 0, 1, 1, 0.0, 0.0, NON, CEN);
btn = new JButton("Tlačítko 3");
nastav(btn, 2, 0, 1, 1, 0.0, 0.0, NON, EAS);
btn = new JButton("Tlač. 4");
nastav(btn, 0, 1, 2, 1, 1.0, 0.0, HOR, CEN);
btn = new JButton("Tlačítko 5");
nastav(btn, 2, 1, 1, 2, 1.0, 1.0, BOT, CEN);
btn = new JButton("Tlačítko 6");
nastav(btn, 0, 2, 1, 1, 0.0, 0.0, NON, EAS);
btn = new JButton("T. 7");
nastav(btn, 1, 2, 1, 1, 0.0, 0.0, NON, WES);
```

```
    return vnitrekPN;
}
```



7.2. Komunikace pomocí Observable-Observer

- princip zasílání a příjmu událostí z GUI lze zobecnit a využít kdekoliv, kde mají objekty komunikovat bez „přímé viditelnosti“
 - třídy o sobě nemusejí vědět – zamezí se nelogickým asociacím
- vychází z návrhových vzorů (kniha Design Patterns, Elements of Reusable Object-Oriented Software; Gamma a kol.)

Výstraha

Z historických důvodů – `Observer` je rozhraní a `Observable` je třída.

■ třída vydavatele je zděděna od `java.util.Observable`

- to jí umožňuje používat (základní) metody:

- ♦ `void addObserver(Observer o)` – zaregistrování posluchače
- ♦ `void setChanged()` – nastavení příznaku změny stavu vydavatele
- ♦ `void notifyObservers()` – signál všem zaregistrovaným posluchačům, že u vydavatele došlo ke změně
- ♦ `void notifyObservers(Object parametr)` – jako předchozí, navíc předá objekt popisující změnu

■ třída odběratele musí implementovat rozhraní `java.util.Observer`, aby mohla být zaregistrována jako posluchač

- představuje to implementaci metody

```
public void update(Observable zdroj, Object param)
```

kteřá je automaticky volána po každém volání `notifyObservers()` od vydavatele

V příkladu je vydavatelem třída `Citac`, která se bude zvětšovat či zmenšovat o jedničku, což uživatel provede stiskem tlačítek `+1` nebo `-1`. Odběrateli budou dvě na sobě nezávislé třídy – první bude odděděna od `JTextField` a druhá od `JScrollbar`. Obě musí implementovat rozhraní `Observer`. Metoda `update()` z tohoto rozhraní má dva parametry. První popisuje objekt zdroje a tento parametr zde není dále využit, protože vydavatel je pouze jeden. Druhý parametr bude využit ve formě objektu třídy `Integer`, ve kterém se předá aktuální (tj. právě změněná) hodnota čítače.

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

// vydavatel
class Citac extends Observable {
    private int hodnota;

    public Citac(int hodnota) {
        setHodnota(hodnota);
    }

    public void setHodnota(int hodnota) {
        this.hodnota = hodnota;
        setChanged(); // došlo ke změně
        // změna je reprezentována novým objektem
        notifyObservers(new Integer(hodnota));
    }

    public void plusJedna() {
        setHodnota(++hodnota);
    }
}
```

```
public void minusJedna() {
    setHodnota(--hodnota);
}

// odběratel
class CitacTextField extends JTextField implements Observer {
    CitacTextField(int hodnota) {
        this.setColumns(6);
        this.setEditable(false);
        this.setHorizontalAlignment(JTextField.CENTER);
        this.setText("" + hodnota);
    }

    public void update(Observable o, Object arg) {
        this.setText(arg.toString());
    }
}

// odběratel
class CitacSlider extends JSlider implements Observer {
    CitacSlider(int hodnota, int min, int max) {
        super(JSlider.HORIZONTAL, min, max, hodnota);
        this.setMajorTickSpacing(2);
        this.setMinorTickSpacing(1);
        this.setPaintTicks(true);
        this.setPaintLabels(true);
    }

    public void update(Observable o, Object arg) {
        int pozice = ((Integer) arg).intValue();
        this.setValue(pozice);
    }
}

// hlavní okno aplikace
public class MujObserver extends JFrame {
    Citac citac;
    CitacTextField ctf;
    CitacSlider csl;
    JButton plusBT, minusBT;

    MujObserver() {
        this.setTitle(getClass().getName());
        this.getContentPane().add(vytvorVnitrek());
        obsluhyUdalosti();
        . . .
    }

    Container vytvorVnitrek() {
        . . .
    }

    private void obsluhyUdalosti() {

```

```

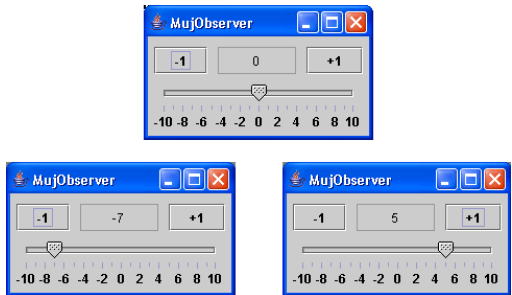
minusBT.addActionListener(new MinusBT());
plusBT.addActionListener(new PlusBT());
citac = new Citac(0);
citac.addObserver(ctf);
citac.addObserver(csl);
}

private class PlusBT implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        citac.plusJedna();
    }
}

private class MinusBT implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        citac.minusJedna();
    }
}

public static void main(String[] args) {
    new MujObserver();
}

```



7.3. Použití ikon

7.3.1. Základní informace

- nejedná se přímo o použití grafiky, ale o umístění obrázku na nejčastěji `JButton`
 - výhodou je, že všechna známá práce s buttonem zůstává zcela stejná
- používáme ikony, tj. objekty s obrázky pevné velikosti splňující rozhraní `javax.swing.Icon`
- pro zjednodušení práce se používá třída `javax.swing.ImageIcon`, která umí vykreslit obrázky ve formátech GIF, JPEG a PNG
- ikony může doprovázet současně zobrazený text, ale toto se používá zřídka

- většinou je ikona doprovázena bublinkovou nápovědou `javax.swing.JToolTip`, což je něco jiného (viz dále)

7.3.2. Problémy při natažení obrázku

```

import java.net.*;
import java.awt.*;
import javax.swing.*;

public class NacteniIkony {
    public static void main(String[] args) throws Exception {
        JFrame oknoF = new JFrame("NacteniIkony");
        oknoF.setLayout(new FlowLayout());
        ImageIcon krizekIK = new ImageIcon("krizek.jpg");
        JButton krizekBT = new JButton("krizek", krizekIK);
        ImageIcon koleckoIK = new ImageIcon("ikony/kolecko.jpg");
        JButton koleckoBT = new JButton(koleckoIK);
        koleckoBT.setMargin(new Insets(0,0,0,0));
        URL trojuhURL = NacteniIkony.class.getResource("ikony/trojuh.jpg");
        ImageIcon trojuhIK = new ImageIcon(trojuhURL);
        JButton trojuhBT = new JButton(trojuhIK);
        trojuhBT.setToolTipText("trojúhelník");

        oknoF.add(krizekBT);
        oknoF.add(koleckoBT);
        oknoF.add(trojuhBT);
        oknoF.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        oknoF.setSize(320, 90);
        oknoF.setVisible(true);
    }
}

```

1. pokud bude program spuštěn z `.class` souboru, lze obrázek natáhnout

```
ImageIcon krizekIK = new ImageIcon("krizek.jpg");
```

2. jako oddělovač adresářů se používá `/`

```
ImageIcon koleckoIK = new ImageIcon("ikony/kolecko.jpg");
```

u takto natahovaných souborů nezáleží na velikosti písmen ve jménu `krizek.jpg` je stejné jako `krizek.JPG`

3. pro spuštění z `.jar` souboru toto jednoduché natažení nefunguje a je třeba použít instanci `java.net.URL`

```
URL trojuhURL = NacteniIkony.class.getResource("ikony/trojuh.jpg");
```

v případě, že URL získáváme v instanční metodě, lze použít

```
URL trojuhURL = getClass().getResource("ikony/trojuh.jpg");
```

Výstraha

při použití URL záleží na velikosti písmen ve jménu souboru `krizek.jpg` je různé od `krizek.JPG`

4. u `JButton`u se ještě přidají k obrázku okraje, kterých se lze zbavit:

```
koleckoBT.setMargin(new Insets(0,0,0,0));
```

5. popis ikony je něco jiného než tooltip (bublinková nápověda)

```
JButton krizekBT = new JButton("krizek", krizekIK);
trojuhBT.setToolTipText("trojúhelník");
```

popisu ikony (současně s obrázkem) se snažíme zbavit – viz dále

6. pro soubory obrázků ikon je lepší použít formát GIF než JPEG, zejména kreslíme-li je sami

7.3.3. Seskupení ikon do nástrojové lišty (toolbar)

■ použijeme komponentu `JToolBar`

```
import java.net.*;
import java.awt.*;
import javax.swing.*;
```

```
public class ToolBarVytvoreni {
    public ToolBarVytvoreni() {
        JFrame oknoF = new JFrame("ToolBarVytvoreni");
        oknoF.setLayout(new BorderLayout());
        oknoF.add(vytvorToolBar(), BorderLayout.PAGE_START);
        oknoF.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        oknoF.pack();
        oknoF.setVisible(true);
    }
}
```

```
ImageIcon nactiObrazekIkony(String jmeno) {
    URLClassLoader urlLoader =
        (URLClassLoader) this.getClass().getClassLoader();
    URL ikonaURL = urlLoader.findResource("ikony/" + jmeno + ".gif");
    ImageIcon ikonaIK = new ImageIcon(ikonaURL);
    return ikonaIK;
}
```

```
JButton vytvorIkonu(String jmenoObrazkuIkony, String toolTip) {
    JButton tlac = new JButton(nactiObrazekIkony(jmenoObrazkuIkony));
    if (tlac.getIcon() != null) {
        tlac.setText(""); // je-li obrazek, není třeba popis
    }
    tlac.setMargin(new Insets(0,0,0,0));
    tlac.setToolTipText(toolTip);
    return tlac;
}
```

```
JToolBar vytvorToolBar() {
    JToolBar hlavniTB = new JToolBar("Jméno ToolBaru");
    JButton tl = vytvorIkonu("krizek", "křížek - tool tip");
    hlavniTB.add(tl);
    hlavniTB.add(vytvorIkonu("kolecko", "kolečko - tool tip"));

    hlavniTB.addSeparator(new Dimension(100, 20));
    hlavniTB.add(vytvorIkonu("trojuh", "trojúhelník - tool tip"));

    hlavniTB.setFloatable(false);
    hlavniTB.setRollover(true);
    return hlavniTB;
}

public static void main(String[] args) throws Exception {
    new ToolBarVytvoreni();
}
```

1. aby `JToolBar` správně pracoval, je vhodné použít jako layout manager hlavního okna `BorderLayout` a `JToolBar` umístit nahoru

```
oknoF.add(vytvorToolBar(), BorderLayout.PAGE_START);
```

2. pojmenování tool baru není nutné, ale pokud bude přemístitelný (*floatable*), je to velmi vhodné

```
JToolBar hlavniTB = new JToolBar("Jméno ToolBaru");
```

3. jednotlivá tlačítka (ikony) můžeme nejjednodušeji v pořadí výskytu

```
hlavniTB.add(tl);
```

4. mezi ikony lze umístit oddělovací mezeru, jejíž rozměr lze volit

```
hlavniTB.addSeparator();
```

nebo

```
hlavniTB.addSeparator(new Dimension(100, 20));
```

5. tool bar je implicitně nastaven jako přemístitelný (*floatable*), což znamená možnost přesunout jej dynamicky doleva, doprava nebo dolů v hlavním okénku

další možnost je „vytáhnout“ jej zcela mimo hlavní okénko

■ pak bude mít nově vzniklé okénko jméno, které jsme použili v konstruktoru `JToolBar`

■ zpět do hlavního okénka se dostane zavřením

6. vlastnost přemístitelnosti lze zakázat (většinou rozumná volba)

```
hlavniTB.setFloatable(false);
```

7. velmi užitečná vlastnost se dá zapnout pomocí

```
hlavniTB.setRollover(true);
```

pak by byla orámovaná pouze ikona s fokusem

bohužel: „*The implementation of a look and feel may choose to ignore this property.*“

7.3.4. Aktivita ikon a souvisejících položek v menu

■ ve většině případů jsou ikony použity pro několik málo nejčastěji používaných akcí, přičemž seznam všech možných akcí je dostupný menu

- pak je velmi vhodné zařídit, aby nebylo nutné psát dvě víceméně stejné obsluhy událostí – jednu pro ikonu a druhou pro položku v menu

- dalším problémem by mohla být synchronizace kontextového znepřístupňování ikon a položek menu

■ oba problémy lze snadno vyřešit tím, že nepoužíváme běžný způsob obsluh událostí, ale použijeme potomka třídy `AbstractAction`, který splňuje rozhraní `Action`

```
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ToolbarAMenuAktivita {
    JTextField vystupTF;
    SpolecnaAkce spolecnaAkce;

    public ToolbarAMenuAktivita() {
        JFrame oknoF = new JFrame("ToolbarAMenuAktivita");
        oknoF.setLayout(new BorderLayout());
        spolecnaAkce = new SpolecnaAkce();
        oknoF.setJMenuBar(vytvorMenuBar());

        oknoF.add(vytvorToolBar(), BorderLayout.PAGE_START);
        vystupTF = new JTextField();
        vystupTF.setHorizontalAlignment(SwingConstants.CENTER);
        oknoF.add(vystupTF, BorderLayout.CENTER);
        oknoF.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        oknoF.pack();
        oknoF.setVisible(true);
    }

    class SpolecnaAkce extends AbstractAction {
        int pocet = 0;
        public SpolecnaAkce() {
            ImageIcon ikona = nactiObrazekIkony("krizek");
            putValue(NAME, "Křížek - jméno");
            putValue(SMALL_ICON, ikona);
            putValue(SHORT_DESCRIPTION, "Křížek - text tooltipu");
            putValue(MNEMONIC_KEY, new Integer(KeyEvent.VK_K));
            putValue(ACCELERATOR_KEY, KeyStroke.getKeyStroke(KeyEvent.VK_W,
                ActionEvent.CTRL_MASK));
        }
    }
}
```

```
public void actionPerformed(ActionEvent e) {
    pocet++;
    vystupTF.setText("" + pocet + ". akce");
}

ImageIcon nactiObrazekIkony(String jmeno) {
    URLClassLoader urlLoader = (
        URLClassLoader) this.getClass().getClassLoader();

    URL ikonaURL = urlLoader.findResource("ikony/" + jmeno + ".gif");
    ImageIcon ikonaIK = new ImageIcon(ikonaURL);
    return ikonaIK;
}

JButton vytvorIkonu(Action spolecnaAkceSMenu) {
    JButton tlac = new JButton(spolecnaAkceSMenu);
    if (tlac.getIcon() != null) {
        tlac.setText(""); // je-li obrazek, není třeba popis
    }
    tlac.setMargin(new Insets(0,0,0,0));
    return tlac;
}

JToolBar vytvorToolBar() {
    JToolBar hlavniTB = new JToolBar("Jméno ToolBaru");
    hlavniTB.add(vytvorIkonu(spolecnaAkce));
    hlavniTB.addSeparator();

    final JCheckBox povoleniCHB = new JCheckBox("Akce křížku povolena", true);
    povoleniCHB.setMnemonic(KeyEvent.VK_A);
    povoleniCHB.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            boolean b = povoleniCHB.isSelected();
            spolecnaAkce.setEnabled(b);
        }
    });
    hlavniTB.add(povoleniCHB);

    return hlavniTB;
}

JMenuBar vytvorMenuBar() {
    JMenuBar menuBar = new JMenuBar();
    JMenu vodorovneMenu = new JMenu("Menu");
    vodorovneMenu.setMnemonic(KeyEvent.VK_M);

    JMenuItem polozkaSvislehoMenu = new JMenuItem(spolecnaAkce);

    // je-li popis, nepoužívat ikonu
    polozkaSvislehoMenu.setIcon(null);
    vodorovneMenu.add(polozkaSvislehoMenu);
    vodorovneMenu.add(new JMenuItem("nefunkční položka"));
}
```

```

        menuBar.add(vodorovneMenu);
        return menuBar;
    }

    public static void main(String[] args) throws Exception {
        new ToolBarAMenuAktivita();
    }
}

```

Základem společné obsluhy a popisu je `javax.swing.AbstractAction`, která se zdědí

```
class SpolecnaAkce extends AbstractAction {
```

■ zde se na jednom místě (typicky v konstruktoru) přehledně nastaví následující vlastnosti:

- `NAME` – popis použitý na řádce v menu a případně i jako popis ikony, zde vypnuto pomocí

```

        if (tlac.getIcon() != null) {
            tlac.setText("");
        }

```

- `SMALL_ICON` – obrázek ikony na tlačítku, případně může být i před popisem v menu, zde vypnuto pomocí

```
polozkaSvislehoMenu.setIcon(null);
```

- `SHORT_DESCRIPTION` – text tool tipu, který se zobrazí nad ikonou i nad položkou menu
- `MNEMONIC_KEY` – zkratková klávesa, typicky písmena A až Z, na velikosti nezáleží
 - ♦ má různý význam v ikoně – volá se odkudkoliv přes Alt+písmeno a zobrazuje se v tool tipu
 - ♦ v menu podtrhává stejné písmeno v popisu, samotná klávesa slouží jako rychlý výběr z rozvinutého menu
 - *mnemonic key* slouží stejně i pro ostatní komponenty (`JCheckBox`)
- `ACCELERATOR_KEY` – použití jen v položkách menu, kde:
 - ♦ vypisuje se za jménem položky a v tool tipu
 - ♦ možnost volit libovolné klávesy (i číslice nebo funkční klávesy)
 - ♦ možnost volit modifikátory a jejich kombinace (Ctrl, Alt, Shift)
 - ♦ pomocí modifikátoru (Ctrl+W) lze zavolat odkudkoliv

■ dále je nutné napsat obsluhu události

```
public void actionPerformed(ActionEvent e)
```

■ nejdůležitější zděděná metoda této třídy je

```
void setEnabled(boolean newValue)
```

která zpřístupní / znepřístupní současně ikonu i položku menu

Poznámka

do toolbaru lze umístit i jinou komponentu, než ikonu, zde

```
final JCheckBox povoleniCHB = new JCheckBox("Akce křížku povolena", true);
```

není to většinou rozumný nápad, rozumné použití je vidět ve Wordu, kde jsou v toolbaru např. nastavení fontů v `JTextField`

Kapitola 8. Grafika

Poznámka k literatuře:

- informací v TIJ3 je velmi málo a zastaralé
- informace v Tut nejsou úplně nejlépe organizovány

Základní charakteristika našeho snažení:

- aktivita, kdy sami chceme vykreslit nějaký grafický obrazec (úsečku, kružnici, atd.) nebo vypsát text
 - nejedná se o vykreslování jednotlivých komponent, např. zobrazování tlačítek apod.
 - ♦ to zajišťuje Swing automaticky

8.1. Jak se komponenty překreslují

- začíná se od hierarchicky nejvyššího kontejneru (často `JFrame`) a postupuje k nižším
 - o překreslování se většinou nemusí žádat, stačí jen komponentu změnit
 - ♦ např. `setText()` způsobí přepsání textu a i případnou změnu velikosti komponenty
 - to je zařízeno tak, že změna vzhledu komponenty způsobí automaticky vyvolání metody `repaint()` z `java.awt.Component`
 - ♦ ta zařídí, že požadavek na vykreslení je umístěn do fronty čekajících a vykreslí se metodou `paintComponent()`, jakmile na něj přijde řada
 - ♦ navíc, změní-li se pozice nebo velikost komponenty, je před `repaint()` zavolána ještě `validate()`
- je třeba si uvědomit, že ve Swingu jsou všechna volání součástí jednoho vlákna včetně volání `repaint()` a včetně reakcí na události
 - pokud jedna část trvá dlouho, ostatní musejí čekat, až doběhne
- aby bylo vykreslování co nejplynulejší (netrhané), používá se implicitně metoda dvojího bufferování (*double-buffered*)
 - komponenty se překreslují do bufferu v pozadí a až když je hotov, jednorázově se přepne na obrazovku
- existuje jednoduchý recept na vylepšení rychlosti vykreslování – nastavit neprůhledné komponenty (těch je prakticky v programu naprostá většina, ne-li všechny) na skutečně neprůhledné (*opaque*)
 - pro to slouží metoda `setOpaque(boolean isOpaque)` z `javax.swing.JComponent`
 - standardní nastavení je totiž `false` (= průhledné) což při vykreslování způsobí, že se musí kompletně vykreslit i komponenty, které jsou pod průhlednou komponentou
 - ♦ u některých komponent ale nastavení záleží na použitém L&F

- je-li komponenta průhledná, nevykresluje se její pozadí nastavené pomocí `setBackground()`
 - ♦ naopak okraje komponenty, jsou-li vykreslovány, se vykreslují vždy

8.2. Grafické možnosti API

- jsou založené na primitivní (= minimální možnosti) třídě `java.awt.Graphics`
 - od JKD 1.2 je k dispozici potomek `java.awt.Graphics2D`
- metody třídy `Graphics2D` umožňují snadno:
 - kreslit čáry libovolné tloušťky
 - vyplňovat obrysy barvami, přechody, texturami
 - posouvat, rotovat, zmenšovat, zkosit grafická primitiva i text
 - pracovat s obrázky atp.

8.2.1. Kam kreslit a jak

- lze kreslit na jakýkoliv objekt zděděný od `javax.swing.JComponent`
 - nelze kreslit na `JFrame`
- pro kreslení je třeba překrýt metodu`protected void paintComponent(Graphics g)`
- to znamená, že musíme vytvořit vlastní třídu zděděnou od `JComponent` (abstraktní třída)
 - prakticky většinou dědíme od `JPanel`
- parametrem je grafický objekt (grafický kontext), který se získá „automaticky“
 - o jeho vytvoření se nemusíme starat, protože tuto metodu nikdy přímo nevoláme
 - ♦ vyvolává se metodou `repaint()`, která dokáže vynutit okamžik kreslení

8.2.2. Souřadnicový systém

- počátek souřadnic `[0, 0]` (typu `int` = počet pixelů) je v levém horním rohu
 - souřadnice `x` (šířka, *width*) roste doprava
 - souřadnice `y` (výška, *height*) roste dolů
- max. hodnota `x` a `y` (velikost prostoru, kam lze kreslit = „plátna“) je dána velikostí komponenty`x = getWidth()-1, y = getHeight()-1`
 - odečtení 1 je nutné, protože souřadnice jsou od 0

- protože se u komponenty mohou vyskytnout okraje, je vhodné použít

```
Insets insets = getInsets();
int minX = insets.left;
int minY = insets.top;
int maxX = getWidth() - insets.left - insets.right - 1;
int maxY = getHeight() - insets.top - insets.bottom - 1;
```

8.2.3. Práce s grafickým kontextem

- objekt třídy `Graphics` z historických důvodů předávaný do `paintComponent()` téměř nikdy nepoužijeme přímo

- přetypovává se na `Graphics2D`

```
Graphics2D g2 = (Graphics2D) g;
```

- tato třída má mnohem větší možnosti

- nese si s sebou následující defaultní nastavení renderovacích atributů

- `Paint` – barva popředí komponenty (černá)
- `Font` – font popisu komponenty (`Dialog`, `PLAIN`, `12 pt`)
- `Stroke` – typ čáry – tloušťka 1, nepřerušovaná čára
- `Transform` – žádná transformace

podrobnosti a ukázky viz v `tutorial\2d\overview\rendering.html`

Poznámka

1. objekt třídy `Graphics` předávaný do `paintComponent()` je pak také předáván do automaticky volané metody `paintBorder()`

pokud tedy změníme nějaké nastavení, promítne se i do případného ohraničení

pokud to nechceme, používáme:

```
Graphics2D g2 = (Graphics2D) g.create(); // kopie
// nějaké nastavení nad kopií
g2.translate(x, y);
...
g2.dispose(); // zrušení kopie
```

2. občas je vidět jako první příkaz metody `paintComponent()` volání `super.paintComponent(g);`

- tím se zajistí, že se nejdříve vykreslí vše, co bylo požadováno v předkovi
- pokud na kreslicí komponentu jen jednorázově kreslíme, pak je tento příkaz zbytečný

- má-li komponenta reagovat na `repaint()`, je tento příkaz vhodný (viz dále)

Příklad 8.1. Ukázka vytvoření kreslicí komponenty a starého (z `Graphics`) a nového (z `Graphics2D`) způsobu kreslení

```
import javax.swing.*;
import java.awt.*;
import java.awt.geom.*;

class MojeKreslicíKomponenta extends JPanel {
    MojeKreslicíKomponenta() {
        setOpaque(true);
    }

    public Insets getInsets() {
        return new Insets(10, 10, 10, 10);
    }

    protected void paintComponent(Graphics g) {
        Insets insets = getInsets();
        int minX = insets.left;
        int minY = insets.top;
        int maxX = getWidth() - insets.left - insets.right - 1;
        int maxY = getHeight() - insets.top - insets.bottom - 1;
        // starý způsob z Graphics
        g.setColor(Color.yellow);
        g.fillRect(minX, minY, maxX, maxY);
        Graphics2D g2 = (Graphics2D) g;
        // nový způsob z Graphics2D
        g2.setPaint(Color.black);
        g2.draw(new Line2D.Double(0, 0, maxX, maxY));
    }
}

public class GrafikaZaklad {
    public static void main(String[] args) {
        JFrame oknoF = new JFrame("GrafikaZaklad");
        oknoF.add(new MojeKreslicíKomponenta());
        oknoF.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        oknoF.setSize(250, 250);
        oknoF.setVisible(true);
    }
}
```

8.3. Základní grafická primitiva

- z `java.awt.geom`
- vykreslují se obrysové přednastavenou barvou pomocí
`g2.draw(Shape s)`
- nebo vyplněné (jsou-li uzavřená) pomocí


```
g2.fill(Shape s)
```

■ od Shape jsou zděděny např.

- přímka – Line2D
- obdélníkové tvary – Rectangle2D, RoundRectangle2D, Ellipse2D, Arc2D
- křivky – CubicCurve2D, QuadCurve2D
- obecný tvar – GeneralPath

```
tutorial\2d\overview\shapes.html
```

■ toto jsou ale abstraktní třídy, které mají vždy dvě vnitřní třídy .Double a .Float

- např. Line2D.Double a Line2D.Float
- ty používají souřadnice x a y buď typu double nebo float

■ prakticky je vykreslení přímky např.

```
g2.draw(new Line2D.Double(0.0, 0.0, 10.0, 10.0));
```

Poznámka

Přestože parametry Line2D.Double() mají být typu double, je možné používat typ int, protože dojde k automatické konverzi na double.

```
g2.draw(new Line2D.Double(0, 0, 10, 10));
```

■ vyplnění obdélníka

```
g2.fill(new Rectangle2D.Double(0.0, 0.0, 10.0, 10.0));
```

■ objekty primitiv nemusíme pokaždé vytvářet znovu (časově náročné), mají dostatečné množství nastavovacích metod, např.

```
setLine(double X1, double Y1, double X2, double Y2)
setLine(Line2D jinaLine)
setLine(Point2D p1, Point2D p2)
```

8.3.1. Porovnání starého a nového způsobu kreslení

■ třída Graphics (od které je zděděna Graphics2D) dává pro vykreslování primitiv k dispozici použitelné metody typu drawLine(), drawRect(), fillRect(), ...

- většinou to ale není výhodné, protože fill() a draw() z Graphics2D dokáží využít polymorfismu

■ dále je třeba si uvědomit, že „starým způsobem“ vykreslená primitiva byly jen „pixels na obrazovce“

- primitiva z java.awt.geom jsou kromě toho navíc objekty v paměti s mnoha užitečnými metodami, např.:

```
♦ boolean intersectsLine(Line2D l)
```

– tj. test, zda úsečka kříží jinou úsečku

```
♦ nebo int outcode(double x, double y)
```

– která pro obdélník určí, kde testovaný bod leží – uvnitř nebo vně (vlevo, vpravo, nahoře, dole)

♦ vytváříme-li aplikaci, která pouze jednorázově nevykresluje, jsou tyto metody velkou pomocí

8.4. Nastavení parametrů barvy a čáry

■ jedná se o barvu obrysu či výplně a čáru obrysu

- jsou to parametry typu rozhraní Paint (barva) a Stroke (čára) nastavované metodami třídy Graphics2D

```
void setPaint(Paint paint) a void setStroke(Stroke s)
```

8.4.1. Rozhraní Paint

■ toto rozhraní implementují třídy:

- Color – jednoduchá barva
- GradientPaint – barevný přechod
- TexturePaint – textura

■ vlastnost se nastavuje pomocí g2.setPaint(Paint p);

■ Color má mnoho možností (viz dokumentace)

- typické použití je Color(int r, int g, int b) pro vytvoření libovolné neprůhledné *true-color* barvy

- nebo použití jedné ze 13 již přednastavených „základních“ konstantních barev

```
♦ black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, yellow
```

```
♦ použití je Color.red a lze použít i velká písmena Color.RED
```

- dále lze používat i průhledné barvy (*transparency*) atd.

■ GradientPaint má typický způsob použití:

```
GradientPaint(float x1, float y1, Color color1,
              float x2, float y2, Color color2)
```

- např. postupný vodorovný přechod od červené ke žluté:

```
GradientPaint redToYellow =
    new GradientPaint(x, y, Color.red,
                     x + sirka, y, Color.yellow);
```

8.4.2. Rozhraní Stroke

- implementuje třída `BasicStroke`
- vlastnost se nastavuje pomocí `g2.setStroke(Stroke s);`

- lze nastavit zejména tloušťku čáry (*width*)
 - ♦ typické použití: `new BasicStroke(2.0f);`




Výstraha

parametr je typu `float`, takže pouze výraz `(2.0)` způsobí chybu při překladu




- pokud je čára přerušovaná, lze ji libovolně nakonfigurovat použitím třídy:

```
BasicStroke(float    width,
            int      cap,
            int      join,
            float    miterlimit,
            float[]  dash,
            float    dash_phase)
```

- `cap` tvar zakončení

-  `CAP_BUTT` – pravoúhlé bez přesahu
-  `CAP_ROUND` – zakulacené s přesahem
-  `CAP_SQUARE` – pravoúhlé s přesahem

- `join` – typ napojení dvou čar

-  `JOIN_BEVEL` – useknuté
-  `JOIN_MITER` – do špičky
-  `JOIN_ROUND` – do kulata

- `miterlimit` – maximální délka špičatého napojení

- `dash` – vzor přerušované čáry

- přerušovaná: `float[] dash = {10.0f};`
- čerchovaná: `float[] dash = {10.0f, 3.0f, 2.0f, 3.0f};`

- `dash_phase` – offset v počtu bodů, kdy má začít přerušovaná čára

```
float[] dash = {10.0f};
BasicStroke dashed =
    new BasicStroke(1.0f,
```

```
BasicStroke.CAP_BUTT,
BasicStroke.JOIN_MITER,
10.0f, dash, 0.0f);
```

Poznámka

Používáme-li základní primitiva a tenké čáry, je vhodné zapnout antialiasing, který často významně vylepší vzhled obrázku

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                    RenderingHints.VALUE_ANTIALIAS_ON);
```

Příklad 8.2. Ukázka grafických primitiv

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class Primitiva2D extends JPanel {
    public void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);

        Dimension d = getSize();
        int gridWidth = d.width / 6;
        int gridHeight = d.height / 2;

        g2.setPaint(Color.white);
        g2.fill(new Rectangle2D.Double(0, 0, d.width, d.height));

        Font font = new Font("SansSerif", Font.PLAIN, 8);
        FontMetrics fontMetrics = g2.getFontMetrics(font);

        g2.setPaint(Color.lightGray);
        g2.draw3DRect(0, 0, d.width - 1, d.height - 1, true);
        g2.draw3DRect(3, 3, d.width - 7, d.height - 7, false);
        g2.setPaint(Color.black);

        int x = 5;
        int y = 8;
        int rectWidth = gridWidth - 2 * x;
        int stringY = gridHeight - 3 - fontMetrics.getDescent();
        int rectHeight = stringY - fontMetrics.getMaxAscent() - y - 5;

        // Line2D
        g2.draw(new Line2D.Double(x, y+rectHeight-1,
            x + rectWidth, y));
        g2.drawString("Line2D", x, stringY);
        x += gridWidth;

        // Rectangle2D
        g2.setStroke(new BasicStroke(2.0f));
        g2.draw(new Rectangle2D.Double(x, y, rectWidth, rectHeight));
        g2.drawString("Rectangle2D", x, stringY);
        x += gridWidth;

        // RoundRectangle2D
        // float vzor[] = {10.0f, 3.0f, 2.0f, 3.0f};
        float vzor[] = {10.0f};
        BasicStroke dashed = new
            BasicStroke(1.0f,
                BasicStroke.CAP_BUTT,
                BasicStroke.JOIN_MITER,
                10.0f, vzor, 0.0f);
```

```
g2.setStroke(dashed);
g2.draw(new RoundRectangle2D.Double(x, y, rectWidth,
    rectHeight, 10, 10));
g2.drawString("RoundRectangle2D", x, stringY);
x += gridWidth;

// Arc2D
g2.setStroke(new BasicStroke(8.0f,
    BasicStroke.CAP_BUTT,
    BasicStroke.JOIN_MITER));
g2.draw(new Arc2D.Double(x, y, rectWidth, rectHeight,
    90, 135, Arc2D.OPEN));
g2.drawString("Arc2D", x, stringY);
x += gridWidth;

// Ellipse2D
g2.setStroke(new BasicStroke(2.0f));
g2.draw(new Ellipse2D.Double(x, y,
    rectWidth, rectHeight));
g2.drawString("Ellipse2D", x, stringY);
x += gridWidth;

// GeneralPath uzavřena - polygon
int x1Points[] = {x, x+rectWidth, x, x+rectWidth};
int y1Points[] = {y, y+rectHeight, y+rectHeight, y};
GeneralPath polygon = new
    GeneralPath(GeneralPath.WIND_EVEN_ODD,
        x1Points.length);
polygon.moveTo(x1Points[0], y1Points[0]);
for (int i = 1; i < x1Points.length; i++) {
    polygon.lineTo(x1Points[i], y1Points[i]);
}
polygon.closePath();
g2.draw(polygon);
g2.drawString("GeneralPath", x, stringY);

// druhý radek
x = 5;
y += gridHeight;
stringY += gridHeight;

// GeneralPath otevřena - polyline
int x2Points[] = {x, x+rectWidth, x, x+rectWidth};
int y2Points[] = {y, y+rectHeight, y+rectHeight, y};
GeneralPath polyline = new
    GeneralPath(GeneralPath.WIND_EVEN_ODD,
        x2Points.length);
polyline.moveTo(x2Points[0], y2Points[0]);
for (int i = 1; i < x2Points.length; i++) {
    polyline.lineTo(x2Points[i], y2Points[i]);
}
g2.draw(polyline);
g2.drawString("GeneralPath otevřená", x, stringY);
```

```

x += gridWidth;

// Rectangle2D
g2.setPaint(Color.red);
g2.fill(new Rectangle2D.Double(x, y, rectWidth, rectHeight));
g2.setPaint(Color.black);
g2.drawString("Rectangle2D", x, stringY);
x += gridWidth;

// RoundRectangle2D
GradientPaint redToYellow = new GradientPaint(x, y, Color.red,
                                              x + rectWidth, y, Color.yellow);

g2.setPaint(redToYellow);
g2.fill(new RoundRectangle2D.Double(x, y, rectWidth,
                                   rectHeight, 10, 10));

g2.setPaint(Color.black);
g2.drawString("RoundRectangle2D", x, stringY);
x += gridWidth;

// Arc2D
g2.setPaint(Color.red);
g2.fill(new Arc2D.Double(x, y, rectWidth, rectHeight,
                        90, 135, Arc2D.OPEN));

g2.setPaint(Color.black);
g2.drawString("Arc2D", x, stringY);
x += gridWidth;

// Ellipse2D
redToYellow = new GradientPaint(x, y, Color.red,
                               x+rectWidth, y, Color.yellow);

g2.setPaint(redToYellow);
g2.fill (new Ellipse2D.Double(x, y, rectWidth, rectHeight));
g2.setPaint(Color.black);
g2.drawString("Ellipse2D", x, stringY);
x += gridWidth;

// GeneralPath
int x3Points[] = {x, x+rectWidth, x, x+rectWidth};
int y3Points[] = {y, y+rectHeight, y+rectHeight, y};
GeneralPath filledPolygon = new
    GeneralPath(GeneralPath.WIND_EVEN_ODD,
               x3Points.length);

filledPolygon.moveTo(x3Points[0], y3Points[0]);
for (int i = 1; i < x3Points.length; i++) {
    filledPolygon.lineTo(x3Points[i], y3Points[i]);
}
filledPolygon.closePath();
g2.setPaint(Color.red);
g2.fill(filledPolygon);
g2.setPaint(Color.black);
g2.draw(filledPolygon);
g2.drawString("GeneralPath", x, stringY);
}

```

```

public static void main(String s[]) {
    JFrame f = new JFrame("Primitiva2D");
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.getContentPane().add(new Primitiva2D());
    f.setSize(750, 300);
    f.show();
}
}

```

Poznámka

pokud chceme tvar vyplnit a navíc orámovat, musíme použít nejdříve `fill()` a pak `draw()`

8.5. Křivky jako grafická primitiva

■ existují dvě

- **QuadCurve2D** – křivka určená dvěma koncovými body a jedním řídícím bodem
 - ♦ má jen jedno zakřivení
 - ♦ `tutorial\2d\display\Quad.html`
- **CubicCurve2D** – Beziérová křivka určená dvěma koncovými body a dvěma řídícími body
 - ♦ má dvě zakřivení
 - ♦ `tutorial\2d\display\Cubic.html`

8.5.1. Obecná křivka – GeneralPath

- může být uzavřená (polygon) nebo otevřená (polyline)
- typicky se vytváří tak, že se nejdříve definují pole souřadnic `x` a `y`
- pak se vytvoří objekt třídy `GeneralPath`, kterému se určí způsob vyhodnocení, zda je nějaký bod uvnitř (*winding rule*) – většinou se použije `WIND_EVEN_ODD`
 - druhý parametr je počet bodů, ze kterých bude křivka skládat
- třetím krokem je nastavení výchozího bodu pomocí `moveTo()`
- pak následuje spojení bodů pomocí
 - `lineTo()` – spojení úsečkou
 - `quadTo()` – spojení křivkou
- poslední krok může být uzavření křivky pomocí `closePath()`

8.6. Afinní transformace

■ Graphics2D poskytuje možnosti transformací

- `translate` – posun souřadnic
- `rotate` – natočení
- `scale` – změna měřítka
- `shear` – zkosení

■ tyto transformace lze slučovat

■ lze je provést nad celým kreslícím prostorem jednotlivě, pomocí metod z Graphics2D

- `translate()`
- `rotate()`
- `scale()`
- `shear()`

Příklad 8.3.

```
public class Transformace extends JPanel {
    protected void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;
        Dimension d = getSize();
        int sirka = d.width / 3;
        int vyska = d.height / 3;
        g2.setStroke(new BasicStroke(10.0f));
        // g2.translate(sirka, vyska);
        // g2.rotate(Math.toRadians(45));
        g2.rotate(Math.toRadians(45), 1.5 * sirka, 1.5 * vyska);
        // g2.scale(0.5, 0.5);
        // g2.shear(0.5, 0.0);
        g2.setPaint(Color.yellow);
        g2.draw(new Rectangle2D.Double(sirka, vyska,
                                     sirka, vyska));

        g2.setPaint(Color.black);
        g2.draw(new Line2D.Double(sirka, 2 * vyska,
                                2 * sirka, vyska));
    }
}
```

■ druhou možností je připravit si objekt třídy AffineTransform a pak pomocí

```
g2.transform(AffineTransform Tx)
```

tuto transformaci provést najednou

8.7. Interakce s uživatelem

■ potřebujeme-li překreslit grafiku, aniž bychom měnili pozici a/nebo velikost základní kreslicí komponenty musíme při reakci na událost volat metodu `repaint()` z kreslicí komponenty

Příklad 8.4.

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;

class Kresleni extends JPanel {
    int pocet = 1;
    protected void paintComponent(Graphics g) {
        // super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        Dimension d = getSize();
        int y = d.height;
        int krok = d.height / 20;
        g2.setStroke(new BasicStroke(10.0f));
        g2.setPaint(Color.yellow);
        g2.draw(new Line2D.Double(d.width / 2, y, d.width / 2,
                                   y - (krok * pocet)));
    }
}

public class Klikani {
    Kresleni kresleni;
    Klikani() {
        JFrame oknoF = new JFrame("Klikani");
        kresleni = new Kresleni();
        JPanel jPN = new JPanel();
        JButton tBT = new JButton("+1");
        tBT.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                kresleni.pocet++;
                kresleni.repaint();
            }
        });
        jPN.add(tBT);
        oknoF.add(jPN, BorderLayout.NORTH);
        oknoF.add(kresleni, BorderLayout.CENTER);
        oknoF.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        oknoF.setSize(250, 250);
        oknoF.setVisible(true);
    }

    public static void main(String[] args) {
        new Klikani();
    }
}
```

8.8. Interakce uživatele s jednotlivými grafickými objekty

- občas potřebujeme, aby grafické objekty reagovaly na události od myši
- grafická primitiva obsahují metodu `contains()`, která zjistí, zda jsme uvnitř objektu
- ♦ pak stačí jen obsloužit události od `MouseListener` (kliknutí) a/nebo `MouseMotionListener` (pohyb)

Příklad 8.5.

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;

class Kresleni extends JPanel implements MouseListener,
                                           MouseMotionListener {

    int pocet;
    Color barva;
    Rectangle2D cara;
    Kresleni() {
        setOpaque(true);
        pocet = 1;
        barva = Color.yellow;
        cara = new Rectangle2D.Double();
        this.addMouseListener(this);
        this.addMouseMotionListener(this);
    }

    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        Dimension d = getSize();
        int y = d.height;
        int krok = d.height / 20;
        g2.setPaint(barva);
        cara.setRect(d.width / 2, y - (krok * pocet),
                    10, (krok * pocet));
        g2.fill(cara);
    }

    public void mouseClicked(MouseEvent e) {
        if (cara.contains(e.getX(), e.getY())) {
            if (barva.equals(Color.yellow)) {
                barva = Color.red;
            }
            else {
                barva = Color.yellow;
            }
        }
        repaint();
    }

    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
    public void mousePressed(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }

    public void mouseDragged(MouseEvent e) { }
    public void mouseMoved(MouseEvent e) {
        if (cara.contains(e.getX(), e.getY())) {
            barva = Color.green;
        }
    }
}
```

```
    }
    else {
        barva = Color.blue;
    }
    repaint();
}
}
```

8.9. Práce s textem a fonty

- srovnáme-li popisy komponent a psaní textu do grafického okénka, pak jediná shodná věc je práce s nastavením fontů

- třída **Font** a metody **getFont()** a **setFont()** fungují

- dále však následují samé odlišnosti

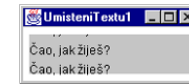
- metody pro výpis textu:

```
drawString(String str, int x, int y)
```

- souřadnice **x** a **y** označují počáteční bod, odkud se bude text vypisovat

- co je míněno pojmem „počáteční bod“?

```
g.drawString(s, 0, 0);
g.drawString(s, 0, 20);
g.drawString(s, 0, d.height - 1);
```



- písmo na řádce je umístěno do tzv. **písmové osnovy**

- tu tvoří několik vodorovných čar, nazývaných odborně **dotažnice**

- tři základní z nich se jmenují **horní dotažnice** (*ascender line*), **základní dotažnice** neboli **účaři** (*baseline*) a **dolní dotažnice** (*descender line*)

- ♦ význam *ascender line* je v typografii jiný – velikost velkých písmen

- metodě **drawString()** se předávají souřadnice levého bodu na základní dotažnici

- to je zásadní rozdíl (a také častý zdroj výše zobrazených chyb) se zadáváním souřadnic grafických primitiv, u nichž se zadává vždy levý horní roh

- použitelnou (nikoliv přesnou!) hodnotu nutného posunutí první řádky textu lze získat **getSize()** třídy **Font**

- udává celkovou velikost aktuálního fontu v typografických bodech

```
Font f = g.getFont();
int vel = f.getSize();
g.drawString(s, 0, 0 + vel);
g.drawString(s, 0, d.height - vel);
```



8.9.1. Metrika fontu

- pokud chceme detailnější informace o aktuálním fontu, musíme použít třídu `FontMetrics`

- získáme i informace „šířkové“, kdy např. můžeme zjistit, zda se nějaký text v použitém fontu a v zadané velikosti ještě vejde na obrazovku

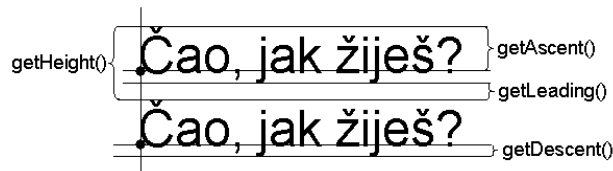
- instanci `FontMetrics` získáme pomocí `getFontMetrics()` třídy `Graphics2D`

- metoda je přetížená

```
FontMetrics getFontMetrics(Font f)
```

- ♦ vrátí metriku libovolného fontu

- `FontMetrics` má pouze metody, které vrací informace, nelze tedy pomocí nich nic měnit



- svislé rozměry fontu vrací `getAscent()`, `getDescent()`, `getLeading()` a `getHeight()`

- `getHeight()` se typicky použije pro posun y-ové souřadnice při výpisu další řádky, tj. je to výška řádky

- ♦ do těchto hodnot se započítávají i akcenty nad velkými písmeny

- toto nemusí být splněno pro všechny fonty

- změna různých rozměrů v závislosti na velikosti fontu a typu rodiny písma



- další skupina metod z této třídy umožňuje získat šířku znaku nebo řetězce v daném fontu

- slouží pro vycentrování textu, zjištění, zda se vejde do okénka atp.

- šířka jednoho znaku `int charWidth(char ch)`

- šířka řetězce znaků braného jako celek

```
int stringWidth(String str)
```

- `int[] getWidths()` – vrátí šířky jednotlivých znaků pro prvních 256 znaků fontu

- pomocí ní můžeme např. snadno najít „úsporné“ (tj. „štíhlé“) písmo

- `int getMaxAdvance()` – vrátí šířku nejširšího znaku ve fontu

8.9.2. Použití fyzických fontů

- grafický kontext nám umožňuje použít všechny fonty, které jsou dostupné v našem počítači

- musíme vytvořit instanci třídy `java.awt.GraphicsEnvironment` voláním její statické metody `getLocalGraphicsEnvironment()`

- pak lze použít metodu `getAllFonts()`, která vrátí pole objektů třídy `Font`

```
GraphicsEnvironment ge =
    GraphicsEnvironment.getLocalGraphicsEnvironment();
Font[] fonty = ge.getAllFonts();
```

- z tohoto pole lze získat již známými metodami jména jednotlivých fontů a ty pak použít

- druhou možností je použít metodu vracející jména rodin písem

```
String[] jmena = ge.getAvailableFontFamilyNames();
```

- a pak pomocí metod `deriveFont()` vytvořit požadovaný řez

- pokusy ukazují, že se můžete spolehnout na všechny logické fonty



Příklad 8.6. Ukázka použití fontů pro popis grafu

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;

class Kresleni extends JPanel {
    static final int OKRAJ = 10;
    int pocet = 1;
    int krok, maxY;
    Kresleni() {
        setOpaque(true);
    }
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        nakresliOsy(g2);
        Dimension d = getSize();
        g2.setStroke(new BasicStroke(30.0f,
                                   BasicStroke.CAP_BUTT,
                                   BasicStroke.JOIN_MITER));

        g2.setPaint(Color.yellow);
        int px = d.width / 2;
        int py = maxY - (krok * pocet);
        g2.draw(new Line2D.Double(px, maxY, px, py));
        vypisHodnotu(g2, px, py);
    }

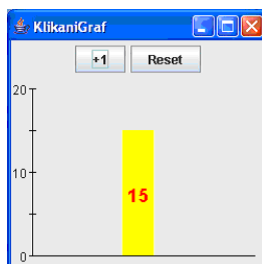
    void nakresliOsy(Graphics2D g2) {
        int horniMez = (pocet + 10) / 10 * 10;
        Dimension d = getSize();
        maxY = d.height - OKRAJ;
        krok = (maxY - OKRAJ) / horniMez;
        g2.setStroke(new BasicStroke(1.0f));
        g2.setPaint(Color.black);
        g2.draw(new Line2D.Double(OKRAJ * 2, maxY,
                                d.width - OKRAJ, maxY));
        g2.draw(new Line2D.Double(OKRAJ * 2, OKRAJ,
                                OKRAJ * 2, maxY));
        FontMetrics fm = g2.getFontMetrics();
        for (int i = 0; i <= horniMez; i += 5) {
            int py = maxY - (krok * i);
            g2.draw(new Line2D.Double(OKRAJ * 2 - 3, py,
                                    OKRAJ * 2 + 3, py));

            if (i % 10 == 0) {
                String s = "" + i;
                int sirka = fm.stringWidth(s);
                int posun = fm.getAscent() / 2;
                g2.drawString(s, OKRAJ * 2 - 5 - sirka, py + posun);
            }
        }
    }
}
```

```
void vypisHodnotu(Graphics2D g2, int px, int py) {
    if (pocet == 0) {
        return;
    }
    g2.setPaint(Color.red);
    Font f = g2.getFont();
    Font fb = new Font(f.getFontName(), Font.BOLD, 18);
    FontMetrics fm = g2.getFontMetrics(fb);
    g2.setFont(fb);
    String s = "" + pocet;
    int sirka = fm.stringWidth(s) / 2;
    int posun = fm.getAscent() / 2;
    int y = (maxY - py) / 2 - posun;
    g2.drawString(s, px - sirka, maxY - y);
}

public class KlikaniGraf {
    Kresleni kresleni;
    KlikaniGraf() {
        JFrame oknoF = new JFrame("KlikaniGraf");
        kresleni = new Kresleni();
        JPanel jPN = new JPanel();
        JButton tBT = new JButton("+1");
        tBT.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                kresleni.pocet++;
                kresleni.repaint();
            }
        });
        JButton resetBT = new JButton("Reset");
        resetBT.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                kresleni.pocet = 0;
                kresleni.repaint();
            }
        });
        jPN.add(tBT);
        jPN.add(resetBT);
        oknoF.add(jPN, BorderLayout.NORTH);
        oknoF.add(kresleni, BorderLayout.CENTER);
        oknoF.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        oknoF.setSize(250, 250);
        oknoF.setVisible(true);
    }

    public static void main(String[] args) {
        new KlikaniGraf();
    }
}
```



Kapitola 9. Schémové jazyky DTD a XSD

9.1. Význam schémových jazyků (schémat)

- pomocí schémových jazyků se definuje nový značkovací jazyk
 - má syntaxi XML
 - ale používá námi vytvořené značky
 - ♦ prakticky – je to stále XML
- význam a možnosti schémových jazyků
 - je to formální definice datových formátů založených na XML
 - ♦ jaké elementy a atributy lze v XML použít a v jakých vzájemných vztazích
 - ♦ určení datových typů elementů či atributů
 - lze kontrolovat správnost (validitu) XML – nejčastější použití
 - schéma slouží jako dokumentace použitého značkovacího jazyka
 - lepší XML editory mohou být schématem řízeny
 - ♦ nabízejí rovnou vhodnou značku a doplňují kód
 - schéma je zdrojem pro automatické vytvoření odpovídajícího objektového modelu – *data binding*

9.2. DTD – *Document Type Definition*

- současný názor na DTD: „největší chyba ve vývoji XML“
- DTD (Definice Typu Dokumentu) patří do obecné skupiny jazyků pro popis schématu dokumentu XML
 - je to první schémový jazyk
- DTD pochází z textově orientovaných dokumentů
 - pro datově orientované dokumenty je méně vhodný – použít místo něj WXS

9.2.1. Výhody a nevýhody DTD

- DTD má jednu zásadní výhodu
 - je v XML od samého počátku a všechny aplikace s ním umějí pracovat
- nevýhody DTD:
 - není napsáno v XML (nelze jej kontrolovat na správnost)
 - nepopisuje datové typy, rozsahy, omezení atd.

- neodporuje jmenné prostory
- nevýhody DTD byly zřejmě od samého počátku a proto rychle vznikaly další schémové jazyky
 - v současné době jsou používány dva, které DTD vytlačují
 - ♦ XML Schema, též „W3C XML Schema“ – WXS, `soubor.XSD`
 - ♦ Relax NG, `soubor.RNG`
- bude popisována jen omezená množina možností s ohledem na využití datově orientovaných dokumentech

9.2.2. Spojení DTD a XML

- DTD je obvykle uloženo v samostatném souboru `.DTD`
 - zřídka je přímou součástí XML dokumentu
- v XML musí být na samém začátku deklarace typu dokumentu (DOCTYPE)

```
<!DOCTYPE kořenový_element SYSTEM "jmeno_souboru.dtd">
```

- soubor `spojeni.dtd` (má pouze jednu řádku kódu)

```
<!ELEMENT spojeni (#PCDATA)>
```

- soubor `spojeni.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE spojeni SYSTEM "spojeni.dtd">
<spojeni>
  Jak připojit DTD ke XML dokumentu
</spojeni>
```

- validace pomocí xerces:

```
>xerces -v spojeni.xml
spojeni.xml: 30 ms (1 elems, 0 attrs, 0 spaces, 37 chars)
```

- parametr `-v` znamená validovat oproti DTD
 - ♦ bez parametru `-v` se ověřuje pouze *well-formed*

- jsou možné i jiné způsoby připojení – viz literatura

9.2.3. Prvky a struktura DTD

- komentáře jsou stejné jako u XML

```
<!-- komenář -->
```

- DTD může obsahovat čtyři typy deklarací

- elementy – používají se často
- atributy – používají se často
- entity – používají se zřídka
- notace – používají se zřídka

Poznámka

Vyskytují-li se v DTD souboru akcenty, je vhodné přidat hlavičku s použitým charsetem stejnou jako v XML

```
<?xml version="1.0" encoding="UTF-8"?>
```

9.2.3.1. Deklarace elementů

- způsob zápisu

```
<!ELEMENT název obsah>
```

- `název` je shodný s příslušnou značkou XML a platí pro něj stejná pravidla pro vytváření
- `obsah` má více podob

9.2.3.1.1. Prázdný element

- nejjednodušší – EMPTY

```
<!ELEMENT br EMPTY>
```

- v XML pak `
</br>` nebo jen `
`

Výstraha

Mezi značkami nesmí být odřádkováno – chybně:

```
<br>
</br>
```

9.2.3.1.2. Modelová skupina

Poznámka

Je to nejčastější použití.

- pro elementy obsahující:

- další elementy
- text

- smíšený obsah (text + elementy)

■ je uzavřena do kulatých závorek, vnořené elementy se oddělují:

- čárkou – v XML musí být ve stejném pořadí

```
<!ELEMENT jméno (křestní, příjmení)>
```

- svislítkem – v XML může být jen jeden z nich

```
<!ELEMENT pohlaví (muž | žena)>
```

- tyto způsoby lze navzájem kombinovat

```
<!ELEMENT jméno ((křestní, příjmení) | (příjmení, křestní))>  
<!ELEMENT osobaPlochá (křestní, příjmení, (muž | žena))>
```

Příklad 9.1.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE osoby SYSTEM "osoby.dtd">
```

```
<osoby>  
  <osobaStrukturovaná>  
    <jméno>  
      <křestní>  
        Bilbo  
      </křestní>  
      <příjmení>  
        Pytlík  
      </příjmení>  
    </jméno>  
    <poohlaví>  
      <muž/>  
    </poohlaví>  
  </osobaStrukturovaná>
```

```
<osobaStrukturovaná>  
  <jméno>  
    <příjmení>  
      Roberts  
    </příjmení>  
    <křestní>  
      Julie  
    </křestní>  
  </jméno>  
  <poohlaví>  
    <žena/>  
  </poohlaví>  
</osobaStrukturovaná>
```

```
<osobaPlochá>  
  <křestní>  
    Pamela  
  </křestní>  
  <příjmení>  
    Anderson  
  </příjmení>  
  <žena/>  
</osobaPlochá>  
</osoby>
```

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!ELEMENT osoby (osobaStrukturovaná | osobaPlochá)* >
```

```
<!ELEMENT osobaStrukturovaná (jméno, pohlaví)>
```

```
<!ELEMENT osobaPlochá (křestní, příjmení, (muž | žena))>
```

```
<!ELEMENT jméno ((křestní, příjmení) | (příjmení, křestní))>
```

```
<!ELEMENT křestní (#PCDATA)>
<!ELEMENT příjmení (#PCDATA)>
```

```
<!ELEMENT pohlaví (muž | žena)>
```

```
<!ELEMENT muž EMPTY>
<!ELEMENT žena EMPTY>
```

■ kromě pořadí elementů lze určit i jejich počet pomocí dalšího znaku za názvem elementu

- vyskytuje se 0 nebo vícekrát

```
<!ELEMENT potomek (syn*, dcera*)>
```

- vyskytuje se 0 nebo jednou (nepovinný výskyt)

```
<!ELEMENT rodina (manželka?, manžel?, potomek*)>
```

- vyskytuje se 1 nebo vícekrát

```
<!ELEMENT zaměstnanec (nadřízený+)>
```

■ lze vytvářet nejrůznější kombinace

- existují alespoň dva proděkan

```
<!ELEMENT fakulta (proděkan, proděkan+)>
```

- pokud je jídlo, pak se skládá z polévky a hlavního chodu

```
<!ELEMENT jídlo (polévka, hlavníChod)?>
```

9.2.3.1.3. Konečný element

■ obsahuje vlastní text

```
<!ELEMENT polévka (#PCDATA)>
```

■ v XML může být např.

```
<polévka>gulášová</polévka>
```

9.2.3.1.4. Element obsahuje smíšená data

■ vlastní text a další elementy

■ pak musí být #PCDATA uvedeno ve skupině jako první, spojení musí být jen pomocí | a musí být opakovací *

```
<!ELEMENT polévka (#PCDATA | kvalita | stáří)*>
<!ELEMENT kvalita (#PCDATA)>
<!ELEMENT stáří (#PCDATA)>
```

■ v XML může být např. :

```
<polévka>
  <stáří>
    včerejší
  </stáří>
  gulášová
  <kvalita>
    vynikající
  </kvalita>
  zlevněná
</polévka>
```

9.2.3.2. Deklarace atributů

■ způsob zápisu

```
<!ATTLIST jméno_elementu deklarace_atributů>
```

■ deklarace atributu má tři části:

- jméno atributu – je shodné s XML
- typ atributu
- povinnost atributu, případně jeho standardní hodnota

9.2.3.2.1. Typ atributu

■ rozeznáváme pět typů

1. CDATA (pozor, u elementů jsou to #PCDATA)

- nejobecnější – skutečná hodnota je obecný text

```
<!ATTLIST polévka množství CDATA>
```

- v XML může být např.:

```
<polévka množství="jedna sběračka">
```

2. NMTOKEN

- skutečná hodnota je jedno slovo (omezení stejná jako u názvů elementů)

```
<!ATTLIST polévka množství NMTOKEN>
```

- v XML může být např. :

```
<polévka množství="2litry">
```

3. NMTOKENS

- skutečná hodnota je několik slov oddělených mezerami

```
<!ATTLIST polévka složení NMTOKENS>
```

- v XML může být např. :

```
<polévka složení="voda maso sůl a-5-přísad">
```

4. ID – identifikátor

- musí mít jedinečnou hodnotu v rámci celého dokumentu a to i pro různé elementy a různé názvy atributů

Výstraha

Hodnota ID musí začínat písmenem:

- ♦ A12345 je správně

- ♦ 12345 je chybně

```
<!ATTLIST polévka označení ID>
```

- v XML může být např. :

```
<polévka označení="gul001">
```

5. uvedení výčtu

```
<!ATTLIST polévka množství (jedenTaliř | dvaTaliře | třiTaliře)>
```

- v XML může být např. :

```
<polévka množství="dvaTaliře">
```

9.2.3.2.2. Povinnost atributu

■ uvádí se za typ atributu

- není-li uvedena, předpokládá se #REQUIRED

■ rozeznáváme tři typy

1. #REQUIRED

- tento atribut musí být v XML vždy uveden

```
<!ATTLIST polévka množství NMTOKEN #REQUIRED>
```

- v XML musí být např. :

```
<polévka množství="1litr">
```

2. #IMPLIED

- tento atribut je volitelný

```
<!ATTLIST polévka množství NMTOKEN #IMPLIED>
```

- v XML může být např. :

```
<polévka>
```

3. standardní (implicitní) hodnota

- používá se zejména u výčtů

```
<!ATTLIST polévka množství (jedenTaliř | dvaTaliře | třiTaliře)
    "jedenTaliř">
```

- v XML může být např.:

```
<polévka>
```

ve smyslu

```
<polévka množství="jedenTaliř">
```

- v případě standardních hodnot se používá v hlavičce XML souboru ještě třetí atribut standalone s hodnotami yes nebo no

- ♦ standalone="yes" znamená, že XML soubor je nezávislý na případném DTD, tj. bez něj obsahuje stejné údaje, jako s ním

```
<?xml version="1.0"
    encoding="UTF-8"
    standalone="no"?>
<!DOCTYPE polévka SYSTEM "polevka-vycet.dtd">
<polévka/>
```

- ♦ standalone="no" – obsah XML souboru závisí na DTD

```
<?xml version="1.0"
    encoding="UTF-8"
    standalone="yes"?>
```

```
<!DOCTYPE polévka SYSTEM "polevka-vycet.dtd">
<polévka/>
```

```
>xerces -v polevka-standalone-no.xml
polevka-standalone-no.xml: 140 ms (1 elems, 1 attrs, 0 spaces, 0 chars)
>xerces -v polevka-standalone-yes.xml
[Error] polevka-standalone-yes.xml:5:11: Attribute "množství" for element ►
type "polévka" has a default value and must be specified in a standalone ►
document.
polevka-standalone-yes.xml: 221 ms (1 elems, 1 attrs, 0 spaces, 0
```

- před standardní hodnotou může být navíc #FIXED
 - ♦ atribut musí mít jenom standardní hodnotu

9.2.3.2.3. Praktické doporučení pro datově orientované dokumenty

■ každý atribut musí být pouze #REQUIRED

- protože ve všech ostatních možnostech jsou jiná data bez použití DTD a s použitím DTD!

■ má-li element více atributů, většinou se deklarují v jednom ATTLIST

```
<!ATTLIST polévka množství CDATA #REQUIRED
                složení NMTOKENS #REQUIRED
                označení ID #REQUIRED>
```

■ na skutečném pořadí atributů v XML souboru pak nezáleží

- je ale vhodné pořadí z DTD dodržovat

9.2.3.3. Parametrické entity

■ zabraňují v DTD opakování stejného zdrojového kódu

- často se používají u společných atributů

```
<!ENTITY % spolecneAtributy
        "pohlavi (muž | žena) #REQUIRED
        titul CDATA #IMPLIED">
```

```
<!ELEMENT lekar (jmeno, telefon)>
<!ATTLIST lekar %spolecneAtributy;>
```

```
<!ELEMENT sestra (jmeno, telefon)>
<!ATTLIST sestra %spolecneAtributy;>
```

```
<!ELEMENT pacient (jmeno, vaha, vyska)>
<!ATTLIST pacient %spolecneAtributy;
                cisloPojisteni ID #REQUIRED>
```

■ příklad viz dále

9.2.3.4. Deklarace entit

■ kromě parametrických entit lze mít v DTD ještě další typy entit

- použitelnost po datově orientované dokumenty je sporná
- podrobnosti viz v literatuře

Příklad 9.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE obezitologie SYSTEM "obezitologie.dtd">

<obezitologie>
  <personal>
    <lekar pohlavi="muž" titul="MUDr.">
      <jmeno>
        <krestni>Pavel
        </krestni>
        <prijmeni>Petr
        </prijmeni>
      </jmeno>
      <telefon>377 123 456
      </telefon>
    </lekar>
    <sestra pohlavi="žena">
      <jmeno>
        <krestni>Vanda
        </krestni>
        <prijmeni>Alexandra
        </prijmeni>
      </jmeno>
      <telefon>377 123 789
      </telefon>
    </sestra>
  </personal>
  <leceneOsoby>
    <nadvaha>
      <pacient pohlavi="muž" cisloPojisteni="VZP1234">
        <jmeno>
          <krestni>Jiří
          </krestni>
          <prijmeni>Štíhlý
          </prijmeni>
        </jmeno>
        <vaha jednotka="kg">150
        </vaha>
        <vyska jednotka="cm">150
        </vyska>
      </pacient>
    </nadvaha>
    <podvyziva>
      <pacient pohlavi="žena" cisloPojisteni="PMV12AB" titul="JUDr.">
        <jmeno>
          <krestni>Otýlie
          </krestni>
          <prijmeni>Otylá
          </prijmeni>
        </jmeno>
        <vaha jednotka="kg">45,5
        </vaha>
```

```
        <vyska jednotka="cm">170
        </vyska>
      </podvyziva>
    </leceneOsoby>
  </obezitologie>

<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT obezitologie (personal, leceneOsoby)>
<!ELEMENT personal (lekar | sestra)*>

<!ENTITY % spolecneAtributy
  "pohlavi (muž | žena) #REQUIRED
  titul CDATA #IMPLIED">

<!ELEMENT lekar (jmeno, telefon)>
<!ATTLIST lekar %spolecneAtributy;>

<!ELEMENT sestra (jmeno, telefon)>
<!ATTLIST sestra %spolecneAtributy;>

<!ELEMENT pacient (jmeno, vaha, vyska)>
<!ATTLIST pacient %spolecneAtributy; cisloPojisteni ID #REQUIRED>

<!ELEMENT jmeno (krestni, prijmeni)>
<!ELEMENT krestni (#PCDATA)>
<!ELEMENT prijmeni (#PCDATA)>
<!ELEMENT telefon (#PCDATA)>

<!ELEMENT leceneOsoby (nadvaha, podvyziva)>
<!ELEMENT nadvaha (pacient)*>

<!ELEMENT vaha (#PCDATA)>
<!ATTLIST vaha jednotka (kg | g | lb) #REQUIRED>

<!ELEMENT vyska (#PCDATA)>
<!ATTLIST vyska jednotka (cm | in) #REQUIRED>

<!ELEMENT podvyziva (pacient)*>
```

9.3. W3C XML Schema – WXS nebo XSD

9.3.1. Základní informace

- referenční popis: <http://www.w3.org/TR/xmlschema-0>
 - další informace: <http://www.kosek.cz>
- nejdůležitější schémový jazyk
 - od května 2001 všeobecně uznávaný standard

- podporován všemi významnými „hráči“ – Sun, IBM, Microsoft, Oracle, open-source SW, atd.
- zkratka WXS nebo často XSD (podle přípony souborů)
 - dále bude používána zkratka XSD
- nevýhody
 - poněkud složitá specifikace
 - „ukecaný“ jazyk – chybí mu určitá elegance
- jeho přímý konkurent Relax NG tyto nevýhody odstraňuje
 - ale není dosud všeobecně přijímán a podporován
- nepřímý konkurent je jazyk Schematron
 - lze použít jen k validaci
 - využívá možnosti jazyka XPath a dokáže tak kontrolovat např. i obsahy elementů v závislosti na jiných elementech
 - ♦ to nedokáže ani XSD ani RNG
 - v současnosti je Schematron používán jako doplněk XSD či RNG
- všechny zmíněné jazyky dodržují konvenci XML
 - lze je validovat proti sobě samým
- v současnosti je velké úsilí sjednotit schémové jazyky pod jeden zastřešující jazyk
 - DSDL (*Document Schema Definition Languages*)
 - dále existují nástroje (např. Trang), které umožňují vzájemnou konverzi těchto jazyků

9.3.2. Praktické použití XSD

- z velké množiny možností bude popisována jen nejnútnejší podmnožina pro datově orientované dokumenty
 - navíc bude používán jen jeden z více možných způsobů zápisu

9.3.3. Začátek XSD souboru

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

- definování jmenného prostoru
 - typicky je označován `xs` (občas i `xsi`)
 - kořenový element je tedy vždy `xs:schema`
- dále se využívá možnost definovat své vlastní datové typy

- významná vlastnost XSD
- nejběžněji restrikci z jichž existujících
 - ♦ možné je ale i rozšíření již existujících (viz dále příklad u prázdného elementu s atributem)
- nový datový typ může být
 - jednoduchý (`xs:simpleType`) – odvozený od základních (viz dále obrázek)
 - složený (`xs:complexType`) – sestavený z jednoduchých typů
- postup práce
 - pro všechny neprázdné koncové elementy nebo atributy vytvořit nové jednoduché datové typy
 - z těchto jednoduchých typů pak sestavit složené datové typy pro každý element až na úroveň typu kořenového elementu
 - na závěr se podle typu kořenového elementu definuje jeden element
- jednoduchý úvodní příklad – XML soubor

```
<otec>
  <jmeno>Pavel</jmeno>
  <deti pocet="2"/>
</otec>
```

- neprázdný koncový element je pouze jeden a to `<jmeno>` typu řetězec
 - atribut je také pouze jeden a to `pocet`
- v XSD souboru připravíme dva nové jednoduché datové typy

Poznámka

v názvu typu se vždy přidává „Type“, aby bylo jasné, že se jedná o datový typ

- první bude sloužit pro element `<jmeno>`

```
<xs:simpleType name="jmenoType">
  <xs:restriction base="xs:string">
  </xs:restriction>
</xs:simpleType>
```

 - ♦ od této chvíle lze používat nový datový typ `jmenoType` ve významu „řetězec uchovávající jméno“
 - zde se nejedná o restrikci, ale o převzetí 1:1
- další nový datový typ je `pocetType`
 - ♦ zde je použit skutečný způsob restrikce (označovaný jako *constraining facets*), např.:
 - omezení říkají, že:
 - celá čísla budou omezena na nezáporná celá čísla

- možné hodnoty budou 0, 1, 2, ..., 9

```
<xs:simpleType name="pocetType">
  <xs:restriction base="xs:nonNegativeInteger">
    <xs:maxExclusive value="10"/>
  </xs:restriction>
</xs:simpleType>
```

Poznámka

Běžně použitelné základní typy a možné restriktce viz podrobně dále.

■ z jednoduchých typů sestavíme postupně složené typy

- to je nutné provést pouze pro element <deti>

```
<xs:complexType name="detiType">
  <xs:attribute name="pocet" type="pocetType" use="required"/>
</xs:complexType>
```

■ teď již máme datové typy všech elementů a připravíme datový typ kořenového elementu <otec>

- to se provede deklarováním vnořených elementů

```
<xs:complexType name="otecType">
  <xs:sequence>
    <xs:element name="jmeno" type="jmenoType"/>
    <xs:element name="deti" type="detiType"/>
  </xs:sequence>
</xs:complexType>
```

■ na závěr připravíme podle datového typu otecType jeden element otec

```
<xs:element name="otec" type="otecType"/>
```

■ celý XSD soubor tedy bude:

```
<?xml version="1.0" encoding="utf-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="jmenoType">
    <xs:restriction base="xs:string">
      </xs:restriction>
    </xs:simpleType>

  <xs:simpleType name="pocetType">
    <xs:restriction base="xs:nonNegativeInteger">
      <xs:maxExclusive value="10"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="detiType">
    <xs:attribute name="pocet" type="pocetType" use="required"/>
```

```
</xs:complexType>
```

```
<xs:complexType name="otecType">
  <xs:sequence>
    <xs:element name="jmeno" type="jmenoType"/>
    <xs:element name="deti" type="detiType"/>
  </xs:sequence>
</xs:complexType>
```

```
<xs:element name="otec" type="otecType"/>
</xs:schema>
```

■ na příkladu je jasně vidět nevýhoda XSD – „ukecanost“

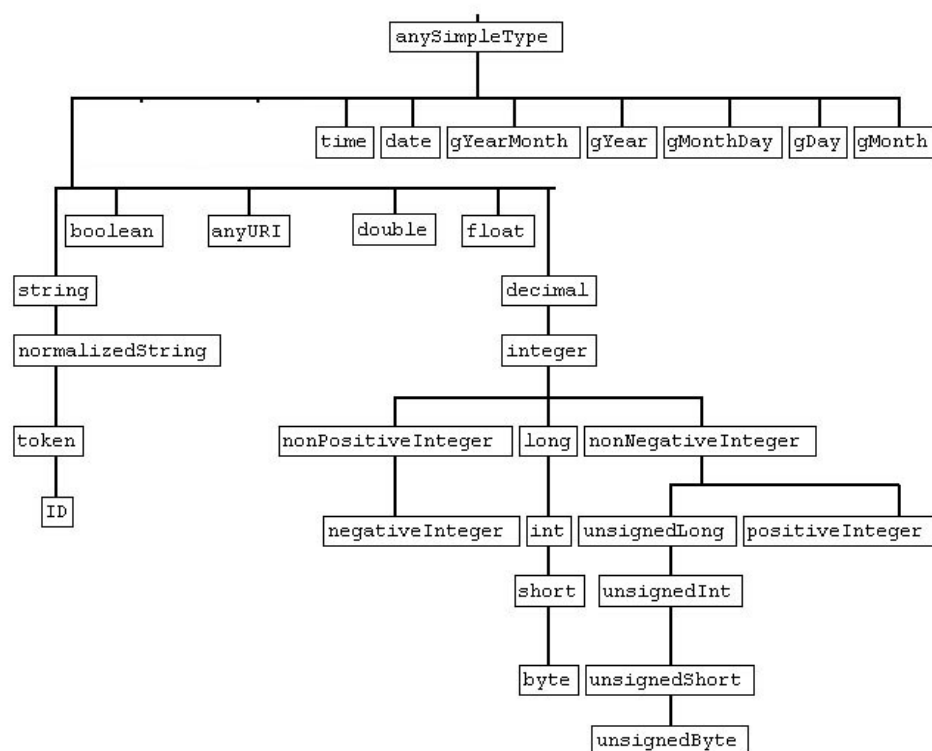
■ výhody této metody nazývané „metoda slepého Benátčana“

- naprosto průzračný způsob sestavování krok po kroku
- po pochopení celého principu je vytváření i složitého XSD víceméně mechanickou záležitostí
- všechny nové typy jsou na základní úrovni (nejsou vnořené), což dále přispívá k přehlednosti

■ stejný příklad se dá pomocí XSD napsat i kompaktněji, ale méně přehledně

- metody s názvy „matrijůška“ nebo „salámová kolečka“ (viz literatura)

9.3.4. Výběr běžně použitelných základních typů



9.3.5. Jaké možnosti nám dávají základní typy

a. datумы a časy

date	datum	YYYY-MM-DD	2004-12-08
time	čas	HH:MM:SS	11:05:48
gYear	rok	YYYY	0001 až 9999
gMonth	měsíc	MM	01 až 12
gDay	den	DD	01 až 31
gYearMonth	rok a měsíc	YYYY-MM	2004-07
gMonthDay	měsíc a den	--MM-DD	--12-08

(úvodní dvě pomlčky jsou nutné!)

b. boolean – hodnoty true, false nebo 1, 0

c. reálná čísla (nepoužívat – použít decimal)

double – reálné číslo dle běžných konvencí

float – dtto

d. decimal – celá i reálná čísla zapsaná pomocí desítkových číslic

■ je vhodné používat místo double a float

- má větší možnosti restrikcí

e. celá čísla

■ není-li uvedeno znaménko, uvažuje se +

integer – celé číslo včetně případného znaménka neomezené velikosti

long – celé číslo na 8 bajtech (-9 223 372 036 854 775 808 až 9 223 372 036 854 775 807)

int – na 4 bajtech (-2 147 483 648 až 2 147 483 647)

short – na 2 bajtech (-32 768 až 32 767)

byte – na 1 bajtu (-128 až 127)

nonPositiveInteger – záporná čísla a nula

negativeInteger – jen záporná čísla

nonNegativeInteger – kladná čísla a nula

positiveInteger – jen kladná čísla

unsignedXXX – nezáporná čísla, bez znaménka, bez nevýznamových nul

f. řetězce

string – libovolný řetězec s libovolným počtem bílých znaků a konci řádek

normalizedString – řetězec bez konců řádek (CR, LF) a bez tabulátorů

token – dtto + každá mezera může být pouze jednou (ne skupiny mezer mezi slovy)

ID – (přejato z DTD) – unikátní řetězec bez bílých znaků začínající písmenem

Výstraha

poslední tři typy jsou vhodné jen pro atributy

```
<polozka>
  data
</polozka>
```

- před a za „data“ jsou mezery a konce řádek – vyhovuje pouze typ `string`
- podrobně viz dále Práce s okrajovými bílými znaky

g. `anyURI` – URI dle běžných konvencí

- např. i jméno souboru

9.3.6. Možnosti restrikcí (*constraining facets*)

Poznámka

Vždy se snažíme nalézt maximální možnou restrikcí – tím lze odhalit při validaci problémy.

a. omezení číselného rozsahu

- pro všechna čísla a časy lze použít `minInclusive`, `minExclusive`, `maxInclusive`, `maxExclusive`

```
<xs:simpleType name="pocetStudentuType">
  <xs:restriction base="xs:nonNegativeInteger">
    <xs:minInclusive value="1" />
    <xs:maxExclusive value="15" />
  </xs:restriction>
</xs:simpleType>
```

- studentů může být 1 až 14

b. omezení počtu platných míst

- pro všechna celá čísla

```
<xs:simpleType name="pocetStudentuType">
  <xs:restriction base="xs:nonNegativeInteger">
    <xs:totalDigits value="2" />
  </xs:restriction>
</xs:simpleType>
```

- studentů může být 0 až 99

c. omezení počtu desetinných míst

- jen pro decimal

- vhodné pro peněžní údaje

```
<xs:simpleType name="cenaType">
  <xs:restriction base="xs:decimal">
    <xs:totalDigits value="6" />
    <xs:fractionDigits value="2" />
    <xs:minInclusive value="0" />
  </xs:restriction>
</xs:simpleType>
```

- cena může být od 0.00 do 9999.99

d. omezení počtu znaků

- pro všechny řetězce

- `length` – pevná délka
- `minLength` – minimální délka
- `maxLength` – maximální délka

- příklad omezení pomocí `length`

```
<xs:simpleType name="pozdravType">
  <xs:restriction base="xs:string">
    <xs:length value="4" />
  </xs:restriction>
</xs:simpleType>
```

- vyhovují pouze pozdravy „ahoj“ a „hola“

- ♦ „čao“ je krátký, „nazdar“ je dlouhý

- pozor na toto omezení, protože do délky řetězce se započítávají i mezery a konce řádek (viz dále)

- ♦ vyhovuje

```
<pozdrav>ahoj</pozdrav>
```

- ♦ nevyhovuje

```
<pozdrav>
  ahoj
</pozdrav>
```

- příklad omezení pomocí `minLength` a `maxLength`

```
<xs:simpleType name="hesloType">
  <xs:restriction base="xs:token">
    <xs:minLength value="5" />
    <xs:maxLength value="15" />
  </xs:restriction>
</xs:simpleType>
```

- délka hesla musí být mezi 5 až 15 znaky včetně

e. enumeration – výčet hodnot

- pro všechny základní typy (s výjimkou `boolean`)

- velmi využívaná restrikce

```
<xs:simpleType name="kodMenyType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="CZK" />
  </xs:restriction>
</xs:simpleType>
```

```

    <xs:enumeration value="EUR" />
    <xs:enumeration value="USD" />
  </xs:restriction>
</xs:simpleType>

```

```

<xs:simpleType name="dphType">
  <xs:restriction base="xs:byte">
    <xs:enumeration value="5" />
    <xs:enumeration value="19" />
    <xs:enumeration value="22" />
  </xs:restriction>
</xs:simpleType>

```

f. pattern – vzor pomocí regulárního výrazu

- pro všechny základní typy (s výjimkou boolean)

- typicky se používá pro řetězce

```

<xs:simpleType name="pscType">
  <xs:restriction base="xs:string">
    <xs:pattern value="\d{3} \d{2}" />
  </xs:restriction>
</xs:simpleType>

```

- \d{3} znamená „tři desítkové číslice“
- vyhovuje 306 14, nevyhovují 30614, 30 614 atd.

- další možnosti pro pattern

\p{L} – jedno jakékoliv písmeno

\p{Lu} – jedno jakékoliv velké písmeno

\p{Ll} – jedno jakékoliv malé písmeno

\p{P} – interpunkce (oddělovače)

\s – bílé znaky (mezera, tabulátor, CR, LF)

\d{1,3} – jedna, dvě nebo tři číslice

- příklady užitečných patternů

"\p{Lu}\p{Ll}+ \p{Lu}\p{Ll}+" – příjmení a jméno, "Novák Jan"

"\p{Lu}\p{Ll}+ \d+" – ulice, "Univerzitní 22"

" \p{Lu}\p{Ll}+ ?[?\p{L}+]* ?\d*" – město, "Praha", "Praha 5", "Praha 10", "Ústí nad Orlicí 1"

[] – označují skupinu

- opakovací mají běžný význam:

+ – jednou nebo vícekrát

* – nula nebo vícekrát

? – nula nebo jednou

bez opakovací – právě jednou

9.3.7. Složené (komplexní) datové typy

- skládáme je z již hotových jednoduchých datových typů, které máme připraveny pro všechny elementy a atributy

- lze u nich určit
 - ♦ pořadí výskytu
 - ♦ počet opakování
 - ♦ povinnost či volitelnost

- pro komplexní typ se používá <xs:complexType>

- jednotlivé podelementy se definují pomocí xs:element

Poznámka

V dále uvedených příkladech nejsou uváděny předcházející definice jednoduchých datových typů

- existuje několik možností skládání

9.3.7.1. Skládání pomocí sequence

- pořadí jednotlivých elementů, které musí být dodrženo

```

<xs:complexType name="jmenoType">
  <xs:sequence>
    <xs:element name="krestni" type="krestniType"/>
    <xs:element name="prijmeni" type="prijmeniType"/>
  </xs:sequence>
</xs:complexType>

```

- vyhovující XML:

```

<jmeno>
  <krestni>Karel</krestni>
  <prijmeni>Čapek</prijmeni>
</jmeno>

```

- počet výskytů (opakování) jednotlivých elementů lze určit pomocí minOccurs a maxOccurs

- implicitně mají oba hodnotu 1 (tj. element je povinný)

```
<xs:complexType name="jmenoType">
  <xs:sequence>
    <xs:element name="krestni" type="krestniType"
      minOccurs="1" maxOccurs="3"/>
    <xs:element name="prijmeni" type="prijmeniType"/>
  </xs:sequence>
</xs:complexType>
```

■ vyhovující XML:

```
<jmeno>
  <krestni>Karel</krestni>
  <krestni>Jaromír</krestni>
  <prijmeni>Erben</prijmeni>
</jmeno>
```

■ neohraničená (neomezená) hodnota je unbounded

```
<xs:complexType name="predmetType">
  <xs:sequence>
    <xs:element name="prerekvizita" type="prerekvizitaType"
      minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="nazev" type="nazevType"/>
  </xs:sequence>
</xs:complexType>
```

■ vyhovující XML:

```
<predmet>
  <prerekvizita>PPA1</prerekvizita>
  <prerekvizita>PPA2</prerekvizita>
  <nazev>PT</nazev>
</predmet>

<predmet>
  <nazev>PC</nazev>
</predmet>
```

9.3.7.2. Skládání pomocí all

■ pořadí jednotlivých elementů nemusí být dodrženo – používat jen v odůvodněných případech

```
<xs:complexType name="jmenoType">
  <xs:all>
    <xs:element name="krestni" type="krestniType"/>
    <xs:element name="prijmeni" type="prijmeniType"/>
  </xs:all>
</xs:complexType>
```

■ vyhovující XML:

```
<jmeno>
  <prijmeni>Čapek</prijmeni>
  <krestni>Karel</krestni>
</jmeno>
```

■ lze použít i minOccurs a maxOccurs, ale velmi omezeně

- oba mohou být jen 0 nebo 1

```
<xs:complexType name="jmenoType">
  <xs:all>
    <xs:element name="krestni" type="krestniType" minOccurs="0"/>
    <xs:element name="prijmeni" type="prijmeniType"/>
  </xs:all>
</xs:complexType>
```

■ vyhovující XML:

```
<jmeno>
  <prijmeni>Čapek</prijmeni>
</jmeno>
```

9.3.7.3. Skládání pomocí choice

■ výběr jedné z možností

Výstraha

xs:enumeration bylo pro jednoduché typy

■ datový typ svobodnyType viz dále v prázdném elementu

```
<xs:complexType name="stavType">
  <xs:choice>
    <xs:element name="svobodny" type="svobodnyType"/>
    <xs:element name="zenaty" type="zenatyType"/>
  </xs:choice>
</xs:complexType>
```

■ vyhovující XML:

```
<stav>
  <svobodny/>
</stav>

<stav>
  <zenaty>2004-01-12</zenaty>
</stav>
```

9.3.7.4. Smíšený obsah

- v datově orientovaných dokumentech zásadně nepoužívat!
- u definice komplexního typu přidáme atribut `mixed="true"`

```
<xs:complexType name="odstavecType" mixed="true">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="tucne" type="tucneType"/>
    <xs:element name="italika" type="italikaType"/>
  </xs:choice>
</xs:complexType>
```

- vyhovující XML:

```
<odstavec> Nějaký text se <tucne>zvýrazněním</tucne>
nebo jiným<italika>zvýrazněním</italika>,
aby to bylo <tucne>pěkné</tucne>.
</odstavec>
```

9.3.7.5. Prázdný element

- element je důležitý pouze svým výskytem nebo atributy (viz dále)

```
<xs:complexType name="svobodnyType">
</xs:complexType>
```

9.3.8. Atributy

- jednoduché typy se pro hodnoty atributů připraví zcela stejně jako pro koncové elementy
 - do složených typů se atributy přidávají až nakonec za elementy pomocí `xs:attribute`
- oproti elementům přibývá ještě atribut `use`, který bude mít v datově orientovaných dokumentech vždy hodnotu `use="required"`, tzn. atribut je povinný
 - další možnosti (známé i z DTD)
 - ♦ `use="implied"` je volitelný
 - ♦ `default="Josef"` je standardní (=předdefinovaná) hodnota
 - obě možnosti zásadně nepoužívat!

```
<xs:complexType name="stavType">
  <xs:choice>
    <xs:element name="svobodny" type="svobodnyType"/>
    <xs:element name="zenaty" type="zenatyType"/>
  </xs:choice>
  <xs:attribute name="krestni" type="krestniType" use="required"/>
</xs:complexType>
```

- vyhovující XML

```
<stav krestni="Karel">
  <svobodny/>
</stav>

<stav krestni="Jan">
  <zenaty>2004-01-12</zenaty>
</stav>
```

9.3.9. Atribut je součástí koncového elementu

- příklad

```
<cena penezniJednotka="CZK">
  120
</cena>
```

- používá se nikoliv běžná restrikce (`xs:restriction`), ale rozšíření již definovaného jednoduchého typu (`xs:extension`)

```
<xs:simpleType name="hodnotaType">
  <xs:restriction base="xs:nonNegativeInteger">
  </xs:restriction>
</xs:simpleType>
```

```
<xs:simpleType name="penezniJednotkaType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="CZK"/>
    <xs:enumeration value="USD"/>
  </xs:restriction>
</xs:simpleType>
```

```
<xs:complexType name="cenaType">
  <xs:simpleContent>
    <xs:extension base="hodnotaType">
      <xs:attribute name="penezniJednotka" type="penezniJednotkaType"
        use="required"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

9.3.10. Prázdný element s atributy

- velmi častá záležitost

```
<obrazek jmeno="pic001.jpg"/>
```

- vyhovující XSD

```
<xs:simpleType name="jmenoType">
  <xs:restriction base="xs:anyURI">
    </xs:restriction>
  </xs:simpleType>

<xs:complexType name="obrazekType">
  <xs:attribute name="jmeno" type="jmenoType" use="required"/>
</xs:complexType>
```

9.3.11. Závěrečná definice kořenového elementu

- máme-li připraveny typy ke všem elementům včetně kořenového, definujeme velmi jednoduše kořenový element, např.:

```
<xs:element name="obrazek" type="obrazekType"/>
```

- a tím příprava XSD souboru končí

9.3.12. Připojení schématu k dokumentu

- pouze některé parsery pro validaci nepotřebují, aby validovaný XML měl přímo v sobě informaci, podle jakého XSD je připraven (a validován)

- většinou ale XML v sobě tuto informaci má

- u XSD je zápis komplikovanější než u DTD, kde se do XML přidala jen jedna izolovaná řádka

- XSD vyžaduje změnu v kořenovém elementu, což je nepříjemné, zvlášť, má-li tento element atributy

- použijeme zápis

```
<kořenovýElement
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
  xs:noNamespaceSchemaLocation="URI{souboru.xsd}">
```

- všechny ostatní řetězce (kromě "kořenovýElement" a "URI{souboru.xsd}") musí být přesně jako v ukázce

- spleteme-li se, validace neproběhne úspěšně

- příklad pro výše uvedený otec

```
<otec
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
  xs:noNamespaceSchemaLocation="otec.xsd">
```

- příklad pro výše uvedený obrazek (má navíc atribut a je to prázdný element, což je hodně netypické)

```
<obrazek
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
  xs:noNamespaceSchemaLocation="obrazek.xsd" jmeno="pic001.jpg"/>
```

9.3.13. Validace pomocí xerces

```
xerces -v -s jmeno-souboru.xml
```

9.3.14. Jmenné prostory

- aby bylo možné sadu XSD značek snáze a jednotně identifikovat, přiřazuje se již v XSD jmenný prostor

- používá se atribut targetNamespace, např.:

```
targetNamespace="urn:x-herout:schemata:otec.1.0"
```

- pojmenování si lze víceméně libovolně zvolit, ale používá se zavedený systém:

urn – pevně daná zkratka (*Uniform Resource Name*)

x-herout – jméno tvůrce značek, x- znamená, že tato sada není nikde oficiálně evidována

schemata – popisují se XSD

otec – jméno „projektu“ (často jméno kořenového elementu)

1.0 – číslo verze a subverze

- tento jmenný prostor se většinou (v XSD) ihned deklaruje jako implicitní

- aby se nemusely v XSD zbytečně psát prefixy

- jako další atribut se uvádí elementFormDefault="qualified"

- což znamená, že celé schéma musí používat elementy zařazené do jmenného prostoru (nikoliv nějaké implicitní)

- celá „hlavička“ tedy je

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="urn:x-herout:schemata:otec.1.0"
  xmlns="urn:x-herout:schemata:otec.1.0"
  elementFormDefault="qualified">
```

- kromě ní už nejsou nutné žádné další úpravy, tj. konstrukce XSD je dále zcela stejná, jako bez jmenného prostoru (viz dříve)

- příklad vyhovujícího XML, které používá implicitní jmenný prostor:

```
<?xml version="1.0" encoding="utf-8"?>
<otec
  xmlns="urn:x-herout:schemata:otec.1.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
  xs:schemaLocation="urn:x-herout:schemata:otec.1.0
    otec-jmenny-prostor.xsd">

  <jmeno>
```



```

    Pavel
  </jmeno>
  <deti pocet="2"/>
</otec>

```

• atribut `xs:schemaLocation` obsahuje

- ♦ název jmenného prostoru zavedený v XSD: `urn:x-herout:schemata:otec.1.0`
- ♦ jméno XSD souboru: `otec-jmenny-prostor.xsd`
- ♦ oba údaje jsou v jedné uvozovkách oddělené navzájem bílými znaky

■ příklad vyhovujícího XML, které používá jmenný prostor pojmenovaný `rodic`

```

<?xml version="1.0" encoding="utf-8"?>
<rodic:otec
  xmlns:rodic="urn:x-herout:schemata:otec.1.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
  xs:schemaLocation="urn:x-herout:schemata:otec.1.0
    otec-jmenny-prostor.xsd">

  <rodic:jmeno>
    Pavel
  </rodic:jmeno>
  <rodic:deti pocet="2"/>
</rodic:otec>

```

9.3.14.1. Jak připravit XML, které využívá značek z více XSD

■ je nutné v jednom (hlavním) XSD importovat jmenné prostory a XSD soubory ostatních (podřízených) XSD

- elementy v hlavním XSD se pak definují pomocí `ref` nikoli `name`
- pak je možné pojmenovávat stejně značky – zde `jmeno`

■ podřízený soubor `deti-jmenny-prostor.xsd`

```

<?xml version="1.0" encoding="utf-8"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="urn:x-herout:schemata:deti.2.0"
  xmlns="urn:x-herout:schemata:deti.2.0"
  elementFormDefault="qualified">

  <xs:simpleType name="jmenoType">
    <xs:restriction base="xs:string">
      </xs:restriction>
    </xs:simpleType>

    <xs:element name="jmeno" type="jmenoType"/>
  </xs:schema>

```

■ hlavní soubor `matka-jmenny-prostor.xsd`

```

<?xml version="1.0" encoding="utf-8"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="urn:x-herout:schemata:matka.1.0"
  xmlns="urn:x-herout:schemata:matka.1.0"
  xmlns:dite="urn:x-herout:schemata:deti.2.0"
  elementFormDefault="qualified">

  <xs:import namespace="urn:x-herout:schemata:deti.2.0"
    schemaLocation="deti-jmenny-prostor.xsd"/>

  <xs:simpleType name="jmenoType">
    <xs:restriction base="xs:string">
      </xs:restriction>
    </xs:simpleType>

  <xs:simpleType name="pocetType">
    <xs:restriction base="xs:nonNegativeInteger">
      <xs:maxExclusive value="10"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="detiType">
    <xs:attribute name="pocet" type="pocetType" use="required"/>
  </xs:complexType>

  <xs:complexType name="matkaType">
    <xs:sequence>
      <xs:element name="jmeno" type="jmenoType"/>
      <xs:element name="deti" type="detiType"/>
      <xs:element ref="dite:jmeno" maxOccurs="9"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="matka" type="matkaType"/>
</xs:schema>

```

■ příklad vyhovujícího XML, které používá jmenný prostor pojmenovaný `rodic`

```

<?xml version="1.0" encoding="utf-8"?>
<rodic:matka
  xmlns:rodic="urn:x-herout:schemata:matka.1.0"
  xmlns:dite="urn:x-herout:schemata:deti.2.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
  xs:schemaLocation="urn:x-herout:schemata:matka.1.0
    matka-jmenny-prostor.xsd
    urn:x-herout:schemata:deti.2.0
    deti-jmenny-prostor.xsd">

  <rodic:jmeno>
    Macecha

```

```

</rodic:jmeno>
<rodic:deti pocet="2"/>
<dite:jmeno>
  Jeníček
</dite:jmeno>
<dite:jmeno>
  Mařenka
</dite:jmeno>
</rodic:matka>

```

9.3.15. Použití prázdné hodnoty

- využívá-li se XML pro spolupráci s databázemi, je občas nutné použít prázdnou hodnotu

- ve smyslu „hodnota nebyla stanovena“

- to lze zařídit, že v XSD souboru použijeme atribut `nillable="true"`

- zajistí, že v elementu může, ale nemusí být hodnota uvedena

- v XML souboru pak použijeme

```
<druhy xs:nil="true"/>
```

- nebo ve stejném významu jen

```
<druhy/>
```

- příklad XSD souboru

```

<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:simpleType name="prvniType">
    <xs:restriction base="xs:string">
      </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="druhyType">
      <xs:restriction base="xs:string">
        </xs:restriction>
      </xs:simpleType>

    <xs:complexType name="autoriType">
      <xs:sequence>
        <xs:element name="prvni" type="prvniType"/>
        <xs:element name="druhy" type="druhyType" nillable="true"/>
      </xs:sequence>
    </xs:complexType>

    <xs:element name="autori" type="autoriType"/>
  </xs:schema>

```

- příklad XML souboru

```

<?xml version="1.0" encoding="utf-8"?>
<autori
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
  xs:noNamespaceSchemaLocation="autori.xsd">

  <prvni>
    Dumas
  </prvni>
  <druhy xs:nil="true"/>
<!-- druha funkcní možnost
  <druhy/>
-->
</autori>

```

- příklad zbytku jiného XML souboru

```

  <prvni>
    Müller
  </prvni>
  <druhy>
    Thurgau
  </druhy>
</autori>

```

9.3.16. Práce s okrajovými bílými znaky

- občas se stane, že validátor hlásí nepochopitelné chyby

- např. pro:

```

<xs:simpleType name="pscType">
  <xs:restriction base="xs:string">
    <xs:pattern value="\d{3} \d{2}" />
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="cisloType">
  <xs:restriction base="xs:positiveInteger">
    </xs:restriction>
</xs:simpleType>

<xs:complexType name="adresaType">
  <xs:sequence>
    <xs:element name="psc" type="pscType"/>
    <xs:element name="cislo" type="cisloType"/>
  </xs:sequence>
</xs:complexType>

  <xs:element name="adresa" type="adresaType"/>

```

■ a XML soubor

```
<?xml version="1.0" encoding="utf-8"?>
<adresa
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
  xs:noNamespaceSchemaLocation="psc-default.xsd">

  <psc>
    123 45
  </psc>
  <cislo>
    789
  </cislo>
</adresa>
```

■ hlásí

```
>xerces -v -s psc-default.xml
[Error] psc-default.xml:8:9: cvc-pattern-valid: Value '
123 45
' is not facet-valid with respect to pattern '\d{3} \d{2}' for type ►
'pscType'.

[Error] psc-default.xml:8:9: cvc-type.3.1.3: The value '
123 45
' of element 'psc' is not valid.
psc-default.xml: 742 ms (3 elems, 1 attrs, 0 spaces, 25 chars)
```

■ po chvíli pokusů zjistíme, že tím, co vadí, je odřádkování před a po hodnotě PSČ a úvodní mezery

• po opravě na:

```
<psc>123 45</psc>
<cislo>
  789
</cislo>
```

■ bude vše v pořádku

• pro element `cislo` jsme ale podobnou úpravu dělat nemuseli

■ vysvětlení je v atributu `xs:whiteSpace`, který může nabývat jedné ze tří hodnot:

1. `preserve` – bílé znaky před, v a za hodnotou zůstávají zcela nedotčeny
2. `replace` – všechny výskyty znaků `#x9` (*tab*), `#xA` (*line feed*) a `#xD` (*carriage return*) jsou nahrazeny znaky `#x20` (*space*)
3. `collapse` – provede se `replace` a následně jsou všechny sekvence mezer nahrazeny pouze jednou mezerou, případná počáteční a koncová mezera se odstraní

■ pro všechna čísla je nastaveno implicitně

```
<xs:whiteSpace value="collapse"/>
```

■ a toto nastavení nelze změnit ani v odvozených typech

- to znamená, že u čísel nezáleží na úvodních a koncových bílých znacích

■ pro řetězce (`base="string"`) je nastaveno

```
<xs:whiteSpace value="preserve"/>
```

- tedy ponechat řetězec přesně tak, jak to je ve zdrojovém XML

- nastavení je možné v odvozených typech měnit

♦ takže pro nastavení:

```
<xs:simpleType name="pscType">
  <xs:restriction base="xs:string">
    <xs:pattern value="\d{3} \d{2}" />
    <xs:whiteSpace value="collapse" />
  </xs:restriction>
</xs:simpleType>
```

♦ je správně např.:

```
<psc>
  123    45
</psc>
```

■ otázkou je, zda je tato změna užitečná

- pro hodnoty atributů v žádném případě, protože jejich hodnoty jsou téměř vždy na stejné řádce jako jejich jména
- pro hodnoty elementů to může mít někdy smysl, např. pokud potřebujeme `string` s restrikcí umístit na novou řádku (viz `psc`)
 - ♦ pak je ale nutné si uvědomit, že aplikace (tj. náš program!) zpracovávající tento XML soubor musí u takového řetězce provést oříznutí počátečních a koncových bílých znaků a náhradu a redukci mezer uvnitř hodnoty, aby vyhovoval vzoru

Kapitola 10. JWDS, StAX, Ant

10.1. JWDS

- Java Web Services Developer Pack 2.0™
- obsahuje klíčové technologie pro zjednodušené vytváření Webových služeb pomocí platformy Java 2
 - Fast Infoset Version 1.0.1
 - **Sun Java Streaming XML Parser Version 1.0 EA**
 - XML and Web Services Security Version 2.0 EA2
 - XML Digital Signature Version 1.0.1
 - **JAXB Version 2.0 EA (Java Architecture for XML Binding)**
 - JAXP Version 1.3.1_01 (Java API for XML Processing)
 - JAXR Version 1.0.8_01 EA (Java API for XML Registries)
 - JAX-WS Version 2.0 EA (Java API for XML Web Services)
 - JAX-RPC Version 1.1.3_01 EA (Java API for XML-based RPC)
 - SAAJ Version 1.3 EA (SOAP with Attachments API for Java)
- **Ant**

10.2. Sun Java Streaming XML Parser (SJSXP)

- je to implementace JSR 173 (*Java Specification Request*) s názvem *Streaming APIs for XML* – StAX
 - kódové jméno Zephyr™
 - založeno na Xerces2
- pokud hledáme informace, je třeba se zajímat o SJSXP nebo StAX nebo `javax.xml.stream` nebo Zephyr nebo `com.sun.xml.stream` nebo JSR 173
- rychlé a jednoduché rozhraní pro sekvenční čtení a zápis XML dokumentů
 - od JDK 1.6 součástí Java Core API
- nabízí dva druhy rozhraní
 - *kurzorové* – nízkourovňové, rychlé
 - `XMLStreamReader` a `XMLStreamWriter`
 - *událostní* – sofistikované
 - `XMLEventReader` a `XMLEventWriter`

Poznámka

dále bude popisováno pouze kurzorové rozhraní

- výhody oproti SAX
 - umí zapisovat a to i velké dokumenty (výhoda oproti DOM)
 - při čtení je kód na jednom místě (u SAX na třech)
 - při čtení je textový obsah elementu vrácen najednou (odpadá postupné skládání)
 - typu *pull-parser*, tzn. čteme na žádost, tj. je možné číst i po částech
 - je nevalidující, tzn. i při explicitně uvedeném XSD nebo DTD v XML dokumentu si těchto souborů nevšimá (nemusejí být dostupné)
- bývalá základní nevýhoda oproti SAX – do JDK 1.6 nebylo součástí Java Core API

10.2.1. Základní postup při zpracování

- přečte XML dokument pomocí kurzorového API

```
import java.io.*;
import java.util.*;
import javax.xml.stream.*;

public class VerifikatorStAX {
//  public static final String SOUBOR = "jidlo-chyba.xml";
  public static final String SOUBOR = "jidlo.xml";

  public static void main(String[] args) {
    try {
      XMLInputFactory f = XMLInputFactory.newInstance();
      XMLStreamReader r = f.createXMLStreamReader(
          new FileReader(SOUBOR));

      int i = 0;
      while (r.hasNext() == true) {
        i++;
        r.next();
      }
      System.out.println("Pocet udalosti: " + i);
    }
    catch (XMLStreamException e) {
      System.out.println("Chyba pri cteni XML souboru");
      e.printStackTrace();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

- objekt třídy `javax.xml.stream.XMLInputFactory` se vytváří klasickou tovární metodou

```
XMLInputFactory f = XMLInputFactory.newInstance();
```

- dále se rozhodneme pro použití kurzorového rozhraní

```
XMLStreamReader r = f.createXMLStreamReader(new FileReader(SOUBOR));
```

- příkaz `r.next()` se posune na další událost v pořadí

- nad objektem `r` lze volat množství metod – viz dále

- problémy při čtení XML souboru lze zachytit obsluhou výjimky `XMLStreamException`

10.2.2. Přehled základních možností čtení

- existuje množství metod, názorný přehled je v tabulce v dokumentaci k `XMLStreamReader`

- základní metoda je `getEventType()`

- vrací typ události, který se dá porovnat s konstantami `XMLStreamConstants`

- ♦ nejdůležitější jsou `ATTRIBUTE`, `START_ELEMENT` a `END_ELEMENT`

- začátek a konec elementu lze otestovat i pomocí `isStartElement()` a `isEndElement()`

- jméno elementu se získá `getLocalName()`

- obsah elementu vrátí `getElementText()`

10.2.2.1. Výpočet celkové váhy

- vypíše celkovou váhu nakoupeného ovoce

```
...
    System.out.println("Celkova vaha: " + getCelkovaVaha(r));
...

static double getCelkovaVaha(XMLStreamReader r) throws Exception {
    double vaha = 0;
    while (r.hasNext() == true) {
        r.next();
        // if (r.getEventType() !=
        //     XMLStreamConstants.START_ELEMENT) {
        if (r.isStartElement() == false) {
            continue;
        }
        if (r.getLocalName().equals("vaha") == true) {
            String v = r.getElementText();
            vaha += Double.parseDouble(v);
        }
    }
    return vaha;
}
```

10.2.3. Zpracování atributů

- hodnota atributu se musí přečíst před případným čtením hodnoty elementu

- opačný postup vyvolá výjimku

- získáváme-li hodnotu atributu podle jeho jména, musí mít první skutečný parametr (s významem `name-spaceURI`) metody `getAttributeValue()` hodnotu `null`

- jsme-li si jisti pořadím elementů, lze použít `nextTag()`

- skočí na počáteční tag dalšího elementu v pořadí

- přeskočí všechny případné komentáře, instrukce pro zpracování apod. a také ukončovací tag právě zpracovávaného elementu

- je to rychlejší, ale méně bezpečný způsob

10.2.3.1. Výpočet celkové ceny

- zpracovává atributy a vypočte celkovou cenu nákupu

```
...
    System.out.println("Celkova cena: " + getCelkovaCena(r));
...

static double getCelkovaCena(XMLStreamReader r) throws Exception {
    double vaha = 0;
    int cena = 0;
    double celkovaCena = 0;

    while (r.hasNext() == true) {
        r.next();
        if (r.isStartElement() == true) {
            if (r.getLocalName().equals("nazev") == true) {
                System.out.println(r.getElementText());
                String a = r.getAttributeValue(null, "jednotkovaCena");
                cena = Integer.parseInt(a);
                System.out.println(r.getElementText());

                r.nextTag(); // preskok na <vaha>
                String v = r.getElementText();
                vaha = Double.parseDouble(v);
                continue;
            }
        }
        if (r.isEndElement() == true
            && r.getLocalName().equals("ovoce") == true) {
            celkovaCena += vaha * cena;
        }
    }
    return celkovaCena;
}
```

10.2.3.2. Všechny objekty v paměti

- uloží všechny hodnoty a atributy najednou do paměti a pak je zpracovává

- třídy `Ovoce` a `ZpracovaniDatVPameti` jsou zcela stejné jako u SAX a DOM

```
...
    ArrayList<Ovoce> ar = getSeznam(r);
    ZpracovaniDatVPameti.tiskniVse(ar);
    System.out.println("Celkova vaha = "
        + ZpracovaniDatVPameti.celkovaVaha(ar));
    System.out.println("Celkova cena = "
        + ZpracovaniDatVPameti.celkovaCena(ar));
...

static ArrayList<Ovoce> getSeznam(XMLStreamReader r) throws Exception {
    ArrayList<Ovoce> ar = new ArrayList<Ovoce>();
    int cislo = 0;
    String nazev = null;
    int jednotkovaCena = 0;;
    double vaha = 0;

    while (r.hasNext() == true) {
        r.next();
        if (r.isStartElement() == true) {
            if (r.getLocalName().equals("ovoce") == true){
                String c = r.getAttributeValue(null, "cislo");
                cislo = Integer.parseInt(c);
            }
            else if (r.getLocalName().equals("nazev") == true) {
                String a = r.getAttributeValue(null, "jednotkovaCena");
                jednotkovaCena = Integer.parseInt(a);
                nazev = r.getElementText();
            }
            else if (r.getLocalName().equals("vaha") == true) {
                String v = r.getElementText();
                vaha = Double.parseDouble(v);
            }
        }

        if (r.isEndElement() == true
            && r.getLocalName().equals("ovoce") == true){
            ar.add(new Ovoce(cislo, nazev, jednotkovaCena, vaha));
        }
    }

    return ar;
}
}
```

10.2.4. Čtení na žádost

- z XML dokumentu lze číst postupně

- StAX si pamatuje pozici posledního čtení

- čtení lze kdykoliv ukončit

- výhodné pro čtení z proudu

Příklad 10.1. Výpis hodnot elementu <nazev>

Program vypisuje postupně (na pokyn uživatele) hodnoty elementů <nazev>. Čtení je možné předčasně ukončit stiskem klávesy k.

```
import java.io.*;
import java.util.*;
import javax.xml.stream.*;

public class NaZadostStAX {
    public static final String SOUBOR = "jidlo.xml";

    public static void main(String[] args) {
        try {
            XMLInputFactory f =
                XMLInputFactory.newInstance();
            XMLStreamReader r = f.createXMLStreamReader(
                new FileReader(SOUBOR));

            while (dalsiNazev() != 'k') {
                String nazev = prectiNazev(r);
                if (nazev == null) {
                    System.out.println("Vstupni soubor je jiz precten");
                    break;
                }
                else {
                    System.out.println("Dalsi ovoce je: " + nazev);
                }
            }
            System.out.println("Konec cteni");
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    static String prectiNazev(XMLStreamReader r)
        throws Exception {
        while (r.hasNext() == true) {
            r.next();
            if (r.isStartElement() == true) {
                if (r.getLocalName().equals("nazev") == true){
                    return r.getElementText();
                }
            }
        }
        return null;
    }

    static char dalsiNazev() throws Exception {
        System.out.print("Stiskni k (=konec) nebo Enter: ");
        byte[] pole = new byte[20];
        System.in.read(pole);
    }
}
```

```
        return (char) pole[0];
    }
}
```

■ vypíše např.:

```
Stiskni k (=konec) nebo Enter:
Dalsi ovoce je: jablka
Stiskni k (=konec) nebo Enter:
Dalsi ovoce je: banány
Stiskni k (=konec) nebo Enter:
Dalsi ovoce je: grapefruity
Stiskni k (=konec) nebo Enter:
Dalsi ovoce je: švestky sušené
Stiskni k (=konec) nebo Enter:
Vstupni soubor je jiz precten
Konec cteni
```

```
Stiskni k (=konec) nebo Enter:
Dalsi ovoce je: jablka
Stiskni k (=konec) nebo Enter:k
Konec cteni
```

10.2.5. Zápis do XML dokumentu

■ generování dokumentu je „přímočaré“

- automaticky převádí & a < na & a <
- lze nastavit použité výstupní kódování
- odřádkování a odsazování elementů nutno provést „ručně“
- další užitečné metody lze nalézt v dokumentaci k XMLStreamWriter

■ výhoda oproti DOM – zapisuje do proudu (*stream*)

- postupné vytváření
- libovolně velký XML dokument

Příklad 10.2.

```
import java.io.*;
import java.util.*;
import javax.xml.stream.*;

public class JidloStAXWrite {
    public static final String SOUBOR = "jidlo-generovano.xml";
    public static final int POCET = 3;

    public static void main(String[] args) {
        try {
            Random r = new Random();
            String kodovani = "utf-8";
            XMLOutputFactory f = XMLOutputFactory.newInstance();
            XMLStreamWriter w = f.createXMLStreamWriter(
                new FileOutputStream(SOUBOR), kodovani);
            w.writeStartDocument(kodovani, "1.0");
            w.writeCharacters("\r\n");
            w.writeComment(" přehled šmakovních dobrůtek ");
            w.writeCharacters("\r\n");
            w.writeStartElement("jidlo");
            for (int i = 1; i <= POCET; i++) {
                w.writeCharacters("\r\n ");
                w.writeStartElement("ovoce");
                w.writeAttribute("cislo", "" + i);
                w.writeCharacters("\r\n ");
                w.writeStartElement("nazev");
                int cena = r.nextInt(40) + 10;
                w.writeAttribute("jednotkovaCena", "" + cena);
                String nazev = "ovoce " + i + " &lt;";
                w.writeCharacters(nazev);
                w.writeEndElement();
                w.writeCharacters("\r\n ");
                w.writeStartElement("vaha");
                double vaha = r.nextDouble() * 10.0;
                String oriznute = String.valueOf(vaha).substring(0, 3);
                w.writeCharacters(oriznute);
                w.writeEndElement();
                w.writeCharacters("\r\n ");
                w.writeEndElement();
            }
            w.writeCharacters("\r\n");
            w.writeEndElement();
            w.writeCharacters("\r\n");
            w.writeEndDocument();
            w.close();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

■ vygeneruje:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- přehled šmakovních dobrůtek -->
<jidlo>
  <ovoce cislo="1">
    <nazev jednotkovaCena="49">ovoce 1 &lt; &lt;</nazev>
    <vaha>1.5</vaha>
  </ovoce>
  <ovoce cislo="2">
    <nazev jednotkovaCena="37">ovoce 2 &lt; &lt;</nazev>
    <vaha>2.9</vaha>
  </ovoce>
  <ovoce cislo="3">
    <nazev jednotkovaCena="29">ovoce 3 &lt; &lt;</nazev>
    <vaha>9.1</vaha>
  </ovoce>
</jidlo>
```

10.3. Ant – Another Neat Tool



10.3.1. Základní informace

■ sestavovací utilita pro multiplatformní použití z `ant.apache.org`

- „nástroj pro mravenčí práci“

■ ve světě vývojářů v Javě všeobecně přijímán

- součást všech významných IDE buď jako plugin (JBuilder, NetBeans) nebo jako vestavěná součást (Eclipse)

`jwsdp-2.0\apache-ant\docs\manual\ide.html`

■ konfigurovatelný a flexibilní systém

■ konzolová aplikace

■ značné množství schopností, významně převyšující možnosti „sestavovacího nástroje“

`jwsdp-2.0\apache-ant\docs\manual\taskoverview.html`

Výstraha

Mnoho stejných akcí lze provést několika rozdílnými způsoby (matoucí).

- podobá se make™ („like Make, but without Make's wrinkles“)

■ základní rozdíly:

1. make

- nastavba příkazového interpreteru – využívá platformově závislé příkazy OS
 - ♦ není přenositelný (`rm` versus `del`)
- používá pro zápis příkazů speciální syntaxi s včetně problematického `<Tab>`

2. Ant

- implementován v Javě za pomoci standardních knihoven
 - ♦ plně přenositelný
- pro zápis příkazů používá formát XML (se všemi jeho výhodami)

10.3.2. Jak Ant získat

- `ant.apache.org` jako jeden .zip soubor, který se pouze rozbalí do `C:\Program Files\Java\ant` (leden 2007 verze 1.7.0.)

- referenční popis (*Ant task reference*) nalezneme v souboru

`ant\docs\appendix_e.pdf`

- pro bezproblémovou činnost je třeba přidat do `PATH` cestu:

`C:\Program Files\Java\ant\bin`

- v manuálu se doporučuje ještě nastavit systémovou proměnnou

`set ANT_HOME="C:\Program Files\Java\ant"`

někdy je nutné nastavit (pokud to již není) `JAVA_HOME`

10.3.3. Použití

- je třeba připravit XML soubor s popisem činností – projekt (`<project>`) (též „sestavovací schéma“)

- jméno souboru je implicitně `build.xml`, který se zpracovává po příkazu

`>ant`

- je výhodné mít pro každý vytvářený projekt jeden soubor

- je možné použít i jiné jméno, pak je spuštění

`>ant -buildfile adresar.xml`

- základní princip je, že `<project>` má minimálně jeden cíl (co se má udělat)

- cíl se označuje `<target>` – logicky oddělená část projektu

- jeden `<target>` může být spuštěn jako defaultní

- v rámci jednoho `<target>` může být víc úkolů (*tasks*)

- úkoly nejsou příkazy OS (byť mnohé mají stejnou syntaxi), ale „služby“ poskytované Antem™

`ant\docs\manual\tasksoverview.html`

Příklad souboru `adresar1.xml`, který v rámci jediného `<target>` (který je navíc defaultní) provede dva úkoly – vytvoří adresář a vypíše zprávu.

`<target>` musí být pojmenován pomocí atributu `name`

Poznámka

Přesto, že názvy úkolů `mkdir` a `echo` jsou stejné, jako příkazy OS, nevykonávají se příkazy OS, ale činnosti z Java tříd.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Adresar" default="vytvorAdresar">
  <target name="vytvorAdresar">
    <mkdir dir="muj-adresar"/>
    <echo message="Hotovo"/>
  </target>
</project>
```

- tento projekt po spuštění vypíše:

```
D:\xml\ant>ant -buildfile adresar1.xml
Buildfile: adresar1.xml
```

```
vytvorAdresar:
[mkdir] Created dir: D:\xml\ant\muj-adresar
[echo] Hotovo
```

```
BUILD SUCCESSFUL
Total time: 0 seconds
D:\xml\ant>
```

- je možné mít více `<target>`

- ty, které nebyly označeny v `<project>` jako default, se spouštějí uvedením svého jména

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Adresar" default="vytvorAdresar">
  <target name="vytvorAdresar">
    <mkdir dir="muj-adresar"/>
    <echo message="Hotovo"/>
  </target>
  <target name="smaz">
    <delete dir="muj-adresar"/>
  </target>
</project>
```

■ tento projekt po spuštění vypíše:

```
D:\xml\ant>ant -buildfile adresar2.xml smaz
Buildfile: adresar2.xml

smaz:
[delete] Deleting directory D:\xml\ant\muj-adresar

BUILD SUCCESSFUL
Total time: 0 seconds
D:\xml\ant>
```

■ pokud bychom soubor projektu přejmenovali na build.xml, bylo by spouštění:

```
D:\xml\ant>ant
Buildfile: build.xml

vytvorAdresar:
[mkdir] Created dir: D:\xml\ant\muj-adresar
[echo] Hotovo

BUILD SUCCESSFUL
Total time: 0 seconds
D:\xml\ant>ant smaz
Buildfile: build.xml

smaz:
[delete] Deleting directory D:\xml\ant\muj-adresar

BUILD SUCCESSFUL
Total time: 0 seconds
D:\xml\ant>
```

10.3.3.1. Atributy <project>

name="Adresar"

■ nepovinný atribut, který má význam pouze pro přehlednost

- doporučuje se uvádět – soubor se typicky jmenuje build.xml, takže z jeho názvu není jasné, k čemu je projekt určen

default="vytvorAdresar"

■ povinný atribut, označuje nejpravděpodobnější/nejčastěji používaný <target>

basedir="."

■ je to základní adresář pro všechny relativní cesty uváděné dále

- jednoduchý, ale velmi účinný způsob, jak udržet pořádek v umístění souborů

■ nepovinný atribut, ale velmi často uváděný

- není-li uveden, má implicitní hodnotu ".", tj. „aktuální adresář“

```
<project name="Adresar" default="vytvorAdresar"
        basedir="d:\zzz">
  <target name="vytvorAdresar">
    <mkdir dir="muj-adresar"/>
    <echo message="Hotovo"/>
  </target>
</project>
```

```
D:\xml\ant>ant -buildfile adresar3.xml
Buildfile: adresar3.xml
```

```
vytvorAdresar:
[mkdir] Created dir: D:\zzz\muj-adresar
[echo] Hotovo
```

BUILD SUCCESSFUL

Poznámka

Jako oddělovač adresářů lze použít \ i / případně je libovolně míchat

10.3.4. Použití <property>

■ element dává možnost nastavit (typicky na začátku) symbolické konstanty

- používají se dále s výhodami symbolických konstant (nastavení jen jednou, čitelnost, atd.)

Výstraha

Jedná se o skutečné konstanty, které se po nastavení nedají měnit.

■ nastavení obecné hodnoty

```
<property name="jmeno" value="hodnota">
```

■ použití je

```
${jmeno}
```

■ konstant může být libovolné množství, každé <property> nastavuje jednu

Poznámka

vedlejším efektem příkladu jsou dva různé způsoby výpisu textu na konzoli

```
<echo message="zprava"/>
<echo>zprava</echo>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Konstanty" default="vytvorAdresar">
  <property name="adresar" value="muj-adresar"/>
  <property name="podadresar" value="vnoreny"/>

  <target name="vytvorAdresar">
    <mkdir dir="${adresar}/${podadresar}"/>
    <echo message="${adresar}/${podadresar}"/>
    <echo>${adresar}/${podadresar}</echo>
  </target>

  <target name="smaz">
    <delete dir="${adresar}/${podadresar}"/>
    <delete dir="${adresar}"/>
  </target>
</project>
```

10.3.4.1. Nastavení hodnoty ve smyslu „jméno souboru nebo adresáře“

- místo value použijeme location

```
<property name="jmeno" location="jmeno">
```

- je-li jmeno úplná cesta, zůstává při použití nezměněna

- je-li to neúplná cesta, použije se v součinnosti s basedir

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Location" default="loc" basedir=".">
  <property name="jmeno-rel" location="src" />
  <property name="jmeno-abs" location="d:\zzz" />

  <target name="loc">
    <echo message="${jmeno-rel}"/>
    <echo message="${jmeno-abs}"/>
  </target>
</project>
```

- vypíše:

```
D:\xml\ant>ant -buildfile location.xml
Buildfile: location.xml
loc:
    [echo] D:\xml\ant\src
    [echo] D:\zzz
BUILD SUCCESSFUL
```

10.3.4.2. Pozor na skládání jmen adresářů

- location se jeví jako vhodný nástroj pro konečná jména adresářů

- při skládání jmen adresářů je ale zcela nevhodný

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Location-chyba" default="loc"
  basedir=".">
  <property name="adresar" location="muj-adresar"/>
  <property name="podadresarLoc" location="vnoreny"/>
  <property name="podadresarVal" value="vnoreny"/>

  <target name="loc">
    <echo message="${adresar}/${podadresarLoc}"/>
    <echo message="${adresar}/${podadresarVal}"/>
  </target>
</project>
```

- vypíše:

```
D:\xml\ant>ant -buildfile location-chyba.xml
Buildfile: location-chyba.xml
loc:
    [echo] D:\xml\ant\muj-adresar/D:\xml\ant\vnoreny
    [echo] D:\xml\ant\muj-adresar/vnoreny
BUILD SUCCESSFUL
```

10.3.4.3. Přednastavené konstanty

- kromě námi pojmenovaných konstant umožňuje <property> použít i přednastavené konstanty

Jsou to:

1. file

- nastavení jmen a hodnot konstant z externího souboru

- nemusíme pak měnit soubor projektu

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Konstanty externe" default="externe">
  <property file="adresare.properties" prefix="adr"/>

  <target name="externe">
    <echo message=
      "vytvarim adresar: ${adr.adresar}/${adr.podadresar}"/>
  </target>
</project>
```

- soubor s nastavením vlastností má nejčastěji příponu .properties (ale není to nezbytné)

- obsah souboru adresare.properties:

```
adresar="muj-adresar"
podadresar=vnoreny
```

Výstraha

V nastavení hodnoty se nesmí používat uvozovky (ve smyslu začátek a konec hodnoty). Vše, co je za znakem = až do konce řádky je hodnota konstanty

■ po spuštění vypíše:

```
externe:
[echo] vytvarim adresar: "muj-adresar"/vnoreny
```

2. environment

■ pro práci se systémovými proměnnými

Výstraha

Jména systémových proměnných jsou *case-sensitive*. Nesouhlasí-li, vypíše se použité jméno, nikoliv hodnota.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Konstanty prednastavene" default="prednastavene">
  <property environment="e"/>

  <target name="prednastavene">
    <echo>${e.Path}</echo>
  </target>
</project>
```

10.3.5. Nastavování cest a opětovné použití nastavení

■ pro nastavení cest (názvů několika adresářů vzájemně oddělených „;“ nebo „:“) použijeme <path>

- je vhodné vytvářenou cestu identifikovat pomocí atributu `id`
- ♦ pak lze snadno vytvářet odkazy na tuto cestu kdekoliv jinde pomocí `refid`

■ skutečný oddělovač adresářů („;“ nebo „:“) si Ant doplní dle konvencí konkrétní platformy

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="NastaveniCesty" default="vypis">
  <path id="mujClasspath">
    <pathelement path="C:\Program Files\java\"/>
    <pathelement location="prelozene/knihovna.jar"/>
  </path>

  <property name="cesta" refid="mujClasspath" />

  <target name="vypis">
    <echo message="${cesta}" />
  </target>
</project>
```

■ vypíše:

```
D:\xml\ant>ant -buildfile classpath.xml
Buildfile: classpath.xml
```

vypis:

```
[echo] C:\Program Files\java;D:\xml\ant\prelozene\knihovna.jar
```

Poznámka

1. potřebujeme-li sofistikované nastavení cesty, kdy nevystačíme s <pathelement>, použijeme <dirset> ve stejném duchu, jako <fileset> – viz dále
2. na základní úrovni (<target>) neexistuje <classpath>

10.3.6. Nastavení jmen souborů

■ používáme <fileset> a opět je vhodné nastavit atribut `id` pro pozdější `refid`

■ soubory, které přidáváme do seznamu, zapíšeme pomocí elementu <include> (lze použít i atribut `includes`)

■ je možné používat masku:

- *.java – všechny soubory .java v zadaném adresáři
- **/*.java – všechny soubory .java v zadaném adresáři a ve všech vnořených podadresářích
- ze seznamu lze vyloučit soubory pomocí elementu <exclude>
- lze použít i atribut `excludes`, kde může být seznam vyloučených souborů oddělených čárkou nebo mezerou

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="NastaveniSouboru" default="vypis">
  <property name="adresar" location="." />

  <fileset id="mojeSoubory"
    dir="${adresar}"
    excludes="a*.xml, p*.xml, z*.xml">
    <include name="**/*.java" />
    <include name="*.xml" />
    <exclude name="build.xml" />
    <exclude name="i*.xml" />
  </fileset>

  <property name="soubory" refid="mojeSoubory" />

  <target name="vypis">
    <echo message="${soubory}" />
  </target>
</project>
```

■ vypíše:

```
D:\xml\ant>ant -buildfile soubory.xml
Buildfile: soubory.xml
```

```
vypis:
    [echo] Akcenty.java;build3.xml;classpath.xml;
        location.xml;optional.xml;soubory.xml;
        src\LideSAX.java;src\ObsluhaChyb.java
```

```
BUILD SUCCESSFUL
```

Poznámka

Jinou možností je použití `<filelist>`

10.3.7. Kombinování `<path>` a `<fileset>`

- `<path>` často vnitřně využívá `<fileset>`

- v `<task>`, který potřebuje `classpath`, se často tato hodnota nastavuje pomocí `refid`

- je samozřejmě možné `classpath` nastavit lokálně s využitím `<fileset>` a/nebo `<dirset>`

- příklad ukazuje nastavení `classpath` pro překlad pomocí `javac`™ s využitím knihoven JAXB

```
<property environment="e"/>
<property name="jwsdp" value="{e.JWSDP_HOME}"/>

<path id="classpathProJAXB">
  <pathelement path="{adresarClassSouboru}" />
  <fileset dir="{jwsdp}"
    includes="jaxb/lib/*.jar" />
  <fileset dir="{jwsdp}"
    includes="jwsdp-shared/lib/*.jar" />
</path>

<target name="generovani">
  <javac>
    <classpath refid="classpathProJAXB" />
```

10.3.8. Možnosti `<target>`

10.3.8.1. Seznam `<target>`

- máme-li více `<target>`, můžeme si jejich seznam vypsát použitím příkazu:

```
-projecthelp
```

```
D:\xml\ant>ant -buildfile zavislosti.xml -projecthelp
Buildfile: zavislosti.xml
```

```
Main targets:
```

```
Other targets:
druhy
prvni
treti
Default target: prvni
```

10.3.8.2. Vzájemné závislosti

- jednotlivé `<target>` mohou záviset pořadím provádění na jiných `<target>`

- závislosti se mohou řetěžit

- používá se atribut `depends`

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Zavislosti" default="prvni">
  <target name="prvni" depends="druhy">
    <echo message="prvni = defaultni"/>
  </target>

  <target name="druhy" depends="treti">
    <echo message="druhy"/>
  </target>

  <target name="treti">
    <echo message="treti"/>
  </target>
</project>
```

- vypíše:

```
D:\xml\ant>ant -buildfile zavislosti.xml
Buildfile: zavislosti.xml
treti:
    [echo] tretí

druhy:
    [echo] druhy

prvni:
    [echo] prvni = defaultni

BUILD SUCCESSFUL
```

- je možné konstruovat složité závislosti, ale snaha je o přímočaré souvislosti

- zacyklí-li se závislosti, Ant vypíše chybové hlášení

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Zavislosti" default="prvni">
  <target name="prvni" depends="druhy">
    <echo message="prvni = defaultni"/>
```

```

</target>

<target name="druhy" depends="prvni">
  <echo message="druhy"/>
</target>
</project>

```

■ vypíše:

```

D:\xml\ant>ant -buildfile zavislosti-cyklus.xml
Buildfile: zavislosti-cyklus.xml

```

```

BUILD FAILED
Circular dependency: prvni <- druhy <- prvni

```

10.3.8.3. Podmíněné provádění <target>

■ bez podmínek je <target> proveden vždy při vyvolání

- lze ale nastavit jeho provádění v závislosti na **existenci** <property>

Výstraha

nezávisí na **hodnotě** <property>

■ používá se dvojice příkazů if – unless (ve smyslu if – else)

```

<?xml version="1.0" encoding="UTF-8"?>
<project name="Podminky" default="druhy">
  <property name="povoleni" value="nezalezi"/>

  <target name="prvni" if="povoleni">
    <echo message="prvni"/>
  </target>

  <target name="druhy" unless="povoleni">
    <echo message="druhy"/>
  </target>
</project>

```

■ vypíše:

```

D:\xml\ant>ant -buildfile if-unless1.xml prvni
Buildfile: if-unless1.xml

```

```

prvni:
    [echo] prvni

```

```

BUILD SUCCESSFUL

```

```

D:\xml\ant>ant -buildfile if-unless1.xml druhy

```

```

Buildfile: if-unless1.xml

```

```

druhy:

```

```

BUILD SUCCESSFUL

```

■ podmínky se dají kombinovat se závislostmi

- je třeba zvýšené opatrnosti

```

<?xml version="1.0" encoding="UTF-8"?>
<project name="Podminky" default="druhy">
  <property name="povoleni" value="nezalezi"/>

  <target name="prvni" if="povoleni">
    <echo message="prvni"/>
  </target>

  <target name="druhy" depends="prvni"
    unless="povoleni">
    <echo message="druhy"/>
  </target>
</project>

```

■ vypíše:

```

D:\xml\ant>ant -buildfile if-unless2.xml
Buildfile: if-unless2.xml

```

```

prvni:
    [echo] prvni

```

```

druhy:

```

```

BUILD SUCCESSFUL

```

■ nesprávně nastavená podmínka v kombinaci se závislostí

- prvni očekává, že před svým spuštěním bude mít vykonanou činnost druhy, což se ale neprovede

```

<?xml version="1.0" encoding="UTF-8"?>
<project name="Podminky" default="prvni">
  <property name="povoleni" value="nezalezi"/>

  <target name="prvni" depends="druhy" if="povoleni">
    <echo message="prvni = defaultni"/>
  </target>

  <target name="druhy" unless="povoleni">
    <echo message="druhy"/>
  </target>
</project>

```

■ **vypiše:**

```
D:\xml\ant>ant -buildfile if-unless3.xml
Buildfile: if-unless3.xml
```

```
druhy:
```

```
prvni:
    [echo] prvni = defaultni
```

```
BUILD SUCCESSFUL
```

10.3.9. Když je něco špatně

■ v případech, kdy Ant™ neprovádí očekávanou činnost, je vhodné použít další parametr příkazové řádky

- podrobnější informace o průběhu `-verbose`

```
D:\xml\ant>ant -buildfile if-unless3.xml -verbose
...
druhy:
Skipped because property 'povoleni' set.
...
```

■ dalším možným parametrem je `-debug`, který vypisuje ještě podrobnější výstup

```
D:\xml\ant>ant -buildfile if-unless3.xml -debug
...
Setting project property: povoleni -> nezalezi
Build sequence for target `prvni' is [druhy, prvni]
Complete build sequence is [druhy, prvni, ]

druhy:
Skipped because property 'povoleni' set.
...
```

10.3.10. Úkoly (tasks)

■ jsou to jednotlivé příkazy v rámci jednoho `<target>`

Jsou tři typů:

1. vestavěné (*built-in*, v dokumentaci nazývané „*core tasks*“)

- např. `<javac>`
- pracují bez jakýchkoliv dalších knihoven

2. volitelné (*optional*)

■ např. `<XmlValidate>`

■ jsou součástí distribuce Ant™ (a také kompletně popsané v dokumentaci), ale ke své práci potřebují typicky nějakou externí knihovnu

- knihovny jsou popsány v *Library Dependencies*

■ některé z nich mohou fungovat okamžitě, protože externí knihovna je součástí např. JDK

- např. `<native2ascii>`

3. vlastní

■ napíšeme si je dle potřeby jako třídy v Javě

`ant\docs\manual\tutorial-writing-tasks.html`

■ přicházejí s produkty třetích stran

■ celkově je úkolů z 1. a 2. skupiny velké množství (více než 100)

• v dokumentaci se uvádí dělení do těchto základních skupin:

Archive Tasks

Audit/Coverage Tasks

Compile Tasks

Deployment Tasks

Documentation Tasks

EJB Tasks

Execution Tasks

File Tasks

Java2 Extensions Tasks

Logging Tasks

Mail Tasks

Miscellaneous Tasks

.NET Tasks

Pre-process Tasks

Property Tasks

Remote Tasks

SCM Tasks

Testing Tasks

10.3.11. Přehled a použití často používaných úkolů

10.3.11.1. <copy> – kopírování souborů a adresářů

lze kopírovat

- soubor do téhož adresáře pod jiným jménem

```
<copy file="stary.txt" tofile="novy.txt"/>
```

- soubor do jiného adresáře pod stejným jménem

```
<copy file="stary.txt" todir="D:\zzz"/>
```

- adresář (včetně vnořených podadresářů)

```
<copy todir="D:\zzz">
  <fileset dir="."/>
</copy>
```

- využití již definovaného seznamu souborů pomocí refid

```
<copy todir="D:\zzz\kopie">
  <fileset refid="mojeSoubory" />
</copy>
```

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<project name="copy" default="copy">
  <property name="adresar" location="." />
```

```
  <fileset id="mojeSoubory"
    dir="${adresar}"
    excludes="a*.xml, p*.xml z*.xml">
    <include name="**/*.java" />
    <include name="*.xml" />
    <exclude name="build.xml" />
    <exclude name="i*.xml" />
  </fileset>
```

```
<target name="copy">
  <copy file="stary.txt" tofile="novy.txt"/>
```

```
  <copy file="stary.txt" todir="D:\zzz"/>
```

```
  <copy todir="D:\zzz">
    <fileset dir="."/>
  </copy>
```

```
<copy todir="D:\zzz\kopie">
  <fileset refid="mojeSoubory" />
</copy>
</target>
</project>
```

10.3.11.2. <delete>

- mazání souborů, adresářů i vnořených podadresářů

- podobně jako u <copy> lze použít pro označení skupiny mazaných souborů <fileset> i pomocí refid

Výstraha

Při použití <fileset> se nebere v potaz nastavení adresáře pomocí atributu dir

- pro mazání prázdných podadresářů lze zvolit atribut

```
includeEmptyDirs="true"
```

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="delete" default="delete">
  <property name="adresar" location="." />
```

```
  <fileset id="mojeSoubory" dir="${adresar}"
    excludes="a*.xml, p*.xml z*.xml">
    <include name="**/*.java" />
    <include name="*.xml" />
    <exclude name="build.xml" />
    <exclude name="i*.xml" />
  </fileset>
```

```
<target name="delete">
  <delete file="novy.txt"/>
```

```
  <delete dir="D:\zzz\src"/>
```

```
<!--
```

```
  <delete dir="D:\zzz\kopie">
    <fileset refid="mojeSoubory" />
  </delete>
```

```
-->
```

```
  <delete includeEmptyDirs="true">
    <fileset dir="D:\zzz" includes="**/*.bak" />
  </delete>
```

```
</target>
```

```
</project>
```

10.3.11.3. <mkdir>

- vytvoří adresář

- jedná-li se o více vnořených adresářů, vytvoří najednou všechny případné chybějící

- jako oddělovač adresářů lze použít / i \

- často se používá `<property>`

```
<mkdir dir="${pocatecni}\vnor1\vnor2\${koncovy}"/>
```

- příklad viz výše

10.3.11.4. `<move>`

- stejné použití jako u `<copy>`, pouze soubory přesune

10.3.11.5. `<javac>`

- překlad `.java` souborů

- `srcdir` je prohledáván rekurzivně (překlad balíků i podbalíků)

- překládají se chybějící `.class` a `.class`, které jsou starší než zdrojové `.java`

- minimální verze je

```
<javac srcdir="."/>
```

- existuje velké množství přepínačů, většinou vystačíme jen s několika

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="javac" default="javac">
  <property environment="e"/>
  <property name="jwsdp" value="${e.JWSDP_HOME}"/>
  <path id="classpathProJAXB">
    <pathelement path="." />
    <fileset dir="${jwsdp}"
      includes="jaxb/lib/*.jar" />
    <fileset dir="${jwsdp}"
      includes="jwsdp-shared/lib/*.jar" />
  </path>

  <target name="javac">
    <javac srcdir="."/>

    <mkdir dir="./class" />
    <javac srcdir="."
      destdir="./class"
      includes="Zakladni.java src/balik/*.java">
      <compilerarg value="-Xlint" />
    </javac>

    <delete includeEmptyDirs="true">
      <fileset dir="." includes="**/*.class" />
    </delete>

    <javac debug="on">
      <src path="." />
    </javac>
  </target>
</project>
```

```
<!--      <dst path="." />
nelze
-->

    <classpath refid="classpathProJAXB" />
    </javac>

  </target>
</project>
```

- vypíše pro zdrojové soubory

```
../Zakladni.java"
```

```
public class Zakladni {
  public static void main(String[] args) {
    System.out.println("Zakladni");
  }
}
```

```
../src/balik/Hlavni.java"
```

```
package balik;
public class Hlavni {
  public static void main(String[] args) {
    System.out.println("Hlavni v baliku");
  }
}
```

```
D:\xml\ant>ant -buildfile javac.xml
Buildfile: javac.xml
```

```
javac:
[javac] Compiling 2 source files
[javac] Compiling 2 source files to D:\xml\ant\class
[delete] Deleting 4 files from D:\xml\ant
[javac] Compiling 2 source files
```

```
BUILD SUCCESSFUL
```

10.3.11.6. `<java>`

- spuštění Java programu

- programy `Zakladni` a `Hlavni` z adresare `./class` by se z příkazové řádky spouštěly:

```
D:\xml\ant>java -cp ./class Zakladni
Zakladni
```

```
D:\xml\ant>java -cp ./class balik.Hlavni
Hlavni v baliku
```

- potřebujeme-li předat JVM nějaké parametry pro spuštění, musíme spustit další instanci JVM pomocí atributu `fork="true"`

- chceme-li spustit `.jar` soubor, musíme opět použít `fork="true"`

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="java" default="java">

  <target name="java">
    <java classname="Zakladni" />

    <java classname="Zakladni" classpath="./class" />

    <java classname="balik.Hlavni">
      <classpath path="./class" />
    </java>

    <java classname="balik.Hlavni"
      classpath="./class"
      fork="true">
      <jvmarg value="-Xmx300M"/>
    </java>

    <java jar="hlavni.jar"
      fork="true">
    </java>
  </target>
</project>
```

- vypíše:

```
D:\xml\ant>ant -buildfile java.xml
Buildfile: java.xml
```

```
java:
 [java] Zakladni
 [java] Zakladni
 [java] Hlavni v baliku
 [java] Hlavni v baliku
 [java] Hlavni v baliku
```

```
BUILD SUCCESSFUL
```

10.3.11.7. <javadoc>

- generování dokumentace

- má značné množství parametrů, pro základní použití stačí jen několik

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="javadoc" default="javadoc">
  <target name="javadoc">
    <javadoc sourcepath="./src"
```

```
      <destfile="doc"
        windowtitle="MujBalik">
        <fileset dir="./src"
          includes="**/*.java" />
        </javadoc>
      </target>
</project>
```

- vypíše:

```
D:\xml\ant>ant -buildfile javadoc.xml
Buildfile: javadoc.xml
```

```
javadoc:
 [javadoc] Generating Javadoc
 [javadoc] Javadoc execution
 [javadoc] Loading source file D:\xml\ant\src\balik\Hlavni.java...
 [javadoc] Loading source files for package balik...
 [javadoc] Constructing Javadoc information...
 [javadoc] Standard Doclet version 1.6.0
 [javadoc] Building tree for all the packages and classes...
 [javadoc] Building index for all the packages and classes...
 [javadoc] Building index for all classes...
```

```
BUILD SUCCESSFUL
```

10.3.11.8. <jar>

- vytvoří `.jar` soubor

- jméno (a případné umístění v adresářích) `.jar` souboru musí být v atributu

```
destfile="aplikace1.jar"
```

(občas je někde vidět `jarfile` místo `destfile`)

- soubory, které mají být do `.jar` přidány, lze specifikovat pomocí `<fileset>`

Výstraha

počáteční adresář z `<fileset>` se NEVkládá do `.jar` (tzn. v `.jar` je o jednu úroveň adresářů méně)

- vždy vytvoří soubor `MANIFEST.MF` v podadresáři `META-INF`

- soubory do `.jar` lze určit i pomocí atributu `basedir`

- vytvářený `MANIFEST.MF` lze doplnit pomocí `<manifest>` a vytvořit přímo spustitelný `.jar`

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="jar" default="jar">
```

```
  <target name="jar">
```

```

<jar destfile="aplikace1.jar">
  <fileset dir="./class" />
  <fileset dir="./src" />
</jar>

<jar destfile="aplikace2.jar"
  basedir="."
  excludes="*. *" >
</jar>

<jar destfile="hlavni.jar"
  basedir="./class">
  <manifest>
    <attribute name="Main-Class"
      value="balik.Hlavni"/>
  </manifest>
</jar>
</target>
</project>

```

10.3.11.9. Časová známka <tstamp>

- výhodná v případě, že názvy souborů nebo adresářů závisejí na aktuálním datumu/čase
- přednastaví tři property: DSTAMP, TSTAMP a TODAY
- lze si nastavit i vlastní property pomocí <format>

```

<?xml version="1.0" encoding="UTF-8"?>
<project name="tstamp" default="tstamp">
  <target name="tstamp">
    <tstamp/>
    <echo message="DSTAMP=${DSTAMP}" />
    <echo message="TSTAMP=${TSTAMP}" />
    <echo message="TODAY=${TODAY}" />

    <tstamp>
      <format property="MujDatum"
        pattern="yyyy-MM-dd" />
      <format property="MujCas"
        pattern="HH:mm:ss" />
    </tstamp>

    <echo message="MujDatum=${MujDatum}" />
    <echo message="MujCas=${MujCas}" />

    <mkdir dir="distribuce${MujDatum}" />

  </target>
</project>

```

- vypíše:

```

D:\xml\ant>ant -buildfile tstamp.xml
Buildfile: tstamp.xml

```

```

tstamp:
  [echo] DSTAMP=20070121
  [echo] TSTAMP=1822
  [echo] TODAY=January 21 2007
  [echo] MujDatum=2007-01-21
  [echo] MujCas=18:22:58
  [mkdir] Created dir: D:\xml\ant\distribuce2007-01-21

```

BUILD SUCCESSFUL

10.3.11.10. <native2ascii> – optional task

- vhodný v případě, kdy budeme šířit svoje zdrojové soubory s i18n (*internationalization*)
- atribut ext udává příponu nově vzniklých souborů

```

<?xml version="1.0" encoding="UTF-8"?>
<project name="native2ascii" default="n2a">
  <target name="n2a">
    <native2ascii encoding="CP1250"
      src="."
      dest="./konverze"
      includes="**/*.java"
      ext=".java"/>
  </target>
</project>

```

10.3.12. XJC pro JDK 1.6

- XJCTask (z JDK 1.5) přestal být distribuován a nahradil jej přímo spustitelný soubor C:\Program Files\Java\jdk1.6.0\bin\xjc.exe
- nepřiliš šťastné řešení
- žádný návod k použití v Antu v oficiální dokumentaci
- použití:

```

<property name="java"
  value="${e.JAVA_HOME}" />

<target name="generovani">
  <mkdir dir="${adresarGenerovanychSouboru}" />
  <!-- pomoci xjc.exe generuje soubory z .xsd souboru -->
  <exec dir="." executable="${java}\bin\xjc">
    <arg line="*.xsd"/>
    <arg line="-d ${adresarGenerovanychSouboru}" />
    <arg line="-p ${nazevBalikuGenerovanychTrid}" />
  </exec>

```

10.3.13. Ukázka komplexního projektu použitelného v praxi

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Univerzalni"
    default="ladeni"
    basedir=".">
    <description>
        Soubor pro veskerou praci s Java soubory na prikazove radce
        P.Herout, 2007

        prikazy:
            init          - vytvori pocatecni adresare
            preklad       - preklad zdrojovych souboru
            ladeni        - spusteni programu (defaultni prikaz)
            dokumentace   - vytvori dokumentaci
            distribuce    - zabali vse do spustitelneho .jar
            uklid         - vymaze vse krome zdrojovych souboru
    </description>

    <!-- properties menene vzdy -->
    <property name="balik"          value="balik"/>
    <property name="souborSMain"    value="Hlavni"/>
    <property name="distribucniJAR" value="aplikace"/>

    <!-- properties menene dle platformy -->
    <property name="kodovaniZdroje" value="windows-1250"/>
    <!-- value="ISO-8859-2" -->

    <!-- properties menene temer nikdy -->
    <property name="zdrojove"      value="src"/>
    <property name="prelozene"     value="class"/>
    <property name="dokumentace"   value="doc"/>
    <property name="distribuce"    location="dist"/>

    <target name="init">
        <mkdir dir="${zdrojove}/${balik}"/>
        <mkdir dir="${prelozene}"/>
    </target>

    <target name="preklad" depends="init" >
        <javac srcdir="${zdrojove}" destdir="${prelozene}"
            debug="on"/>
    </target>

    <target name="ladeni" depends="preklad" >
        <java classname="${balik}.${souborSMain}">
            <classpath path="${prelozene}" />
        </java>
    </target>

    <target name="dokumentace" >
```

```
        <javadoc sourcepath="${zdrojove}"
            destdir="${dokumentace}"
            windowtitle="${balik}">
            <fileset dir="${zdrojove}"
                includes="**/*.java" />
        </javadoc>
    </target>

    <target name="distribuce" depends="preklad, dokumentace" >
        <mkdir dir="${distribuce}/${zdrojove}"/>
        <mkdir dir="${distribuce}/${dokumentace}"/>

        <native2ascii encoding="${kodovaniZdroje}"
            src="${zdrojove}"
            dest="${distribuce}/${zdrojove}"
            includes="**/*.java"
            ext=".java"/>

        <copy todir="${distribuce}">
            <fileset dir="${prelozene}"/>
        </copy>

        <copy todir="${distribuce}/${dokumentace}">
            <fileset dir="${dokumentace}"/>
        </copy>

        <tstamp>
            <format property="MujDatum"
                pattern="yyyy-MM-dd"/>
        </tstamp>

        <jar destfile="${distribucniJAR}${MujDatum}.jar"
            basedir="${distribuce}">
            <manifest>
                <attribute name="Main-Class"
                    value="${balik}.${souborSMain}"/>
            </manifest>
        </jar>

        <delete dir="${distribuce}"/>
    </target>

    <target name="uklid" >
        <delete dir="${prelozene}"/>
        <delete dir="${dokumentace}"/>
        <!-- maze distribucni .jar !!! -->
        <delete>
            <fileset dir="." includes="*.jar" />
        </delete>
    </target>
</project>
```

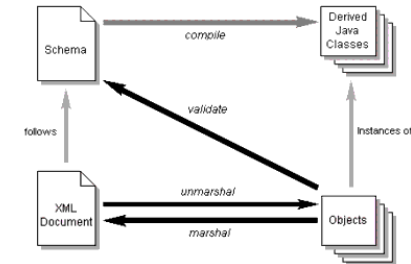
10.3.14. Doporučeno k přečtení:

ant\docs\ant_in_anger.html

Kapitola 11. *Java Architecture for XML Binding* – JAXB

11.1. Základní informace

- od JDK 1.6 součástí Java Core API
 - stále se vyvíjí, nyní verze JAXB 2.0
- automatické mapování mezi XML dokumentem a odpovídajícími Java třídami přes XSD



- využívá se faktu, že struktura XML dokumentu byla již jednou detailně popsána v .XSD souboru
 - ♦ není tedy nutné připravovat obdobné třídy v Javě
 - ty mohou vzniknou automatickou generací
- vhodné pro XML dokumenty, kde:
 - známe dopředu schéma
 - obsahují hodně strukturované nebo opakující se informace
 - potřebujeme XML načítat i upravovat
 - nejsou příliš rozsáhlé (MB)
- nástroje a API JAXB dovolují
 - jednorázově vygenerovat API našich tříd na základě XSD
 - načíst XML dokument do paměti do objektového modelu (*unmarshal*), který přesně odpovídá struktuře XML dokumentu
 - objekty v paměti lze editovat a validovat
 - ♦ jsme zcela odstíněni od XML formátu
 - objektový model lze zapsat do XML dokumentu (*marshal*)
- návod k použití (a asi 15 příkladů) lze nalézt v JWSDP tutoriálu (java.sun.com) a nainstalovat:

```
JWSDP-tutorial\doc\index.html
```

- do `.java` zdrojového souboru musíme importovat

```
import javax.xml.bind.*;
```

- případně i:

```
import javax.xml.bind.util.*;
```

11.1.1. Podpora z Ant

- pro práci je velmi vhodné využít Ant

- soubor `build.xml` má čtyři cíle (*target*)
 - ♦ generování – proběhne pouze jednou na začátku práce
 - ♦ ladění – spouští se po každé úpravě zdrojového kódu v Javě (*default target*)
 - ♦ mazání – proběhne pouze jednou na konci práce
 - ♦ distribuce – proběhne pouze jednou na konci práce

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<project name="Prace s JAXB"
  default="ladeni"
  basedir=".">
```

```
<description>
  Univerzální generacní, překladací a ladící
  předpis pro JAXB
  pro JDK 1.6 -- xjc.exe
  P.Herout, leden 2007
```

```
Spusteni:
>ant generovani -- z .xsd vygeneruje prislusne
                  tridy v Jave, prelozi je
                  a pripravi k nim dokumentaci

>ant              -- preklada a spusti ladenou
                  aplikaci

>ant mazani       -- smaze vsechny generovane
                  soubory a adresare

>ant distribuce -- vyrobi primo spustitelny .jar
</description>
```

```
<!-- Nutno zmenit vzdy -->
<property name="souborSFunkci_main()_ladeneAplikace"
  value="VahaJidloJAXB"/>
<property name="nazevBalikuLadeneAplikace"
  value=""/> <!-- pokud neni prazdny,
               na konci musi byt tecka,
               napr. "balik." -->
```

```
<property name="nazevBalikuGenerovanychTrid"
  value="jidlobalik"/>
```

```
<!-- Menit jen pokud nevyhovuji jmena
      generovanych adresaru -->
<property name="adresarZdrojovychSouboru"
  value="."/>
<property name="adresarGenerovanychSouboru"
  value="generovano"/>
<property name="adresarClassSouboru"
  value="."/>
<property name="adresarGenerovaneDokumentace"
  value="docsapi"/>
```

```
<!-- Od tohoto mista nic nemenit,
      pokud presne nevite, co delate -->
<property environment="e"/>
<property name="java"
  value="${e.JAVA_HOME}"/>

<!--
Nastaveni:
<property name="jwsdp"
<path id="classpathProJAXB"
      - nejsou treba, JAXB je soucasti Java Core API
-->
```

```
<target name="generovani">
  <mkdir dir="${adresarGenerovanychSouboru}" />
  <!-- task xjc pro JDK1.6 nefunguje -->
  <!-- pomoci xjc.exe generuje soubory z .xsd souboru -->
  <exec dir="." executable="${java}\bin\xjc">
    <arg line="*.xsd"/>
    <arg line="-d ${adresarGenerovanychSouboru}"/>
    <arg line="-p ${nazevBalikuGenerovanychTrid}"/>
  </exec>
```

```
<!-- prelozi generovane soubory -->
<mkdir dir="${adresarClassSouboru}" />
<javac
  srcdir="${adresarGenerovanychSouboru}"
  destdir="${adresarClassSouboru}"
  debug="on">
</javac>
```

```
<!-- vyrobi dokumentaci k vygenerovanym souborum -->
<mkdir dir="${adresarGenerovaneDokumentace}" />
<javadoc
  sourcepath="${adresarGenerovanychSouboru}"
  destdir="${adresarGenerovaneDokumentace}"
  windowtitle="${nazevBalikuGenerovanychTrid}"
  useexternalfile="yes">
  <fileset
    dir="${adresarGenerovanychSouboru}"
```

```

        includes="**/*.java"
        excludes="**/impl/**/*.*" />
</javadoc>

<!-- vyrobi .jar z vygenerovanych souboru -->
<jar
    destfile="${navezBalikuGenerovanychTrid}.jar"
    basedir="."
    excludes="*.*" >
</jar>
</target>

<target name="ladeni">
    <echo message="Preklad a spusteni aplikace..." />
    <delete>
        <fileset
            dir="${adresarClassSouboru}"
            includes="*.class" />
        </delete>
    <javac
        srcdir="${adresarZdrojovychSouboru}"
        destdir="${adresarClassSouboru}"
        includes="${souborSFunkci_main()_ladeneAplikace}.java"
        debug="on">
<!-- <compilerarg value="-Xlint" /> -->
    </javac>
    <java
        classname=
"${navezBalikuLadeneAplikace}${souborSFunkci_main()_ladeneAplikace}"
        fork="true">
    </java>
</target>

<target name="mazani">
    <delete dir="${adresarGenerovanychSouboru}" />
    <delete dir="${adresarGenerovaneDokumentace}" />
    <delete dir=
"${adresarClassSouboru}/${navezBalikuGenerovanychTrid}" />
    <delete>
        <fileset dir="${adresarClassSouboru}">
            <include name="**/*.class"/>
            <include name="${navezBalikuGenerovanychTrid}.jar"/>
        </fileset>
    </delete>
</target>

<target name="distribuce">
    <!-- vyrobi spustitelny .jar -->
    <delete file="${souborSFunkci_main()_ladeneAplikace}.jar" />
    <jar
        destfile="${souborSFunkci_main()_ladeneAplikace}.jar"

```

```

        basedir="."
        includes="**/*.class">
        <manifest>
            <attribute
                name="Main-Class"
                value="${souborSFunkci_main()_ladeneAplikace}"/>
        </manifest>
    </jar>
</target>
</project>

```

11.2. Generování souborů z XSD

■ provádí se jednorázově na začátku práce pomocí XJC (*XML to Java Compiler*)

- využívá přímo spustitelný soubor – Java\jdk1.6.0\bin\xjc.exe.

♦ vygeneruje .java soubory

- ty je nutné přeložit pomocí standardního `javac`
- je velmi vhodné vygenerovat i dokumentaci pomocí `javadoc`

■ soubor `jidlo.xsd`

```

<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:simpleType name="cisloType">
        <xs:restriction base="xs:nonNegativeInteger">
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="jednotkovaCenaType">
        <xs:restriction base="xs:nonNegativeInteger">
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="navezOvoceType">
        <xs:restriction base="xs:string">
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="navezType">
        <xs:simpleContent>
            <xs:extension base="navezOvoceType">
                <xs:attribute name="jednotkovaCena"
                    type="jednotkovaCenaType"
                    use="required"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>

```

```

<xs:simpleType name="vahaType">
  <xs:restriction base="xs:double">
    <xs:minInclusive value="0" />
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="ovoceType">
  <xs:sequence>
    <xs:element name="nazev" type="nazevType"/>
    <xs:element name="vaha" type="vahaType"/>
  </xs:sequence>
  <xs:attribute name="cislo" type="cisloType"
    use="required"/>
</xs:complexType>

<xs:complexType name="jidloType">
  <xs:sequence>
    <xs:element name="ovoce" type="ovoceType"
      minOccurs="1"
      maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="jidlo" type="jidloType"/>
</xs:schema>

```

■ po spuštění Antu příkazem:

```
>ant generovani
```

se vytvoří následující podadresáře a jeden .jar soubor:

- generovano – .java soubory vzniklé činností XJC
- docsapi – soubory dokumentace vygenerované pomocí javadoc z adresáře generovano
- jidlobalik – .class soubory vzniklé překladem adresáře generovano pomocí javac
- jidlobalik.jar – zabalené .class soubory z adresáře jidlobalik

11.2.1. Princip bindingu

■ po spuštění XJC se vygenerují z jidlo.xsd celkem čtyři .java soubory

- tři soubory odpovídají novým komplexním typům (xs:complexType) z .XSD souboru
 - ♦ JidloType – manipulace s kořenovým elementem <jidlo>
 - ♦ OvoceType – manipulace s elementem <ovoce>
 - ♦ NazevType – manipulace s elementem <nazev>

Poznámka

Pro element <vaha> a atributy cislo a jednotkovaCena, které byly v .XSD souboru deklarovány jako xs:simpleType, nepotřebujeme speciální třídy (viz dále).

- poslední je ObjectFactory.java
 - ♦ tovární třída, která umí pracovat s objekty zmíněných tříd a se třídou JAXBElement (viz dále)

- generované soubory musí být uloženy v balíku, např. jidlobalik

- tyto soubory se jednorázově přeloží pomocí javac

- dále je téměř nezbytné vygenerovat pomocí javadoc dokumentaci

■ průzkumem dokumentace zjistíme:

- každý element nebo atribut má na příslušné úrovni getry a setry. To znamená, že v .java souborech je jejich hodnota buď primitivního datového typu (pro vaha) nebo objekt třídy z Java Core API (viz dále).

```
double getVaha()
void setVaha(double value)
```

- pro xs:simpleType atributy a elementy se používají následující datové objekty

- ♦ reálné číslo (zde element vaha) – primitivní typ double
- ♦ celé číslo (zde atributy cislo a jednotkova cena) – objekt třídy java.math.BigInteger
- ♦ řetězec (zde element nazev) – objekt třídy String

- pokud byl element v .XSD souboru označen s vícenásobným výskytem, např. maxOccurs="unbounded"

```

<xs:complexType name="jidloType">
  <xs:sequence>
    <xs:element name="ovoce" type="ovoceType"
      minOccurs="1"
      maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

```

- ♦ je vygenerován getr poskytující typovaný java.util.List těchto položek (zde ovoce)
 - List je zcela standardní kolekce (viz dříve)

11.3. Čtení XML dokumentu

- velmi jednoduché
- potřebujeme dva balíky

```
1. import javax.xml.bind.*;
```


- vytvoření `Unmarshalleru`

```
2. import jidlobalik.*;
```

- práce s našimi třídami

■ po načtení `Unmarshallerem` se v paměti vytvoří strom objektů

■ překlad a spuštění z příkazové řádky s využitím `.jar` souboru

```
>javac -cp jidlobalik.jar;. VahaJidloJAXB.java
>java -cp jidlobalik.jar;. VahaJidloJAXB
Celkova vaha: 7.05
```

■ překlad a spuštění z příkazové řádky s využitím `.class` souborů

```
>javac VahaJidloJAXB.java
>java VahaJidloJAXB
Celkova vaha: 7.05
```

11.3.1. Výpočet celkové váhy

Program vypíše celkovou váhu nakoupeného ovoce

```
import java.io.*;
import java.util.*;
import javax.xml.bind.*;
import jidlobalik.*;

public class VahaJidloJAXB {
    public static final String BALIK = "jidlobalik";
    public static final String VSTUP = "jidlo.xml";

    public static void main(String[] args) {
        try {
            JAXBContext jc = JAXBContext.newInstance(BALIK);
            Unmarshaller u = jc.createUnmarshaller();
            JAXBElement element = (JAXBElement) u.unmarshal(
                new File(VSTUP));

            JidloType jidlo = (JidloType) element.getValue();
            List<OvoceType> seznamOvoce = jidlo.getOvoce();

            double celkovaVaha = 0;
            for (OvoceType o: seznamOvoce) {
                celkovaVaha += o.getVaha();
            }
            System.out.println("Celkova vaha: " + celkovaVaha);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
}
}
```

■ `JAXBContext` je jakýsi vstupní jednotící bod pro začátek práce

- parametrem jeho metody `newInstance()` může být i několik různých balíků s vygenerovanými třídami

- ♦ názvy jednotlivých balíků by se pak navzájem oddělovaly dvojtečkou

- slouží jako jediný objekt jak pro načítání, tak i pro zápis (*marshalling* – viz dále) XML souboru

■ objekt třídy `Unmarshaller` slouží k převodu XML souborů do stromu objektů Java tříd

Výstraha

Není to infoset DOM! V paměti jsou jen objekty Java tříd jednorázově správně naplněné z načteného XML souboru. JAXB zcela odštiňuje další zpracování od XML formátu.

- načítán nemusí být pouze XML soubor, ale množství jiných zdrojů, viz dokumentaci k `Unmarshaller`

■ `JAXBElement` je obecná třída pro popis libovolného XML elementu

- vytvořena pomocí `Unmarshaller` představuje netypovaný kořenový element XML souboru
- konkrétní (typovaný) kořenový element získáme až použitím metody `getValue()` a přetypováním

■ od této chvíle už pracujeme jen se známými dříve vygenerovanými třídami

- zpracování údajů představuje pouze průchod seznamem

■ potřebujeme-li rozlišit výjimky, lze použít `JAXBException` a její podtřídy

11.4. Čtení dokumentu včetně zpracování atributů

■ neliší se čtení hodnoty atributu a hodnoty elementu

- což bylo v SAX, DOM, StAX

■ výrazně se liší zpracování hodnoty reálného a a celého čísla v XML

- reálné číslo v XML se zpracovává jako `double`
- celé číslo v XML se převádí na objekt třídy `java.math.BigInteger`
 - ♦ z něj lze dostat typ `int` voláním `intValue()` (viz též dále)

11.4.1. Výpočet celkové ceny

Program zpracovává atributy a vypočte celkovou cenu nákupu

```
...
double celkovaCena = 0;
for (OvoceType o: seznamOvoce) {
    BigInteger bi = o.getNazev().getJednotkovaCena();
    int jednotkovaCena = bi.intValue();
    double vaha = o.getVaha();
    celkovaCena += vaha * jednotkovaCena;
}
System.out.println("Celkova cena: " + celkovaCena);
...
```

11.4.2. Všechny objekty v paměti

Program uloží všechny hodnoty a atributy do seznamu námi vytvořené třídy. Provede tak transformaci seznamu jednoho typu třídy (vytvořené JAXB), která je komplikovaná, na druhý seznam naší třídy, která je jednoduchá (nebo předem určená, ...).

Třídy `Ovoce` a `ZpracovaniDatVPameti` jsou zcela stejné jako u SAX a DOM.

```
...
ArrayList<Ovoce> ar = new ArrayList<Ovoce>();

for (OvoceType o: seznamOvoce) {
    int cislo = o.getCislo().intValue();
    int jednotkovaCena = o.getNazev().getJednotkovaCena().intValue();
    String nazev = o.getNazev().getValue();
    double vaha = o.getVaha();
    ar.add(new Ovoce(cislo, nazev, jednotkovaCena, vaha));
}

ZpracovaniDatVPameti.tiskniVse(ar);
System.out.println("Celkova vaha = "
    + ZpracovaniDatVPameti.celkovaVaha(ar));
System.out.println("Celkova cena = "
    + ZpracovaniDatVPameti.celkovaCena(ar));
```

11.5. Změna hodnot a vytvoření nových elementů

■ máme v paměti objekty a každá proměnná objektu má svůj `getr` i `setr`

- hodnoty lze snadno měnit

Výstraha

Pro celé číslo je nutno použít metody třídy `java.math.BigInteger`

■ lze též přidávat elementy

- samozřejmě jen tam, kde to XSD dovolí, ale to je již ošetřeno konstrukcí setrů

■ pro vytváření nových elementů slouží vygenerovaná třída `ObjectFactory`

- poskytuje příslušně pojmenované tovární metody, např.:

```
createOvoceType()
createNazevType()
```

- viz vygenerovaná dokumentace k API

■ pokud je element v seznamu elementů, přidává se na konec kolekce (seznamu) známou metodou `add()`

- přidání na jiné místo musí být řešeno prostředky pro práci s kolekcemi
 - ♦ např. `List` změnit na `ArrayList` a přidávat pomocí indexu (viz dále)

11.6. Zápis do XML dokumentu

■ jednoduchý a podobá se zápisu pomocí třídy `Transformer` (DOM)

■ je třeba vytvořit `Marshaller`

- vytváří se podobně jako `Unmarshaller` pomocí objektu třídy `JAXBContext`

```
Marshaller m = jc.createMarshaller();
```

- ten umí zapisovat do XML souboru

- ♦ též do mnoha jiných výstupů – `Stream`, `Writer`, ...
- ♦ dokáže také transformaci do SAX nebo DOM
- ♦ podrobnosti viz dokumentaci k `Marshaller`

■ podmínky zápisu do XML souboru lze nastavit pomocí `setProperty()`

■ nejčastěji používané jsou:

- `setProperty(Marshaller.JAXB_ENCODING, "windows-1250");`
 - ♦ použité výstupní kódování
 - defaultně UTF-8
- `setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);`
 - ♦ odřádkování (pomocí jen „\n“) za každým elementem
 - ♦ současně odsazuje vnořené elementy o defaultně 4 mezery (nelze měnit)
 - ♦ defaultně je nastaven na `Boolean.FALSE`, tj. neodřádkovává a neodsazuje
- `setProperty(Marshaller.JAXB_NO_NAMESPACE_SCHEMA_LOCATION, "jidlo.xsd");`
 - ♦ přidání odkazu na validační .XSD soubor do kořenového elementu XML souboru

Příklad změny ceny u jablek a přidání dalšího ovoce – broskve. Výsledek uloží do souboru `jidlo-zmena.xml`.

```
import java.io.*;
import java.math.*;
import java.util.*;
import javax.xml.bind.*;
import jidlobalik.*;

public class ZmenaAZapisJAXB {
    public static final String BALIK = "jidlobalik";
    public static final String VSTUP = "jidlo.xml";
    public static final String VYSTUP = "jidlo-zmena.xml";
    public static final String SCHEMA = "jidlo.xsd";

    public static void main(String[] args) {
        try {
            JAXBContext jc = JAXBContext.newInstance(BALIK);
            Unmarshaller u = jc.createUnmarshaller();
            JAXBElement element = (JAXBElement) u.unmarshal(
                new File(VSTUP));

            JidloType jidlo = (JidloType) element.getValue();
            ArrayList<OvoceType> seznamOvoce =
                (ArrayList<OvoceType>) jidlo.getOvoce();

            // zmena ceny jablek
            for (OvoceType o: seznamOvoce) {
                String nazev = o.getNazev().getValue();
                if (nazev.equals("jablka") == true) {
                    o.getNazev().setJednotkovaCena(new BigInteger("43"));
                }
            }

            // vyrobění nového ovoce
            ObjectFactory of = new ObjectFactory();
            OvoceType noveOvoce = of.createOvoceType();
            noveOvoce.setCislo(new BigInteger("5"));
            NazevType n = of.createNazevType();
            n.setJednotkovaCena(new BigInteger("33"));
            n.setValue("broskve");
            noveOvoce.setNazev(n);
            noveOvoce.setVaha(3.1);

            // přidání nového ovoce
            seznamOvoce.add(0, noveOvoce);

            Marshaller m = jc.createMarshaller();
            m.setProperty(Marshaller.JAXB_ENCODING, "windows-1250");
            m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
            // jen pro JDK 1.5
            m.setProperty(Marshaller.JAXB_NO_NAMESPACE_SCHEMA_LOCATION,
                SCHEMA);
            m.marshal(element, new FileOutputStream(VYSTUP));
        }
    }
}
```

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

vygeneruje:

```
<?xml version="1.0" encoding="windows-1250" standalone="yes"?>
<jidlo xsi:noNamespaceSchemaLocation="jidlo.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <ovoce cislo="5">
        <nazev jednotkovaCena="33">broskve</nazev>
        <vaha>3.1</vaha>
    </ovoce>
    <ovoce cislo="1">
        <nazev jednotkovaCena="43">jablka</nazev>
        <vaha>2.5</vaha>
    </ovoce>
    ...
</jidlo>
```

- pro přidání nového ovoce jinam než na konec (zde na první místo), lze např. použít:

```
...
ArrayList<OvoceType> seznamOvoce = (ArrayList<OvoceType>) jidlo.getOvoce();
...
seznamOvoce.add(0, noveOvoce);
```

11.7. Validace

- provádí se pomocí `javax.xml.validation.Validator` stejně jako u DOM

Výstraha

Existuje též *deprecated* třída `javax.xml.bind.Validator`. Ta se nachází v balíku `javax.xml.bind`, který potřebujeme pro další práci s JAXB třídami. Kompilátor hlásí konflikt jmen, proto se používá plně kvalifikované jméno `javax.xml.validation.Validator`.

- nejprve je nutné vytvořit objekt XSD schématu

```
SchemaFactory sf = SchemaFactory.newInstance(
    XMLConstants.W3C_XML_SCHEMA_NS_URI);
Source souborSchema = new StreamSource(new File("jidlo.xsd"));
Schema schemaXSD = sf.newSchema(souborSchema);
```

- pak se vytvoří validátor a případně se mu nastaví (stejně jako u SAX) způsob reakce na chyby
 - zde se pouze připraví hlášení o chybě, které se vypíše v obsluze výjimky (viz dále)

```

javax.xml.validation.Validator validator = schemaXSD.newValidator();
validator.setErrorHandler(new ChybyZjisteneValidatorem());
...

class ChybyZjisteneValidatorem implements ErrorHandler {
    public void generuj(String kategorie, SAXException e)
        throws SAXException {
        throw new SAXException(kategorie + ": " + e.toString());
    }

    public void warning(SAXParseException e) throws SAXException {
        generuj("Varovani", e);
    }

    public void error(SAXParseException e) throws SAXException {
        generuj("Chyba", e);
    }

    public void fatalError(SAXParseException e) throws SAXException {
        generuj("Fatalni chyba", e);
    }
}

```

■ validace je pak možná minimálně ve dvou časových bodech

- před zpracováním souboru Unmarshallem

```

Source vstupniSoubor = new StreamSource(new File("jidlo.xml"));
validator.validate(vstupniSoubor);

```

- ♦ to prakticky znamená, že validace je zcela nezávislá na JAXB

- po načtení do paměti

```

JAXBSource pamet = new JAXBSource(jc, element);
validator.validate(pamet);

```

Poznámka

Kromě toho lze stejným postupem validovat infoset DOM v paměti atp.

11.7.1. Ukázka dvojí validace

```

import java.io.*;
import java.math.*;
import java.util.*;
import javax.xml.bind.*;
import javax.xml.bind.util.JAXBSource;
import javax.xml.validation.SchemaFactory;
import javax.xml.validation.Schema;
import javax.xml.validation.Validator;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;

```

```

import javax.xml.XMLConstants;
import org.xml.sax.*;
import jidlobalik.*;

```

```

public class ValidaceJAXB {
    public static final String BALIK = "jidlobalik";
    public static final String VSTUP = "jidlo.xml";
    public static final String SCHEMA = "jidlo.xsd";

    public static void main(String[] args) {
        try {
            SchemaFactory sf = SchemaFactory.newInstance(
                XMLConstants.W3C_XML_SCHEMA_NS_URI);
            Source souborSchema = new StreamSource(new File(SCHEMA));
            Schema schemaXSD = sf.newSchema(souborSchema);

            javax.xml.validation.Validator validator = schemaXSD.newValidator();
            validator.setErrorHandler(new ChybyZjisteneValidatorem());

            // validace vstupního souboru
            Source vstupniSoubor = new StreamSource(new File(VSTUP));
            validator.validate(vstupniSoubor);

            JAXBContext jc = JAXBContext.newInstance(BALIK);
            Unmarshaller u = jc.createUnmarshaller();
            JAXBElement element = (JAXBElement) u.unmarshal(
                new File(VSTUP));

            JidloType jidlo = (JidloType) element.getValue();
            List<OvoceType> seznamOvoce = jidlo.getOvoce();

            // chybná změna vahy grapefruitu
            for (OvoceType o: seznamOvoce) {
                String nazev = o.getNazev().getValue();
                if (nazev.equals("grapefruity") == true) {
                    o.setVaha(-1.0);
                }
            }

            // validace objektu v paměti
            JAXBSource pamet = new JAXBSource(jc, element);
            validator.validate(pamet);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}

```

- pokud je chyba v souboru jidlo.xml (záporná váha -2.5) vypíše:

```

Chyba: org.xml.sax.SAXParseException:
cvc-minInclusive-valid:

```

```
Value '-2.5' is not facet-valid with respect to minInclusive '0.0E1'
for type 'vahaType'.
```

- pokud je chybně opraven údaj v paměti (záporná váha -1.0) vypíše:

```
Fatalni chyba: javax.xml.bind.MarshalException
- with linked exception:
[org.xml.sax.SAXException: Chyba: org.xml.sax.SAXParseException:
cvc -minInclusive-valid:
Value '-1.0' is not facet-valid with respect to minInclusive '0.0E1'
for type 'vahaType'.]
```

11.8. Příprava kompletně nového dokumentu

- je třeba připravit kořenový element `jidlo`

- opět pomocí tovární metody třídy `ObjectFactory`

- dále je stejný postup jako při úpravě dokumentu

- před zápisem je nutno připravit objekt třídy `JAXBElement`

- díky schopnostem `Marshalleru` (odřádkování, odsazování apod.), je kód výrazně jednodušší než při použití `StAX`

```
import java.io.*;
import java.math.*;
import java.util.*;
import javax.xml.bind.*;
import jidlobalik.*;

public class JidloWriteJAXB {
    public static final String VYSTUP = "jidlo-generovano.xml";
    public static final int POCET = 3;

    public static void main(String[] args) {
        try {
            Random r = new Random();
            JAXBContext jc = JAXBContext.newInstance("jidlobalik");

            ObjectFactory of = new ObjectFactory();

            // vyrobene noveho jidla - korenovy element
            JidloType jidlo = (JidloType) of.createJidloType();
            List<OvoceType> seznamOvoce = jidlo.getOvoce();

            // vyrobene ovoci
            for (int i = 1; i <= POCET; i++) {
                OvoceType o = of.createOvoceType();
                o.setCislo(new BigInteger(String.valueOf(i)));
                NazevType n = of.createNazevType();
                int cena = r.nextInt(40) + 10;
```

```
n.setJednotkovaCena(new BigInteger(String.valueOf(cena)));
String nazev = "ovoce " + i + " & <";
n.setValue(nazev);
o.setNazev(n);
double vaha = r.nextDouble() * 10.0;
String oriznute = String.valueOf(vaha).substring(0, 3);
o.setVaha(Double.parseDouble(oriznute));
seznamOvoce.add(o);
}
```

```
Marshaller m = jc.createMarshaller();
m.setProperty(Marshaller.JAXB_ENCODING, "windows-1250");
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
m.setProperty(Marshaller.JAXB_NO_NAMESPACE_SCHEMA_LOCATION,
    "jidlo.xsd");
```

```
// element pro Marshaller
JAXBElement element = of.createJidlo(jidlo);
m.marshal(element, new FileOutputStream(VYSTUP));
}
catch (Exception e) {
    e.printStackTrace();
}
}
```

vygeneruje např.:

```
<?xml version="1.0" encoding="windows-1250" standalone="yes"?>
<jidlo xsi:noNamespaceSchemaLocation="jidlo.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <ovoce cislo="1">
        <nazev jednotkovaCena="45">ovoce 1 & < </nazev>
        <vaha>0.9</vaha>
    </ovoce>
    <ovoce cislo="2">
        <nazev jednotkovaCena="42">ovoce 2 & < </nazev>
        <vaha>9.7</vaha>
    </ovoce>
    <ovoce cislo="3">
        <nazev jednotkovaCena="35">ovoce 3 & < </nazev>
        <vaha>7.8</vaha>
    </ovoce>
</jidlo>
```

Kapitola 12. Java a národní prostředí

- problematika je značně složitější než jen vstup a výstup akcentovaných znaků:
 - vstup a výstup akcentovaných znaků ale i dalších speciálních znaků
 - použití akcentovaných znaků ve zdrojových kódech (I/O, dokumentační komentáře, ...)
 - převod velkých písmen na malá a naopak
 - řazení řetězců podle pravidel abecedního řazení národního jazyka
 - formátování čísel
 - práce s datem a časem, měnou apod.
- používané zkratky:
 - `i18n` – `internationalization`
 - ♦ psaní programu tak, aby se dal snadno přizpůsobit libovolnému národnímu prostředí
 - `l10n` – `localization`
 - ♦ převedení programu do dané jazykové mutace
- národní zvláštnosti nemají být řešeny „natvrdo“ ve zdrojovém kódu
 - popsat v konfiguračních souborech
 - využívat služeb operačního systému
- řešení problémů typu řazení, formátování čísel, datumů apod.
 - využívat specializované třídy/metody z Java Core API respektující národní zvyklosti
- podrobný rozbor viz Java Tutorial

12.1. Kódování

Poznámka

Místo termínu „text s akcentovanými znaky“ bude dále používán termín „čeština“.

- problém zobrazení češtiny vnitřně v Javě v podstatě neexistuje
 - vnitřně používá Unicode (dvoubajtový `char`)
- Java je ale provozována téměř výhradně na operačních systémech, které mají znaky téměř výhradně osmibitové
 - fonty, pomocí nichž se texty zobrazují
 - textové soubory, ve kterých jsou texty uloženy

- problém – přizpůsobit šestnáctibitové znaky Javy osmibitovým znakům jejího okolí

12.1.1. Podporovaná kódování

- podrobný rozbor viz přednáška z PPA1
- Java přímo podporuje množství kódování, která mají v Core API zavedené zkratky
 - podle pokusů se někdy rozlišují a někdy nerozlišují velká a malá písmena zkratk
 - použijete-li nevyhovující zkratku kódování, bude vyhozena výjimka `UnsupportedEncodingException`
- kanonická jména čtyř nejběžnějších kódování:
 - `ISO-8859-2` – kódování dle mezinárodní normy (často v prostředí Unixů)
 - ♦ známé též pod názvem `ISO LATIN2`
 - ♦ v Javě se občas vyskytne jako `ISO8859_2`
 - `windows-1250` – proprietární kódování firmy Microsoft
 - ♦ od `ISO-8859-2` se liší jen v několika málo znacích (z akcentovaných znaků češtiny jsou to jen š, Š, ť, ě, ě, ž, Ž)
 - ♦ implicitní pro JDK pod Windows !
 - ♦ známé též pod názvem `CP1250`
 - ♦ v Javě se občas vyskytne jako `Cp1250`
 - `IBM852` – kódování používané v konzolovém okénku Windows!
 - ♦ občas je označováno jako `DOS Latin2` nebo `PC Latin 2`
 - ♦ nezaměňovat s `ISO LATIN2`, které je zcela odlišné
 - ♦ v Javě se občas vyskytne jako `Cp852`
 - `UTF-8` – jedno z nejrozšířenějších kódování pro Unicode
 - ♦ v Javě se občas vyskytne jako `UTF8`

12.2. Čeština v programu

- jména identifikátorů – nejjednodušší problém – pouze zajistit, aby byl program správně přeložen
 - JDK pod Windows má implicitní kódování `windows-1250` – překlad bez jakýchkoli problémů
- texty výpisů – složitější
 - překlad bez problémů
 - výpis na konzoli s problémy – špatné akcenty – konzole je v `IBM852` – řešení viz dále

```
public class CestinaIdentifikatory {
    public static void main(String[] args) {
        int pěknýČeskýČítač = 1;
        System.out.println("pěknýČeskýČítač = "
                           + pěknýČeskýČítač);
    }
}
```

přeložení a spuštění:

```
D:\zzz>javac CestinaIdentifikatory.java
D:\zzz>java CestinaIdentifikatory
pěknýČeskýČítač =1
D:\zzz>type CestinaIdentifikatory.java
public class CestinaIdentifikatory {
    public static void main(String[] args) {
        int pěknýČeskýČítač = 1;
        System.out.println("pěknýČeskýČítač = "
                           + pěknýČeskýČítač);
    }
}
```

- v .class souboru je použito UTF-8

- .class soubory jsou plně přenositelné na jiné platformy

12.2.1. Zdrojový kód není v implicitním kódování

- je-li zdrojový soubor v jiném než implicitním kódování, nastává již problém s překladem

- tento soubor můžeme získat přenosem z jiné platformy (typicky z Linuxu) nebo (nevhodným) nastavením editoru
- pro soubor UTF8.java, který má jinak zcela stejný obsah jako předchozí soubor:

```
public class UTF8 {
    public static void main(String[] args) {
        int pěknýČeskýČítač = 1;
        System.out.println("pěknýČeskýČítač = "
                           + pěknýČeskýČítač);
    }
}
```

po pokusu o překlad dostaneme:

```
D:\zzz>javac UTF8.java
UTF8.java:3: illegal character: \8250
    int p-řkn|ž-řesk|ž-řsta-ž = 1;
    ^
UTF8.java:3: illegal character: \733
    int p-řkn|ž-řesk|ž-řsta-ž = 1;
    ^
UTF8.java:3: not a statement
    int p-řkn|ž-řesk|ž-řsta-ž = 1;
    ^
```

- možným – ale neelegantním – řešením je pomocí externího programu pro změnu kódování převést soubor do implicitního kódování

12.2.1.1. Řešení prostředky JDK – encoding

- předchozí případ lze elegantně řešit přepínačem encoding programu java.exe

- rozpoznává všechna dříve uvedená kódování

- překlad a spuštění souboru UTF8.java (řešení špatného kódování konzole viz dále)

```
D:\zzz>javac -encoding UTF-8 UTF8.java
D:\zzz>java UTF8
pěknýČeskýČítač =1
D:\zzz>
```

- tento způsob je výhodný v případě, že zdrojový soubor ladíme, tj. editujeme

- všechny nápisy v editoru zdrojového kódu vidíme v češtině

Výstraha

Záludnou chybou je uložení zdrojového souboru v UTF-8 s BOM (*Byte Order Mark*)

- to často provede „editor o své vlastní vůli“

- s BOM javac.exe nepočítá a překlad skončí chybou

- ta je o to horší, že chybu při neznalosti principu BOM nelze odhalit, protože v editoru se BOM nezobrazuje

```
D:\zzz>javac -encoding UTF-8 UTF8BOM.java
UTF8BOM.java:1: illegal character: \65279
?public class UTF8BOM {
^
1 error
D:\zzz>
```

- BOM je nutné se zbavit, což lze např. pomocí editoru SciTe

12.2.1.2. Řešení prostředky JDK – native2ascii

- zdrojové soubory v Javě nemají možnost (narozdíl od souborů XML) defiovat použité kódování

- předáváme-li zdrojový soubor v nějakém kódování, záleží na zkušenostech příjemce, aby použité kódování správně rozpoznal a pak použil v přepínači encoding

- správné rozpoznání není triviální záležitost

- elegantním řešením problému je skutečnost, že každý akcentovaný znak může být ve zdrojovém souboru zapsán pomocí sekvence \uXXXX

- xxxx je kódový bod znaku v Unicode, např. znak 'ě' – '\u011b'

- nahradíme-li takto ve zdrojovém souboru všechny akcentované znaky, dostaneme ASCII soubor

- odstraníme trvale závislost na různém kódování

- soubor je plně přenositelný a bez problémů přeložitelný

- ovšem

- převod znaků se velmi obtížně se provádí "ručně"

- na převod použijeme program native2ascii.exe, který je součástí JDK

- ♦ jeho parametr je encoding má zcela stejný význam jako u `javac.exe`
- ♦ rozdíl od předchozího způsobu je v tom, že `native2ascii.exe` používá autor zdrojového kódu, který by měl znát použité kódování
- ♦ `native2ascii.exe` vypisuje svůj výstup implicitně na konzoli
- ♦ zapisujeme-li výstup do souboru, je vhodné (bezpečnější) použít pomocný soubor (zde `tmp.java`) a ten pak přejmenovat na původní soubor

```
D:\zzz>type IBM852.java
public class IBM852 {
    public static void main(String[] args) {
        int pěknýČeskýČítač = 1;
        System.out.println("pěknýČeskýČítač ="
            + pěknýČeskýČítač);
    }
}

D:\zzz>native2ascii -encoding IBM852 IBM852.java
public class IBM852 {
    public static void main(String[] args) {
        int p\u011bkn\u00fd\u010cesk\u00fd\u010c\u00eda\u010d = 1;
        System.out.println("p\u011bkn\u00fd\u010cesk\u00fd\u010c\u00eda\u010d ="
            + p\u011bkn\u00fd\u010cesk\u00fd\u010c\u00eda\u010d);
    }
}

D:\zzz>native2ascii -encoding IBM852 IBM852.java tmp.java
D:\zzz>del IBM852.java
D:\zzz>ren tmp.java IBM852.java
D:\zzz>javac IBM852.java
D:\zzz>java IBM852
p\u011bkn\u00fd\u010cesk\u00fd\u010c\u00eda\u010d=1
D:\zzz>
```

- tento způsob provádíme na samém konci práce, když je zdrojový soubor odladěný a my jej archivujeme, či předáváme jinam

- takto upravený zdrojový soubor se mimořádně špatně edituje

- potřebujeme-li (rozsáhlejší editaci), použijeme `native2ascii.exe` s přepínačem `reverse`
 - ♦ provede převod sekvencí '`\uXXXX`' na jejich akcentovanou podobu ve zvoleném kódování
 - ♦ v příkladu vznikající soubor `IBM852.java` přepíše původní `IBM852.java` (nepoužil se pomocný soubor)

```
D:\zzz>native2ascii -reverse -encoding IBM852 IBM852.java IBM852.java
D:\zzz>type IBM852.java
public class IBM852 {
    public static void main(String[] args) {
        int pěknýČeskýČítač = 1;
        System.out.println("pěknýČeskýČítač ="
            + pěknýČeskýČítač);
    }
}

D:\zzz>_
```

12.3. České výpisy na konzoli

Poznámka

Spouštíme-li programy ze SciTe nebo Eclipse, s tímto problémem se nesetkáme, protože ty již mají výstup v GUI Windows, tj. v implicitním kódování `windows-1250`.

- konzolový výstup mají většinou jen zkušební nebo cvičné programy
 - stejný princip ale použijeme i při práci se soubory
- základem veškerých změn kódování jsou třídy `InputStreamReader` a `OutputStreamWriter`

- představují spojení mezi 8bitovými znaky operačního systému a 16bitovými znaky Javy
- při čtení a při zápisu probíhá překódování znaků podle přednastaveného dekódování

- ♦ ve Windows pro vstup i výstup přednastaveno kódování `windows-1250`

- pro změnu kódování použijeme pro obě třídy konstruktor

- ♦ první parametr je instance `InputStream` nebo `OutputStream` (tedy binární zpracování souboru!)

- ♦ druhý parametr je řetězec určující nové kódování

- nově nastavený `OutputStreamWriter` se pak použije např. jako parametr `PrintWriter`

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

```
public class NaKonzoli {
    public static void main(String[] args) throws Exception {
        OutputStreamWriter oswDef =
            new OutputStreamWriter(System.out);
        System.out.println("Implicitní kodování konzole: "
            + oswDef.getEncoding());

        /* IBM852 je výstupní kódování češtiny v DOSovém okénku */
        OutputStreamWriter osw =
            new OutputStreamWriter(System.out, "IBM852");
        System.out.println("Nastavene kodování konzole: "
            + osw.getEncoding());
    }
}
```

```
PrintWriter p = new PrintWriter(osw);
p.print("Příšerně žlutoučký kůň úpěl dábelské ódy.\n");
p.print("áčďěšíňóřšťúůýž\n");
p.print("PŘÍŠERNĚ ŽLUŤOUČKÝ KŮŇ ÚPĚL ĎÁBELSKÉ ÓDY.\n");
p.print("ÁČĎĚŠÍŇÓŘŠŤÚŮÝŽ\n");
p.flush();
```

příklad a spuštění

```
D:\zzz>javac NaKonzoli.java
D:\zzz>java NaKonzoli
Implicitní kodování konzole: Cp1250
Nastavene kodování konzole: Cp852
Příšerně žlutoučký kůň úpěl dábelské ódy.
áčďěšíňóřšťúůýž
PŘÍŠERNĚ ŽLUŤOUČKÝ KŮŇ ÚPĚL ĎÁBELSKÉ ÓDY.
ÁČĎĚŠÍŇÓŘŠŤÚŮÝŽ
D:\zzz>
```

- používají se zde „stará“ označení kódování (`Cp1250`, `Cp852`)

12.3.1. Vstup češtiny z konzole

■ princip je velmi podobný výstupu na konzoli

■ zde jsou ukázány dvě možnosti vstupu

- pomocí `InputStreamReader` – tento způsob se pak použije pro vstup ze souborů
- pomocí `Scanner` – běžný způsob načítání z klávesnice

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

public class IOKonzole {
    public static void main(String[] args) throws Exception {
        OutputStreamWriter osw =
            new OutputStreamWriter(System.out, "IBM852");
        PrintWriter p = new PrintWriter(osw);
        InputStreamReader isr =
            new InputStreamReader(System.in, "IBM852");
        BufferedReader ib = new BufferedReader(isr);
        Scanner sc = new Scanner(System.in, "IBM852");

        p.print("Zadej akcentované znaky: ");
        p.flush();
        String s = sc.nextLine();
        p.print("Zadal jsi: " + s + "\n");
        p.flush();

        p.print("Zadej další akcentované znaky: ");
        p.flush();
        s = ib.readLine();
        p.print("Zadal jsi: " + s + "\n");
        p.flush();
    }
}
```

```
D:\zzz>javac IOKonzole.java
D:\zzz>java IOKonzole
Zadej akcentované znaky: Dáša
Zadal jsi: Dáša
Zadej další akcentované znaky: Dáša
Zadal jsi: Dáša
D:\zzz>
```

12.4. Čeština v souborech

■ situace prakticky stejná, jako s češtinou z konzole

■ třída `Reader` čte bajty, které konvertuje na znaky podle implicitního kódování

- totéž platí pro třídu `Writer` – zapisuje 16bitové znaky, ale v souboru se objeví jejich 8bitová náhrada

■ znamená to, že `Reader` a `Writer` lze správně použít jen pro implicitní kódování

- to lze zjistit ze systémové vlastnosti `file.encoding` příkazem `System.getProperty("file.encoding")`

■ zpracování pomocí jiného kódování je stejné jako u konzole – třídy `InputStreamReader` a `OutputStreamWriter`

- nikdy se nesnažíme o změnu `file.encoding` !!!

Ukázka, jak načíst soubor v jiném kódování než `windows-1250`. Zkratka požadovaného kódování se zadá z příkazové řádky a soubor se kontrolně opiše na obrazovku způsobem známým z předchozí části. Zadáme-li z příkazové řádky UTF-8, vypíše se správný text. Zadáme-li např. `windows-1250`, dostaneme částečně nesmyslný výpis.

```
import java.io.*;

public class SouborCteni {
    public static void main(String[] args) throws Exception {
        OutputStreamWriter osw =
            new OutputStreamWriter(System.out, "IBM852");
        PrintWriter p = new PrintWriter(osw);
        String kodovani = args[0];

        FileInputStream fis =
            new FileInputStream("vstup.utf8");
        InputStreamReader isr =
            new InputStreamReader(fis, kodovani);
        BufferedReader br = new BufferedReader(isr);

        String radka;
        while ((radka = br.readLine()) != null) {
            p.println(radka);
            p.flush();
        }
        fis.close();
    }
}
```

```
D:\zzz>javac SouborCteni.java
D:\zzz>java SouborCteni UTF-8
?public class UTF8 {
    public static void main(String[] args) {
        int pěknýČeskýČítač = 1;
        System.out.println("pěknýČeskýČítač ="
            + pěknýČeskýČítač);
    }
}
D:\zzz>java SouborCteni windows-1250
?public class UTF8 {
    public static void main(String[] args) {
        int pš?knš?ššeskš?šš-tašš = 1;
        System.out.println("pš?knš?ššeskš?šš-tašš ="
            + pš?knš?ššeskš?šš-tašš);
    }
}
D:\zzz>
```

Totéž lze použít i pro výstupní soubor. Zde máme možnost vytvořit více souborů se stejným obsahem, ale jiným kódováním. Typ kódování se zadá z příkazové řádky a je to současně i přípona souboru se jménem `vystup`. Metoda `toUpperCase()` třídy `String` převádí zcela správně akcentované znaky, bez

ohledu na jejich kódování. Je to tím, že řetězec kun (nebo akcenty) je v Javě 16bitový Unicode – nemá s výstupním kódováním nic společného.

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

```
public class SouborZapis {
    public static void main(String[] args) throws Exception {
        String kodovani = args[0];
        String jmenoSouboru = "vystup." + kodovani;

        FileOutputStream fos = new FileOutputStream(jmenoSouboru);
        OutputStreamWriter osw =
            new OutputStreamWriter(fos, kodovani);
        PrintWriter p = new PrintWriter(osw);

        String kun = "Příšerně žlutoučký kůň úpěl ďábelské ódy";
        String akcenty = "áčďěěíňóřšťůůýž";
        p.println(kun);
        p.println(akcenty);
        p.println(kun.toUpperCase());
        p.println(akcenty.toUpperCase());

        p.close();
    }
}
```

```
D:\zzz>javac SouborZapis.java
D:\zzz>java SouborZapis IBM852
D:\zzz>type vystup.IBM852
Příšerně žlutoučký kůň úpěl ďábelské ódy
áčďěěíňóřšťůůýž
PŘÍŠERNĚ ŽLUTOUČKÝ KŮŇ ÚPEĽ ĎÁBELSKÉ ÓDY
ÁČĎĚĚÍŇÓŘŠŤŮŮÝŽ
D:\zzz>
```

12.5. Převody mezi různými kódováními uvnitř programu

- v rámci programu potřebujeme získat z řetězce znaky v daném kódování
 - metoda `byte[] getBytes(String kodovani)` třídy `String`
 - pro řetězec vrátí pole bajtů ve zvoleném kódování
- opačný postup je, máme-li pole bajtů v určitém kódování a potřebujeme z něj získat řetězec
 - konstruktor `String(byte[] bytes, String kodovani)`

Jak lze snadno zjistit kódy jednotlivých znaků pro různá kódování.

```
public class PrevodStringu {
    public static String byteNaHexa(byte[] b) {
        String s = " ";
        for (int i = 0; i < b.length; i++) {
```

```
        int j = (b[i] < 0) ? 256 + b[i] : b[i];
        s = s + s.format("%02x ", j);
    }
    return s;
}

public static void main(String[] args) throws Exception {
    String test = "áčďěěíňóřšťůůýž";
    String[] kodovani = {"windows-1250", "ISO-8859-2",
                        "IBM852", "UTF-8",
                        "UTF-16BE", "UTF-16LE",
                        "UTF-16"};
    for (int i = 0; i < kodovani.length; i++) {
        byte[] b = test.getBytes(kodovani[i].trim());
        System.out.println(kodovani[i] + ":" + byteNaHexa(b));
    }
}
```

```
windows-1250: e1 e8 ef e9 ec ed f2 f3 f8 9a 9d fa f9 fd 9e
ISO-8859-2   : e1 e8 ef e9 ec ed f2 f3 f8 b9 bb fa f9 fd be
IBM852       : a0 9f d4 82 d8 a1 e5 a2 fd e7 9c a3 85 ec a7
UTF-8        : c3 a1 c4 8d c4 8f c3 a9 c4 9b c3 ad c5 88 c3 b3 c5 99 c5 a1 c5 ▶
a5 c3 ba c5 af c3 bd c5 be
UTF-16BE     : 00 e1 01 0d 01 0f 00 e9 01 1b 00 ed 01 48 00 f3 01 59 01 61 01 ▶
65 00 fa 01 6f 00 fd 01 7e
UTF-16LE     : e1 00 0d 01 0f 01 e9 00 1b 01 ed 00 48 01 f3 00 59 01 61 01 65 ▶
01 fa 00 6f 01 fd 00 7e 01
UTF-16       : fe ff 00 e1 01 0d 01 0f 00 e9 01 1b 00 ed 01 48 00 f3 01 59 01 ▶
61 01 65 00 fa 01 6f 00 fd 01 7e
```

12.6. Třída Locale

- nejdůležitější třída pro práci s národním prostředím
 - je z balíku `java.util`
- ovlivňuje činnost mnoha dalších tříd
 - lze vytvořit její instanci, která je pak dalšími třídami používána
 - ♦ problém změny národního prostředí se tak redukuje na změnu pomocí jedné řádky kódu
- uschovává informace o jazyku a o oblasti (zemi, státu)
- statická metoda `Locale.getDefault()` vrací objekt, který popisuje aktuální nastavení pro právě platnou instanci JVM
 - nastavení se shoduje s nastavením, které lze změnit v operačním systému
 - pro Windows XP je to ve Start/Control Panel/Regional Settings
- dále bude používán pojem *lokalita* = kombinace jazyka a země

Výpis hodnot pro přednastavenou lokalitu.

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

```
public class MojeLocale {
    public static void main(String[] args) throws Exception {
        Locale d = Locale.getDefault();
        System.out.println("Země : " + d.getCountry());
        System.out.println("Jazyk: " + d.getLanguage());
        System.out.println("Země : " + d.getDisplayCountry());
        System.out.println("Jazyk: " + d.getDisplayLanguage());
        System.out.println("ISO země : " + d.getISO3Country());
        System.out.println("ISO jazyk: " + d.getISO3Language());
    }
}
```

vypiše pro Control Panel/Regional Setting/Czech:

```
Země : CZ
Jazyk: cs
Země : Česká republika
Jazyk: čeština
ISO země : CZE
ISO jazyk: ces
```

■ metoda `void setDefault(Locale newLocale)`

- dokáže pro konkrétní instanci JVM (ne pro operační systém!) změnit lokalitu

■ instanci třídy `Locale` dostaneme pomocí `Locale(String jazyk, String zeme)`

- je možné pracovat s více lokalitami najednou

■ příklady použití konstruktoru a zkratk jazyků a zemí

```
czLocale = new Locale("cs", "CZ");
skLocale = new Locale("sk", "SK");
usLocale = new Locale("en", "US");
gbLocale = new Locale("en", "GB");
```

■ objekty `Locale` jsou pouze identifikátory, které samy nejsou schopny žádné činnosti

- poskytují se jiným třídám jako parametry jejich "výkonných" metod
 - ♦ tyto metody ale nemusejí podporovat všechny možné kombinace jazyků a zemí
 - ♦ podporované lokality lze u každé třídy, která je s nimi schopna pracovat, zjistit voláním statické metody `getAvailableLocales()`

– vrátí pole všech podporovaných `Locale`, např.:

```
Locale[] l = DateFormat.getAvailableLocales();
```

♦ podle pokusů patří lokality "cs_CZ" a "sk_SK" vždy mezi podporované

12.7. Tisk dle národních zvyklostí

■ před JDK 1.5 málo přehledné použití několika různých tříd

- `NumberFormat` pro čísla, měny

- `DecimalFormat` pro speciální formát celých a reálných čísel

- `DateFormat` a `SimpleDateFormat` pro data a časy

■ od JDK 1.5 lze použít metodu `format()` ze tříd `PrintStream` a `String`

- nedokáže ale předchozí třídy plně nahradit

12.7.1. Základní principy `format()`

Poznámka

Používáme jen tehdy, když nám nestačí možnosti `System.out.println()`.

■ jasná inspirace možnostmi jazyka C

■ možnost volit šířku výpisu, zarovnávání doprava či doleva, nevýznamové mezery a množství dalších věcí

- vyčerpávající informace i s příklady viz `java.util.Formatter`

■ základní statická metoda `String.format(parametr)`

- poskytnete řetězec, který se dá dále zpracovat (poslat na výstup – *stream*, soubor, GUI, ...)

■ stejná metoda je ve třídě `PrintStream`, tj. `System.out.format()`

■ vyřešeno přenositelně odřádkování pomocí `%n`, např.:

```
String.format("nova radka%n");
```

■ základní princip je formátovací řetězec jako první parametr a pak jednotlivé položky jako další parametry

■ základní pravidla

- ve formátovacím řetězci jsou za znakem `%` formátovací znaky
- kolik je znaků `%`, tolik musí být dalších parametrů (s výjimkou `%n`)
- `String.format("i = %d, j = %d%n", i, j);`

12.7.1.1. Výpis celého čísla v desítkové soustavě

■ používá se `%d`, např. pro `int i = -1234;`

```
String.format("i = %d%n", i); // i = -1234
```

- počet míst lze stanovit, pak se doplňují mezery zleva, tj. zarovnání doprava, např.:

```
String.format("i = %7d%n", i); // i = -1234
```

- počet míst lze stanovit a zarovnat doleva (zbylé místo se doplní mezerami), např.:

```
String.format("i = %-7dahoj%n", i); // i = -1234 ahoy
```

- lze vynutit výpis i + znaménka, např. pro `int i = 1234`;

```
String.format("i = %+7d%n", i); // i = +1234
```

- vynutí se výpis nevýznamových nul, např.:

```
String.format("i = %07d%n", i); // i = -001234
```

12.7.1.2. Výpis celého čísla v jiných soustavách

- osmičková soustava, např. pro `int j = 30`;

```
String.format("j = %o%n", j); // j = 36
```

- šestnáctková soustava, např. pro `int j = 30`;

```
String.format("j = %X%n", j); // j = 1E
```

- počet míst lze určit, např.:

```
String.format("j = %3X%n", j); // j = 1E
```

- lze vynutit nevýznamové nuly, např. pro: `int j = 10`;

```
String.format("j = %02X%n", j); // j = 0A
```

12.7.1.3. Výpis znaku

- používá se `%c`, např. pro `char c = 'a'`;

```
String.format("c = %c%n", c); // c = a
```

- lze použít přetypování a lze vypsát více proměnných najednou

```
String.format("Znak %c ma ASCII hodnotu: %d%n", c, (int) c);  
  
// Znak a ma ASCII hodnotu: 97
```

12.7.1.4. Výpis reálného čísla

- výpis jako běžné reálné číslo `%f`, např. pro `double d = 1234.567`;

```
String.format("d = %f%n", d); // d = 1234,567000
```

Poznámka

Desetinný oddělovač je závislý na lokalitě – pro ČR je to čárka, nikoliv tečka.

- výpis ve vědeckotechnické notaci `%g`, např. pro `double d = 1234.567`;

```
String.format("d = %g%n", d); // d = 1.234567e+03
```

Poznámka

Desetinný oddělovač je tečka.

- lze nastavit počet míst celkem (10) a počet míst za desetinným oddělovačem (1), číslo bude zaokrouhleno

```
String.format("d = %10.1f%n", d); // d = 1234,6
```

- lze použít zarovnání doleva, výpis nevýznamových nul, oddělovač řádů apod. stejně, jako u celého čísla

12.7.1.5. Výpis řetězce

- používá se `%s`, např. pro `String s = "Ahoj lidi"`;

```
String.format("s = %s%n", s); // s = Ahoj lidi
```

- řetězec lze vypsát velkými písmeny

```
String.format("s = %S%n", s); // s = AHOJ LIDI
```

- lze stanovit šířku výpisu, výpis bude zarovnán doprava

```
String.format("s = |%11s|%n", s); // s = | Ahoj lidi |
```

- výpis lze zarovnat i doleva

```
String.format("s = |%-11s|%n", s); // s = |Ahoj lidi |
```

12.7.2. Formátování dle lokality

- `format()` využívá přednastavenou lokalitu

- vhodnější je však danou lokalitu zadat pro každé použití

- `String.format(lokalita, formátovací_řetězec, tisknuté_parametry)`

12.7.2.1. Tisk a načtení čísel

- u čísel se v národních zvyklostech mění desetinný oddělovač a oddělovač řádů
 - jejich používání se ve `format()` vynutí pomocí znaku „,“ (čárka) bezprostředně za znakem „%“

Ukázka používaných oddělovačů v různých lokalitách. Průzkum typu oddělovače řádů v české lokalitě. Formátovací řetězec `"%,.2f \t%s%n"` znamená:

- `,` – tiskni oddělovač řádů
- `.2` – tiskni na dvě desetinná místa se zaokrouhlením
- `f` – tiskni reálné číslo
- `\t` – tiskni mezeru a tabulátor
- `%s` – tiskni řetězec (zde název lokality)
- `%n` – tiskni odřádkovací znaky dle konvencí platformy

Výpis znaku „ě“ (kódový bod `\u011b`) v dalším tisku je proto, aby byl zřejmý rozsah a uložení bajtů.

```
import java.util.*;

public class CiskaLocale {
    public static void main(String[] args) throws Exception {
        double d = 1234567.894;
        Locale[] lo = { new Locale("cs", "CZ"),
                        new Locale("sk", "SK"),
                        new Locale("en", "US"),
                        new Locale("de", "DE") };
        for (int i = 0; i < lo.length; i++) {
            System.out.format(lo[i], "%,.2f \t%s%n", d,
                              lo[i].getDisplayName());
        }

        double d1 = 1234.5;
        String s = String.format(lo[0], "ě %, .2f", d1);
        byte[] b = s.getBytes("UTF-16BE");

        for (int i = 0; i < b.length; i++) {
            int j = (b[i] < 0) ? 256 + b[i] : b[i];
            System.out.format("%02x ", j);
        }
    }
}
```

vypiše:

```
1 234 567,89 čeština (Česká republika)
1 234 567,89 Slovak (Slovakia)
1,234,567.89 English (United States)
```

```
1.234.567,89 German (Germany)
01 1b 00 20 00 31 00 a0 00 32 00 33 00 34 00 2c 00 35 00 30
```

- v české typografii je oddělovač řádů pevná mezera šířky čtvrt čtverčíku („čtverčík“ je typografický termín)

- Java tento znak nepoužívá (měl by kódový bod `\u020F` [*narrow no-break space*])

- místo něj používá pevnou mezeru (kódový bod `\u00A0` [*no-break space*])

- ◆ není-li tento znak k dispozici v používaném fontu, zobrazí se nesmyslný znak, např. při výpisu na konzoli: `1á234á567,89`

- budou-li ale takto formátovaná čísla na vstupu, bude jako oddělovač řádů velmi pravděpodobně použita normální mezera (kódový bod `\u0020`)

- ◆ to je důvod, proč se prakticky téměř nikdy nepodaří takto formátovaná čísla načíst

Ukázka načtení reálného čísla. Použijeme známý `Scanner` s nastavenou lokalitou. V české lokalitě zadáme číslo bez oddělovače řádů, protože znak `\u00A0` se zadává z klávesnice špatně a znak mezery by byl interpretován jako konec čísla.

```
import java.util.*;

public class CiskaLocaleCteni {
    public static void main(String[] args) throws Exception {
        Locale[] lo = { new Locale("cs", "CZ"),
                        new Locale("en", "US") };
        Scanner sc = new Scanner(System.in).useLocale(lo[1]);
        System.out.print("Zadej realne cislo (1,234,567.89): " );
        double d = sc.nextDouble();
        System.out.println(d);

        sc = new Scanner(System.in).useLocale(lo[0]);
        System.out.print("Zadej realne cislo (1234567,89): " );
        d = sc.nextDouble();
        System.out.println(d);
    }
}
```

vypiše:

```
Zadej realne cislo (1,234,567.89): 98,765.432
98765.432
Zadej realne cislo (1234567,89): 98765,432
98765.432
```

12.7.2.2. Tisk označení měny

- zde je vhodné použít třídu `java.text.NumberFormat`

- její instanci vhodnou pro výpis měny získáme tovární metodou `getCurrencyInstance()`

- u instance `NumberFormat` pak využíváme metodu `format()`, která je zcela jiná než dříve popisovaný `String.format()`

- při výpisu se bere ohled na to, kolik desetinných míst se při výpisu měny používá
 - ♦ např. koruna se skládá ze sta haléřů – výpis je na dvě desetinná místa

Výpis měnových hodnot. Znak \$ je uveden (dle amerických zvyklostí) automaticky před číslem. Symbol měny je možné též získat ze třídy `Currency` a pak použít známý `String.format()`.

```
import java.util.*;
import java.text.*;

public class MenaLocale {
    public static void main(String[] args) throws Exception {
        NumberFormat nf;
        double d = 1234567.894;
        Locale[] lo = { new Locale("cs", "CZ"),
            new Locale("sk", "SK"),
            new Locale("en", "US"),
            new Locale("de", "DE") };
        for (int i = 0; i < lo.length; i++) {
            nf = NumberFormat.getCurrencyInstance(lo[i]);
            String s = nf.format(d);
            System.out.println(s + "\t" + lo[i].getDisplayName());
        }

        Currency cur = Currency.getInstance(lo[0]);
        String symbol = cur.getSymbol();
        System.out.format(lo[0], "%,.2f %s %n", d, symbol);
    }
}
```

vypíše:

```
1 234 567,89 Kč   čeština (Česká republika)
1 234 567,89 Sk   Slovak (Slovakia)
$1,234,567.89     English (United States)
1.234.567,89 €    German (Germany)
1 234 567,89 Kč
```

12.7.3. Formátování datumu a času

- konvence v zobrazování datumu a času jsou v různých lokalitách velmi odlišné
 - třída `java.text.DateFormat` spolehlivě funguje pro různé lokality

Ukázka různých výpisů datumu.

```
import java.util.*;
import java.text.*;

public class DatumLocale {
    public static void main(String[] args) throws Exception {
        DateFormat df;
        Date d = new Date();
        Locale[] lo = { new Locale("cs", "CZ"),
```

```
        new Locale("sv", "SE"),
        new Locale("en", "US"),
        new Locale("de", "DE") };
        for (int i = 0; i < lo.length; i++) {
            df = DateFormat.getDateInstance(DateFormat.DEFAULT, lo[i]);
            String s = df.format(d);
            System.out.println(s + "\t" + lo[i].getDisplayName());
        }
    }
}
```

vypíše:

```
4.3.2007         čeština (Česká republika)
2007-mar-04      Swedish (Sweden)
Mar 4, 2007      English (United States)
04.03.2007       German (Germany)
```

12.7.3.1. Změna množství vypisované informace

- metoda `getDateInstance()` má dva parametry – první určuje množství vypisované informace (mod výpisu)
 - konstanty `DateFormat` – `DEFAULT`, `SHORT`, `MEDIUM`, `LONG`, `FULL`

Tabulka 12.1.

	Česká republika	United States
DEFAULT	4.3.2007	Mar 4, 2007
SHORT	4.3.07	3/4/07
MEDIUM	4.3.2007	Mar 4, 2007
LONG	4. březen 2007	March 4, 2007
FULL	Neděle, 4. březen 2007	Sunday, March 4, 2007

12.7.3.2. Výpis času

- platí stejná pravidla, pouze se instance třídy `DateFormat` získá metodou `getTimeInstance()`
 - opět má jako první parametr konstanty `DateFormat` – `DEFAULT`, `SHORT`, `MEDIUM`, `LONG`, `FULL`
 - prakticky použitelné jsou:
 - ♦ `DEFAULT` – 19:53:20
 - ♦ `SHORT` – 19:53

12.7.3.3. Pomocí `format()`

- dává možnost vytvořit si vlastní formát datumu a času
 - nastavení lokality však tento výpis ovlivňuje velmi málo

- ♦ prakticky jen výpis názvů měsíců a názvů dnů slovem
- formátovací řetězec je vždy uvozen `%t` za nímž následuje další písmeno
 - to může určovat buď jeden údaj, např. `%tB` plné jméno měsíce, `%tA` plné jméno dne, `%tH` hodina ve 24hodinovém cyklu atd.
 - ♦ může být závislý na lokalitě
 - nebo určuje skupinu údajů, např. `%tT` výpis hodina:minuta:sekunda, `%tF` výpis rok-měsíc-den
 - ♦ není závislý na lokalitě
- všechny možnosti jsou přehledně popsány v `java.util.Formatter`
- velmi zajímavé je, že skutečným parametrem může být buď objekt třídy `Calendar` (nebo potomků – `GregorianCalendar`) nebo proměnná typu `long`, ve které je počet milisekund od 1.1.1970

Výpis časových údajů pomocí `format()`.

```
import java.util.*;
import java.text.*;

public class DatumLocaleFormat {
    public static void main(String[] args) throws Exception {
        Calendar c = Calendar.getInstance();
        Date d = c.getTime();
        long l = d.getTime();
        Locale[] lo = { new Locale("cs", "CZ"),
                        new Locale("sv", "SE"),
                        new Locale("en", "US"),
                        new Locale("de", "DE") };
        for (int i = 0; i < lo.length; i++) {
            System.out.format(lo[i], "%tB %tA %tF %t %s %n",
                              l, c, l, lo[i].getDisplayName());
        }
    }
}
```

vypíše:

```
březen Neděle 2007-03-04 čeština (Česká republika)
mars söndag 2007-03-04 Swedish (Sweden)
March Sunday 2007-03-04 English (United States)
März Sonntag 2007-03-04 German (Germany)
```

12.7.4. Řazení řetězců

12.7.4.1. Porovnávání řetězců

- při porovnávání jen na rovnost pomocí `String.equals()` žádné problémy nenastanou
- chceme-li např. řadit řetězce podle abecedy a k tomu využít metodu `String.compareTo()`, pak české znaky v řetězci způsobí potíže

- tato metoda vrací číslo menší nebo větší než nula v případě nerovnosti porovnávaných řetězců
- jakmile je v řetězci akcentovaný znak, pak svým kódováním zcela „vybočuje z řady“ neakcentovaných znaků
 - ♦ např. v české abecedě je posloupnost B, C, Č, D – kódové body těchto znaků jsou `\u0042`, `\u0043`, `\u010C`, `\u0044`
- pro tento případ máme třídu `java.text.Collator` spolupracující s `Locale`
 - v Java Core API v dokumentaci k `Collator` je čeština se svým třífázovým řazením uváděna jako příklad
 - „For example, in Czech, "e" and "ř" are considered primary differences, while "e" and "ě" are secondary differences, "e" and "E" are tertiary differences...”
 - prakticky to znamená, že pro porovnávání řetězců v češtině je již vše připraveno

12.7.4.2. Způsoby řazení v češtině podle normy

- řazení v češtině je definováno normou
- řazení probíhá ve třech fázích
 - v první fázi se nerozlišuje velikost znaků a také se nerozlišují některé akcenty
 - ♦ znaky s takzvanou „primární řadící platností“ jsou:
`a b c č d e f g h ch i j k l m n o p q r ř s š t u v w x y z ž`
 - ve druhé fázi se pro stejné řetězce z první fáze berou v úvahu znaky se „sekundární řadící platností“:
`á ď é ě í ň ó ť ú ů ý`
 - které se v první fázi řadily jako:
`a d e e i n o t u u y`
 - ♦ jsou-li dvě slova až na akcenty stejná, pak se slovo bez akcentů dává na první místo a slovo s akcenty na druhé
- ve třetí fázi se navíc rozlišuje i velikost znaků
 - ♦ nejprve se řadí slova s malými písmeny a pak slova s velkými písmeny
- seřadíme-li příklady uváděné v normě programem v Javě, dostaneme výsledek přesně podle normy
 - stačí jen zajistit, aby `Collator` používal správnou lokalitu
 - protože se jedná o absolutní řazení, je možné řadit nejen pole řetězců, ale i řetězce v odpovídajících kolekcích

Ukázka abecedního řazení podle normy – vstupní pole řetězců je právě v opačném pořadí.

```
import java.util.*;
import java.text.*;
```

```
import java.io.*;

public class Razeni {
    public static void main(String[] args) throws Exception {
        String[] ret = { "nový věk",
                        "Nový Svět", "Nový svět",
                        "nový Svět", "nový svět",
                        "Nový Svet", "Nový svet",
                        "abc traktoristy",
                        "ABC nástrojaře", "abc nástrojaře",
                        "ABC kováře", "ABC klempíře",
                        "abc frézaře", "ABC",
                        "Abc", "abc", "A", "a" };

        Arrays.sort(ret, new CeskyAbecedniComparator());

        for (int i = 0; i < ret.length; i++) {
            System.out.println(ret[i]);
        }
    }

    class CeskyAbecedniComparator implements Comparator<String> {
        private Collator ceskyCol = Collator.getInstance(
            new Locale("cs", "CZ"));

        public int compare(String s1, String s2) {
            return ceskyCol.compare(s1, s2);
        }
    }
}
```

vypíše:

```
a
A
abc
Abc
ABC
abc frézaře
ABC klempíře
ABC kováře
abc nástrojaře
ABC nástrojaře
abc traktoristy
Nový svet
Nový Svet
nový svět
nový Svět
Nový svět
Nový Svět
nový věk
```

12.7.5. Označování začátků a konců slov

■ detekovat jednotlivá slova je při práci s textem nutnost

- zlom řádek
- zvýraznění (výběr) slov „dvojklikem“, ...

■ pokud slova detekujeme „ručně“, používáme zásadně konstrukci

```
if (Character.isLetter(ch)) {
```

- případně ještě metody `isDigit()`, `isLetterOrDigit()`, `isLowerCase()` a `isUpperCase()`

■ na automatickou detekci slov, vět a případně i možných zlomů řádek existuje `java.text.BreakIterator`

- podobně jako u datumů a časů získáváme instanci `BreakIterator` pomocí statických metod:

- ♦ `getCharacterInstance()` – detekuje hranice znaků (znak ch)

- ♦ `getWordInstance()` – detekuje hranice slov

- ♦ `getSentenceInstance()` – detekuje hranice vět

- ♦ `getLineInstance()` – detekuje místa možného zlomu řádek

- všechny fungují pro přednastavenou lokalitu

- ♦ pro jinou lokalitu použijeme jejich přetížené verze s parametrem typu `Locale`

- instanci je nutné nejdříve nastavit zkoumaný text metodou

```
void setText(String text)
```

- `BreakIterator` používá vlastní ukazatel na hranice slov

- ♦ dá se posouvat doprava – `next()` a doleva – `previous()`

- ♦ dá se nastavit na začátek – `first()` a na konec – `last()`

- ♦ další metody vrací hraniční indexy od zadaného offsetu – `following(int offset)` a `preceding(int offset)`

- ♦ všechny vrací `int` – index do řetězce

- `BreakIterator` za samostatné úseky považuje i mezery a interpunkci

- ♦ je třeba použít metodu `Character.isLetter()` pro znak na dané hranici

- procházíme-li postupně celý řetězec, použijte se `BreakIterator.DONE` pro ukončení cyklu

Ukázka, jak lze vyseparovat jednotlivá česká slova z textu.

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



```

public class BreakIterator1 {
    public static void main(String[] args) throws Exception {
        String veta = "Příšerně 'žlutoučký' kůň úspěl ďábelské " +
            "ódy, které se nedaly poslouchat.";

        BreakIterator wbi = BreakIterator.getWordInstance();
        wbi.setText(veta);
        int zac = wbi.first();
        int kon = wbi.next();

        while (kon != BreakIterator.DONE) {
            String slovo = veta.substring(zac, kon);
            if (Character.isLetter(slovo.charAt(0))) {
                System.out.println(slovo);
            }
            zac = kon;
            kon = wbi.next();
        }
    }
}

```

vypíše:

```

Příšerně
žlutoučký
kůň
úspěl
ďábelské
ódy
které
se
nedaly
poslouchat

```

Kapitola 13. Tabulky

13.1. Základní informace

- Swing poskytuje širokou podporu pro tabulkové zobrazení
 - základní třída `javax.swing.JTable`
 - podpůrné třídy (7) a rozhraní (4) jsou v balíku `javax.swing.table`
- kromě nich jsou v balíku `javax.swing` další podpůrné třídy a rozhraní společné i pro `JTree`
 - `CellEditor`
 - `ListCellRenderer`
 - `ListSelectionModel`
 - `Renderer`
 - `AbstractCellEditor`
 - ♦ dohromady umožňují vytvořit jakkoliv složitou tabulku s libovolnými funkcemi

13.2. Základní princip

- je nutné mít dvou (tří) stupňovou organizaci
 1. na první (datové) úrovni je datový model
 - typicky splňuje rozhraní `javax.swing.table.TableModel`
 - ♦ je to rozhraní mezi datovou a prezentační vrstvou
 - datový model může být složen z více vrstev:
 - ♦ ze třídy implementující rozhraní `TableModel`
 - typicky se třída datové úrovně dědí od `AbstractTableModel`
 - pak se překrývá jen několik málo metod
 - tato třída je nutná, protože se předává do prezentační úrovně
 - viz dále třídu `PrimitivniDatovyModel`
 - ♦ z vlastních dat získaných libovolným způsobem
 - toto je „třetí“ vrstva
 - není nezbytná, data mohou být součástí předchozí třídy
 - data jsou organizována ve dvourozměrném poli typu `Object`

- není to ale podmínkou – viz příklady dále

– viz dále třídu `Data`

♦ naprosto nejjednodušší je použít `DefaultTableModel`

– pak odpadá implementace jakékoliv metody

– pro omezené možnosti se příliš nepoužívá

- všechny hodnoty musejí být v paměti
- potíže se změnou hodnot

2. na druhé (prezentační) úrovni je `JTable` jako vizuální komponenta

- typicky v konstruktoru přebírá datový model

```
new JTable(new TableModel());
```

13.2.1. Primitivní použití

■ výpis jen základních datových typů a řetězců bez nároků na vzhled

- pokud se vyskytnou v buňkách tabulky objekty, musejí mít vhodně překrytou `toString()`

13.2.1.1. Vrstva vlastních dat

■ zde jsou použita statická pole, ale principiálně na tom vůbec nezáleží

- podstatné je, aby data vyhovovala metodám datové vrstvy

■ všechny položky jsou objekty

- lze využít boxing (automatický *wrapper*) – viz `false` u jablek
- není do ale vhodné všude, protože často potřebujeme pro netriviální zobrazení odlišit typy dat, např. `Integer` od `Double`

```
import java.util.*;
```

```
public class Data {
    public static final int NAZEV = 0;
    public static final int JEDNOTKOVA_CENA = 1;
    public static final int VAHA = 2;
    public static final int DATUM_SPOTREBY = 3;
    public static final int DOVOZ = 4;

    public static String[] zhlavi = {
        "Název", "Jednotková cena", "Váha",
        "Datum spotřeby", "Dovoz"
    };

    public static Object[][] hodnoty = {
```

```
{ "jablka", new Integer(10), new Double(2.5),
  new GregorianCalendar(2005, Calendar.MAY, 1),
  false // vyuziva boxing
},
{ "banány", new Integer(25), new Double(2),
  new GregorianCalendar(2005, Calendar.MAY, 2),
  new Boolean(true)
},
{ "grapefruit", new Integer(19), new Double(0.75),
  new GregorianCalendar(2005, Calendar.MAY, 3),
  new Boolean(true)
},
{ "švestky sušené", new Integer(32), new Double(1.8),
  new GregorianCalendar(2005, Calendar.MAY, 4),
  new Boolean(false)
}
};
}
```

13.2.1.2. Vrstva metod datové vrstvy

■ při použití `DefaultTableModel` není potřebná

■ typicky se ale používá dědění od `AbstractTableModel`

```
import javax.swing.table.*;
```

```
public class PrimitivniDatovyModel
    extends AbstractTableModel {
    public int getRowCount() {
        return Data.hodnoty.length;
    }
    public int getColumnCount() {
        return Data.hodnoty[0].length;
    }
    public Object getValueAt(int row, int column) {
        return Data.hodnoty[row][column];
    }
}
```

13.2.1.3. Prezentační vrstva

```
import javax.swing.*;
import javax.swing.table.*;
```

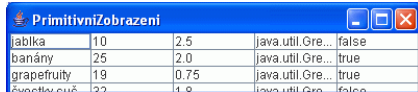
```
public class PrimitivniZobrazeni extends JFrame {
    private JComponent nastaveniTabulky() {
        JTable tabTB = new JTable(new DefaultTableModel(
            Data.hodnoty, Data.zhlavi));
        // JTable tabTB = new JTable(new PrimitivniDatovyModel());
        return tabTB;
    }
}
```

```

private PrimitivniZobrazeni() {
    super("PrimitivniZobrazeni");
    this.add(nastaveniTabulky());
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setSize(400, 90);
    this.setVisible(true);
}

public static void main(String[] args) {
    new PrimitivniZobrazeni();
}
}

```



	10	2.5	java.util.Gre...	false
jablka	10	2.5	java.util.Gre...	false
banány	25	2.0	java.util.Gre...	true
grapefruity	19	0.75	java.util.Gre...	true
kyvedlovouš	22	1.8	java.util.Gre...	false

■ nedokonalosti primitivního řešení

- všechny hodnoty se vypisují jako řetězce (přes `toString()`) a jsou zarovnány vlevo
 - ♦ to viditelně vadí u výpisu datumu
- výpis datumu neposkytuje očekávanou informaci
- není zobrazeno záhlaví
- všechny sloupce mají stejnou šířku
 - ♦ šířku sloupců není možné měnit
- není možné rolovat řádky

13.3. Vylepšování funkčnosti

13.3.1. Rolování řádek a sloupců

■ typické přidání důležité funkčnosti pomocí velmi jednoduché změny

- využije se `JScrollPane`

■ změna jen v prezentační vrstvě

■ pro změnu šířky sloupců se využívají konstanty z `JTable`

- ty mj. udávají, jak se bude měnit šířka jednotlivých sloupců, změníme-li tažením myši šířku jednoho sloupce
- podle potřeby se automaticky použije vodorovný i svislý posuvník
 - ♦ `AUTO_RESIZE_OFF` – změna šířky jen jednoho sloupce
- další nastavení používají jen svislý posuvník

♦ `AUTO_RESIZE_ALL_COLUMNS` – změna všech sloupců

♦ `AUTO_RESIZE_LAST_COLUMN` – změna jednoho sloupce a posunutí všech vpravo

♦ `AUTO_RESIZE_NEXT_COLUMN` – změna jen dvou sloupců

♦ `AUTO_RESIZE_SUBSEQUENT_COLUMNS` – proporcionální změna všech sloupců vpravo

```

public class RolovaniZobrazeni extends JFrame {
    private JComponent nastaveniTabulky() {
        JTable tabTB = new JTable(new PrimitivniDatovyModel());
        // tabTB.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
        // tabTB.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
        // tabTB.setAutoResizeMode(JTable.AUTO_RESIZE_LAST_COLUMN);
        // tabTB.setAutoResizeMode(JTable.AUTO_RESIZE_NEXT_COLUMN);
        // tabTB.setAutoResizeMode(JTable.AUTO_RESIZE_SUBSEQUENT_COLUMNS);
        JScrollPane rolSP = new JScrollPane(tabTB);
        return rolSP;
    }

    private RolovaniZobrazeni() {
        super("RolovaniZobrazeni");
        this.add(nastaveniTabulky());
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(400, 100);
        this.setVisible(true);
    }

    public static void main(String[] args) {
        new RolovaniZobrazeni();
    }
}

```



	A	B	C	D	E
jablka	10	2.5	java.util.Gr...	false	
banány	25	2.0	java.util.Gr...	true	

■ zobrazí se záhlaví sloupců, ale jen jako písmena

Poznámka

Sloupce se dají přehazovat.

13.3.2. Zobrazení záhlaví

■ nutná změna datové vrstvy

- přibude implementace `getColumnName(int column)`

■ prezentační vrstva se nemění

```

public class ZahlaviZobrazeni extends JFrame {
    private JComponent nastaveniTabulky() {

```

```

    JTable tabTB = new JTable(new ZhlaviDatovyModel());
    ...
}

public class ZhlaviDatovyModel extends AbstractTableModel {
    public int getRowCount() { ...
    public int getColumnCount() { ...
    public Object getValueAt(int row, int column) { ...

    public String getColumnName(int column) {
        return Data.zahlavi[column];
    }
}

```

Název	Jednotková...	Váha	Datum spot.	Dovoz
jablka	10	2,5	java.util.Gr...	false
banány	25	2,0	java.util.Gr...	true
grapefruity	19	0,75	java.util.Gr...	true
švestky su...	32	1,8	java.util.Gr...	false

13.4. Uživatelsky konfigurovatelné zobrazování hodnot

- primitivní zobrazení většinou nevyhovuje
- jsou tři možnosti zlepšení
 - vždy se využívají zobrazovače (*renderer*) sloupců, které lze individuálně nastavit

13.4.1. Datová vrstva informuje prezentační vrstvu o typech dat

- základní datové typy mají připraveny defaultní zobrazovače
 - stačí pouze stanovit, jakého typu jsou objekty v daném sloupci
 - ♦ metoda `Class<?> getColumnClass(int column)`
- prezentační vrstva se nemění

```

public class TypoveSloupceDatovyModel extends AbstractTableModel {
    public int getRowCount() { ...
    public int getColumnCount() { ...
    public Object getValueAt(int row, int column) { ...
    public String getColumnName(int column) { ...

    public Class<?> getColumnClass(int column) {
        if (column == Data.JEDNOTKOVA_CENA) {
            return Integer.class;
        }
        return Number.class;
    }
}

```

```

    if (column == Data.VAHA) {
        return Double.class;
    }
    // return Number.class;
}
    if (column == Data.DOVOZ) {
        return Boolean.class;
    }
    return String.class;
}
}

```

Název	Jednotková...	Váha	Datum spot.	Dovoz
jablka	10	2,5	java.util.Gr...	<input type="checkbox"/>
banány	25	2	java.util.Gr...	<input checked="" type="checkbox"/>
grapefruity	19	0,75	java.util.Gr...	<input checked="" type="checkbox"/>
švestky su...	32	1,8	java.util.Gr...	<input type="checkbox"/>

- pro čísla lze použít `Number.class`
 - je ale lepší specifikovat typ co nejpřesněji – zde `Double.class`
 - ♦ zobrazuje desetinnou čárku (přebírá nastavení z platného *Locale*), `Number.class` tečku
- `boolean` se zobrazuje jako `JCheckBox`
- vše ostatní jako `String`

13.4.2. Využití vlastního zobrazovače

Poznámka

Tam, kde nám vyhovují defaultní zobrazovače, je předchozí postup ponecháván beze změny.

- možnost specifikovat způsob zobrazení zcela podle našich představ
 - umístění, zarovnání, barvy, fonty, formáty, ...
- v datové vrstvě se většinou změní pouze:


```

public Class<?> getColumnClass(int column) {
    return getValueAt(0, column).getClass();
}

```
- v prezentační vrstvě se nastaví pro každou třídu objektů v libovolném sloupci instance vlastního zobrazovače
 - není-li explicitně specifikován zobrazovač, použije se defaultní – viz předchozí příklad pro `Integer`

```

import java.util.GregorianCalendar;
import javax.swing.*;

```

```

public class VlastniZobrazovacZobrazeni extends JFrame {
    private JComponent nastaveniTabulky() {

```

```

JTable tabTB = new JTable(new VlastniZobrazovacDatovyModel());
tabTB.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
tabTB.setDefaultRenderer(String.class, new VypisRetezce());
tabTB.setDefaultRenderer(Double.class, new VypisVahy());
tabTB.setDefaultRenderer(GregorianCalendar.class,
    new VypisDatumu());
JScrollPane rolSP = new JScrollPane(tabTB);
return rolSP;
}
...

```

■ způsob má malou nevýhodu v tom, že např. všechny případné sloupce typu `Double` se formátují stejně

- řešení viz dále

■ vlastní zobrazovač může být potomek různých tříd

13.4.2.1. Zobrazovač je potomkem třídy `DefaultTableCellRenderer`

■ nejjednodušší

- používá se pro zarovnání a změnu barev (je to potomek `JLabel`)
- nelze pracovat s hodnotou buňky

```

import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;

public class VypisRetezce extends DefaultTableCellRenderer {
    VypisRetezce() {
        this.setHorizontalAlignment(SwingConstants.CENTER);
        this.setVerticalAlignment(SwingConstants.CENTER);
        this.setBackground(Color.green);
    }
}

```

13.4.2.2. Zobrazovač implementuje rozhraní `TableCellRenderer`

■ je to potomek libovolné komponenty, který navíc implementuje rozhraní `TableCellRenderer`

- lze nastavit naprosto cokoliv i v závislosti na sloupci a řádce
 - ♦ např. dva `Double` v různých sloupcích se mohou zobrazovat různě
- dále lze nastavit různé zobrazení pro vybranou či nevybranou buňku nebo buňku s fokusem

```

import java.awt.*;
import java.util.*;
import java.text.*;
import javax.swing.*;
import javax.swing.table.*;

public class VypisVahy extends JLabel

```

```

    implements TableCellRenderer {
    static DecimalFormat df;
    static {
        NumberFormat nf = NumberFormat.getInstance(
            new Locale("cs", "CZ"));
        df = (DecimalFormat) nf;
        df.applyPattern("#,##0.00");
    }

    VypisVahy() {
        this.setHorizontalAlignment(JLabel.RIGHT);
        this.setFont(new Font("Dialog", Font.PLAIN, 12));
        this.setOpaque(true);
    }

    public Component getTableCellRendererComponent(JTable table,
        Object value, boolean isSelected,
        boolean hasFocus, int row, int column) {
        double vaha = ((Double) value).doubleValue();
        String s = df.format(vaha) + " ";
        this.setText(s);
        if (isSelected == true) {
            this.setForeground(Color.red);
            this.setBackground(Color.blue);
        }
        else {
            this.setForeground(Color.black);
            this.setBackground(Color.white);
        }
        return this;
    }
}

```

■ zde je použita komponenta `JLabel`

- nastavuje se za všech okolností český způsob vypisování desetinné čárky a odsazení řádů mezerou, výpis je na dvě desetinná místa s nevýznamovými nulami
- v konstruktoru se nastavuje zarovnání doprava a normální font
- metoda `getTableCellRendererComponent()` je z rozhraní `TableCellRenderer`
 - ♦ umožňuje získat hodnotu zobrazované buňky, její pozici na řádce a sloupci a informaci o tom, zda je vybrána a má klávesnicový fokus
 - ♦ hodnotu lze libovolně zformátovat
 - zde je u vybrané buňky změna barvy popředí
 - chceme-li nastavit i barvu pozadí, musí být komponenta nastavena na neprůhlednou

```

        this.setOpaque(true);

```

■ pro zobrazení hodnoty možné použít libovolnou vhodnou komponentu, málokdy však použijeme něco jiného než `JLabel` nebo `JCheckBox`

- pro výpis datumu je použit stejný postup

```
import java.awt.*;
import java.util.*;
import java.text.*;
import javax.swing.*;
import javax.swing.table.*;

public class VypisDatumu extends JLabel
    implements TableCellRenderer {
    static SimpleDateFormat sdf = new SimpleDateFormat("d.MM.yyyy");
    VypisDatumu() {
        this.setFont(new Font("MonoSpaced", Font.BOLD, 12));
        this.setHorizontalAlignment(JLabel.RIGHT);
        this.setForeground(Color.blue);
        this.setBackground(Color.yellow);
        this.setOpaque(true);
    }

    public Component getTableCellRendererComponent(JTable table,
        Object value, boolean isSelected,
        boolean hasFocus, int row, int column) {
        String s = sdf.format(((GregorianCalendar) value).getTime());
        this.setText(s);
        return this;
    }
}
```

Název	Jednotková...	Váha	Datum spot...	Dovoz
jablka	10	2,50	1. 05. 2005	<input type="checkbox"/>
banány	25	2,00	2. 05. 2005	<input checked="" type="checkbox"/>
grapefruity	19	0,75	3. 05. 2005	<input checked="" type="checkbox"/>
švestky suš...	32	1,80	4. 05. 2005	<input type="checkbox"/>

13.5. Práce s jednotlivými sloupci na úrovni prezentační vrstvy

- na tabulku lze pohlížet jako na pole sloupců
 - ♦ objekty sloupců lze získat na prezentační vrstvě jako instance `TableColumn`
 - ♦ každému sloupci pak lze přiřadit množství vlastností
 - nejužívanější jsou nastavení šířky sloupce
 - `setMinWidth(40)`; – méně nelze zmenšit
 - `setPreferredWidth(50)`; – zobrazená šířka
 - `setMaxWidth(100)`; – více nelze roztáhnout
 - sloupce lze nastavit zobrazovač

- nesouvisí vůbec s informacemi o třídě poskytovanými datovou vrstvou
 - lze používat různé způsoby zobrazení nastavené jen na prezentační úrovni
- je to nejvhodnější způsob

```
public class NastaveniSloupceZobrazeni extends JFrame {
    private JComponent nastaveniTabulky() {
        JTable tabTB = new JTable(new ZhlaviDatovyModel());
        tabTB.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
        TableColumn tc = tabTB.getColumnModel().getColumn(Data.VAHA);
        tc.setMinWidth(40);
        tc.setPreferredWidth(50);
        tc.setMaxWidth(100);
        tc.setCellRenderer(new VypisVahy());
        JScrollPane rolSP = new JScrollPane(tabTB);
        return rolSP;
    }
    ...
}
```

Název	Jednotková cena	Váha	Datum spotřeby	Dovoz
jablka	10	2,50	java.util.Greg...	false
banány	25	2,00	java.util.Greg...	true
grapefruity	19	0,75	java.util.Greg...	true
švestky suš...	32	1,80	java.util.Greg...	false

13.6. Práce se záhlavím

- potřebujeme vypsát záhlaví jinak, než standardním fontem do jedné řádky
 - ♦ princip je stejný, jako při použití formátovače hodnot ve sloupcích
 - ♦ pouze se použije `setHeaderRenderer()`

```
public class VlastniZahlaviZobrazeni extends JFrame {
    private JComponent nastaveniTabulky() {
        JTable tabTB = new JTable(new ZhlaviDatovyModel());
        tabTB.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
        VypisZahlavi z = new VypisZahlavi();
        TableColumnModel tcm = tabTB.getColumnModel();
        for (int i = 0, n = tcm.getColumnCount(); i < n; i++) {
            TableColumn tc = tcm.getColumnModel(i);
            tc.setHeaderRenderer(z);
        }
        JScrollPane rolSP = new JScrollPane(tabTB);
        return rolSP;
    }
    ...
}
```

- zobrazovač záhlaví se vytváří principiálně zcela stejně, jako zobrazovač normálních buněk

```
import java.awt.*;
import javax.swing.*;
```

```
import javax.swing.table.*;

public class VypisZahlavi extends JPanel
    implements TableCellRenderer {
    public Component getTableCellRendererComponent(JTable table,
        Object value, boolean isSelected,
        boolean hasFocus, int row, int column) {
        this.removeAll();
        String s = (String) value;
        String[] radky = s.split(" "); // kazde slovo na nove radce
        this.setLayout(new GridLayout(radky.length, 1));
        for (int i = 0; i < radky.length; i++) {
            JLabel l = new JLabel(radky[i], JLabel.CENTER);
            // l.setFont(new Font("Dialog", Font.PLAIN, 12));
            // l.setForeground(Color.red);
            // l.setBackground(Color.blue);
            // l.setOpaque(true);
            this.add(l);
        }
        LookAndFeel.installBorder(this, "TableHeader.cellBorder");
        return this;
    }
}
```

Název	Jednotková cena	Váha	Datum spotřeby	Dovoz
jablka	10	2.5	java.util.Gr...	false
banány	25	2.0	java.util.Gr...	true
grapefruity	19	0.75	java.util.Gr...	true
švestky su...	32	1.8	java.util.Gr...	false

■ volání `this.removeAll()`; je nezbytné

- bez ní by se texty smíchaly do jednoho
- ♦ metoda `getTableCellRendererComponent()` je totiž volána pro záhlaví všech sloupců

13.7. Změna hodnot v tabulce

Poznámka

U všech dále popisovaných způsobů dojde k překreslení změněného obsahu buňky, tj. k viditelnému provedení akce, až po kliknutí na jinou buňku. Potvrzení změny stiskem <Enter> nestačí. Návod na změnu tohoto chování viz dále a též v Java Tutorial.

13.7.1. Editace základních datových typů

- pro základní datové typy např. `Integer`, `Double`, `Boolean` nebo `String` je změna jednoduchá – nemusí se připravovat žádný editor
- je třeba pouze změnit datovou vrstvu přidáním metod:

- ♦ `boolean isCellEditable(int row, int column)`
 - pokyn pro prezentační vrstvu, aby této buňce dovolila změnu
 - v příkladu je změna zakázána pro sloupec „Název“
- ♦ `void setValueAt(Object value, int row, int column)`
 - datová vrstva provedenou změnu uloží
- žádná další akce (např. v prezentační vrstvě) není třeba
- ♦ pokud jsou v prezentační vrstvě použity vlastní zobrazovače, nijak to nevedí
- jednoduchost je zaplácena některými potížemi
- položky, které je možno měnit, se vybírají dvojklikem
 - ♦ to není příliš vhodné chování, protože běžně očekáváme pouze jedno kliknutí
- při zadávání reálného čísla je nutné jako desetinný oddělovač použít tečku, nikoliv zobrazovanou čárku

```
public class ZmenaDatovyModel extends AbstractTableModel {
    public int getRowCount() { ...
    public int getColumnCount() { ...
    public Object getValueAt(int row, int column) { ...
    public String getColumnName(int column) { ...
    public Class<?> getColumnClass(int column) { ...

    public boolean isCellEditable(int row, int column) {
        if (column == Data.NAZEV) {
            return false;
        }
        return true;
    }

    public void setValueAt(Object value, int row, int column) {
        Data.hodnoty[row][column] = value;
    }
}
```

■ v prezentační vrstvě nejsou nutná žádná speciální nastavení

```
public class ZmenaZobrazeni extends JFrame {
    private JComponent nastaveniTabulky() {
        JTable tabTB = new JTable(new ZmenaDatovyModel());
        tabTB.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
        tabTB.setDefaultRenderer(Double.class, new VypisVahy());
        tabTB.setDefaultRenderer(GregorianCalendar.class,
            new VypisDatumu());
        JScrollPane rolSP = new JScrollPane(tabTB);
        return rolSP;
    }
    ...
}
```

Název	Jednotková...	Váha	Datum spot.	Dovoz
jablka	10	2,00	1.05.2005	<input checked="" type="checkbox"/>
banány	25	2,00	2.05.2005	<input checked="" type="checkbox"/>
grapefruity	19	0,75	3.05.2005	<input checked="" type="checkbox"/>
švestky su...	32	1,80	4.05.2005	<input type="checkbox"/>

13.7.2. Použití editoru na bázi DefaultCellEditor

- pro nejčastější předpokládané způsoby editace lze využít třídy DefaultCellEditor

- tyto způsoby editace jsou pro hodnoty typu:
 - zapnuto/vypnuto – využívá se JCheckBox
 - výběr z nabídnutých možností – využívá se JComboBox
 - obecný řetězec – využívá se JTextField nebo často jeho potomek JFormattedTextField
- DefaultCellEditor dokáže pracovat pouze s nimi

- DefaultCellEditor se používá v prezentační vrstvě pro editaci hodnot v konkrétních sloupcích

- je to častý případ použití, protože v různých sloupcích mohou být hodnoty stejných datových typů, ale my potřebujeme, aby se editovaly rozdílně
- pro konkrétní sloupec lze zvolený editor nastavit metodou setCellEditor()
 - existují dva základní způsoby použití – viz následující příklady

- následující příklady pro změnu položky ve sloupci Dovoz budou mít s využitím JComboBox tuto funkčnost

Název	Jednotková...	Váha	Datum spot.	Dovoz
jablka	10	2,00	1.06.2005	ano
banány	25	2,00	2.05.2005	ano
grapefruity	19	0,75	3.05.2005	NE
švestky su...	32	1,80	4.05.2005	

Poznámka

Použití JComboBox pro hodnoty typu zapnuto/vypnuto je ve skutečnosti zbytečný „luxus“.

- aby bylo možné tuto funkčnost dosáhnout, musí se oproti předchozímu příkladu pozměnit metoda setValueAt() v datové vrstvě

```
public class ZmenaDatovyModel extends AbstractTableModel {
    public int getRowCount() { ...

    public void setValueAt(Object value, int row, int column) {
        if (value instanceof String
            && column == Data.DOVOZ) {
            if (value.toString().equals("ano") == true) {
                Data.hodnoty[row][column] = new Boolean(true);
            }
            else {
```

```
        Data.hodnoty[row][column] = new Boolean(false);
    }
}
else {
    Data.hodnoty[row][column] = value;
}
}
```

13.7.2.1. Využití instance JComboBox přímo v prezentační vrstvě

- v prezentační vrstvě vytvoříme komponentu JComboBox jako část editoru, která se předá do konstruktoru DefaultCellEditor

- pomocí metody setCellEditor() ze třídy TableColumn pak editor přiřadíme konkrétnímu sloupci

```
public class ZmenaZobrazeni extends JFrame {
    private JComponent nastaveniTabulky() {
        JTable tabTB = new JTable(new ZmenaDatovyModel());
        tabTB.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
        tabTB.setDefaultRenderer(Double.class, new VypisVahy());
        tabTB.setDefaultRenderer(GregorianCalendar.class,
            new VypisDatumu());
        tabTB.setDefaultEditor(GregorianCalendar.class,
            new DatumEditor());

        TableColumn tc = tabTB.getColumnModel().getColumn(Data.DOVOZ);
        JComboBox dovozJCB = new JComboBox();
        dovozJCB.addItem("ano");
        dovozJCB.addItem("NE");
        tc.setCellEditor(new DefaultCellEditor(dovozJCB));
    }
}
```

13.7.2.2. Využití třídy DefaultCellEditor pro konstrukci vlastní třídy

- předchozí způsob, kdy se přímo v prezentační vrstvě připraví část editoru, je nekonceptní – míchají se do sebe různé věci

- třídu vlastního editoru lze připravit děděním třídy DefaultCellEditor

- základní použití s komponentami JTextField, JCheckBox a JComboBox je jednoduché
- ukázka sofistikovaného využití je uvedena v Java Tutorial

- třída editoru

```
import javax.swing.*;
```

```
public class DovozeEditorDefault extends DefaultCellEditor {
    public DovozeEditorDefault() {
        super(new JComboBox());
    }
}
```



```

        JComboBox jcb = (JComboBox) this.getComponent();
        jcb.addItem("ano");
        jcb.addItem("NE");
    }
}

```

■ prezentační vrstva

```

public class ZmenaZobrazeni extends JFrame {
    private JComponent nastaveniTabulky() {
        JTable tabTB = new JTable(new ZmenaDatovyModel());
        tabTB.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
        tabTB.setDefaultRenderer(Double.class, new VypisVahy());
        tabTB.setDefaultRenderer(GregorianCalendar.class,
            new VypisDatumu());
        TableColumn tc = tabTB.getColumnModel().getColumn(Data.DOVOZ);
        tc.setCellEditor(new DovozEditorDefault());
    }
}

```

13.7.3. Vytvoření vlastního editoru na bázi libovolné komponenty

■ nestačí-li nám předchozí způsoby, lze (stejně jako u zobrazovačů) vytvořit třídu velmi obecného editoru

- vnitřek editoru může tvořit libovolná vhodná komponenta
- editor musí být třída implementující rozhraní `TableCellEditor` a potažmo i rozhraní `CellEditor`
 - ♦ zde je celkem 8 metod včetně řešení událostí
 - ♦ je to obecné, ale poněkud zdlouhavé řešení
- pro zjednodušení se běžně používá následující postup:
 - ♦ je vhodné dědit od `javax.swing.AbstractCellEditor`, která odstíní od nutnosti řešit problémy s událostmi
 - stačí implementovat metodu `getCellEditorValue()`
 - vrací nově nastavenou hodnotu do metody `setValueAt()` datové vrstvy
 - ♦ dále třída musí implementovat rozhraní `javax.swing.table.TableCellEditor`
 - metodu `getTableCellEditorComponent()`
 - v ní se stanoví, jak výsledek práce editoru změní hodnotu vybrané (měněné) buňky datové vrstvy

■ daný editor musíme nastavit na prezentační úrovni dvěma základními způsoby

- jako defaultní pro všechny sloupce, ve kterých se objekt dané třídy vyskytuje
- pro konkrétní sloupec (jako v předchozím případě)

```

public class ZmenaZobrazeni extends JFrame {
    private JComponent nastaveniTabulky() {
        JTable tabTB = new JTable(new ZmenaDatovyModel());
        tabTB.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
        tabTB.setDefaultRenderer(Double.class, new VypisVahy());
        tabTB.setDefaultRenderer(GregorianCalendar.class,
            new VypisDatumu());
        // první způsob
        tabTB.setDefaultEditor(Boolean.class, new DovozEditor());
        // druhý způsob
        // TableColumn tc = tabTB.getColumnModel().getColumn(Data.DOVOZ);
        // tc.setCellEditor(new DovozEditor());
    }
}

```

■ pro editaci dovozu použijeme (jako dříve) komponentu `JComboBox`

- funkčnost bude stejná, jako v předchozích případech

```

import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;

```

```

public class DovozEditor extends AbstractCellEditor
    implements TableCellEditor {

    private JComboBox dovozCB;

    public DovozEditor() {
        dovozCB = new JComboBox();
        dovozCB.addItem("ano"); // index 0
        dovozCB.addItem("NE"); // index 1
    }

    public Object getCellEditorValue() {
        return dovozCB.getSelectedItem();
    }

    public Component getTableCellEditorComponent(JTable table,
        Object value, boolean isSelected,
        int row, int column) {
        boolean b = (Boolean) value;
        dovozCB.setSelectedIndex(b == true ? 0 : 1);
        return dovozCB;
    }
}

```

Název	Jednotková	Váha	Datum spot.	Dvov
jablka	10	2,00	1.06.2005	NE
banány	25	2,00	2.05.2005	ano
grapefruity	19	0,75	3.05.2005	NE
švestky su...	32	1,80	4.05.2005	

■ pro editaci datumu použijeme komponentu `JTextField`

```
import java.awt.*;
import java.text.SimpleDateFormat;
import java.util.GregorianCalendar;
import javax.swing.*;
import javax.swing.table.*;

public class DatumEditor extends AbstractCellEditor
    implements TableCellEditor {

    static SimpleDateFormat sdf = new SimpleDateFormat("d.MM.yyyy");
    private JTextField datumTF;

    public DatumEditor() {
        datumTF = new JTextField();
        datumTF.setFont(new Font("MonoSpaced", Font.BOLD, 12));
        datumTF.setHorizontalAlignment(JLabel.RIGHT);
        datumTF.setForeground(Color.black);
    }

    public Object getCellEditorValue() {
        String[] udaje = datumTF.getText().split("\\D");
        int den = Integer.parseInt(udaje[0]);
        int mesic = Integer.parseInt(udaje[1]);
        int rok = Integer.parseInt(udaje[2]);
        GregorianCalendar gc = new GregorianCalendar(rok,
            mesic - 1, den);

        return gc;
    }

    public Component getTableCellEditorComponent(JTable table,
        Object value, boolean isSelected,
        int row, int column) {
        GregorianCalendar gc = (GregorianCalendar) value;
        String s = sdf.format(gc.getTime());
        datumTF.setText(s);
        return datumTF;
    }
}
```

Název	Jednotková	Váha	Datum spot.	Dovoz
jablka	10	2,00	1. 06. 2005	<input type="checkbox"/>
banány	25	2,00	2. 05. 2005	<input type="checkbox"/>
grapefruity	19	0,75	3. 05. 2005	<input checked="" type="checkbox"/>
jablka	10	2,50	1. 05. 2005	<input checked="" type="checkbox"/>
švestky su...	32	1,80	4. 05. 2005	<input type="checkbox"/>

13.8. Řazení v tabulce

- není nutné implementovat, protože v Java Tutoriál je připravená třída `TableSorter`

Poznámka

Měla by být součástí Java Core API od 1.6.

- jedná se o návrhový vzor dekorátor pro `TableModel`

- pro řazení stačí jen klikat na záhlaví
 - řadí vzestupně i sestupně
- po Ctrl-click je možné sekundární řazení podle dalšího sloupce

```
import java.util.GregorianCalendar;
import javax.swing.*;

public class RazeniZobrazeni extends JFrame {
    private JComponent nastaveniTabulky() {
        TableSorter sorter = new TableSorter(new ZmenaDatovyModel());
        JTable tabTB = new JTable(sorter);
        sorter.setTableHeader(tabTB.getTableHeader());
        tabTB.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
        tabTB.setDefaultRenderer(Double.class, new VypisVahy());
        tabTB.setDefaultRenderer(GregorianCalendar.class,
            new VypisDatumu());
        tabTB.setDefaultEditor(GregorianCalendar.class,
            new DatumEditor());
        JScrollPane rolSP = new JScrollPane(tabTB);
        return rolSP;
    }
    ...
}
```

Název	Jednotková	Váha	Datum spot.	Dovoz
banány	25	2,00	2. 05. 2005	<input checked="" type="checkbox"/>
grapefruity	19	0,75	3. 05. 2005	<input checked="" type="checkbox"/>
jablka	10	2,50	1. 05. 2005	<input checked="" type="checkbox"/>
švestky su...	32	1,80	4. 05. 2005	<input type="checkbox"/>

13.9. Data jsou organizována jako na sobě nezávislá pole

- často se stane, že data nejsou homogenní a/nebo není vhodné, aby byla uložena v jednom dvourozměrném poli
- první dva příklady budou dodržovat dosud používanou architekturu tří tříd, tj. data, třída s metodami datové vrstvy a prezentační vrstva
 - v těchto příkladech je ukázáno, jak použít v jednom sloupci nehomogenní data (viz „suma“ ve sloupci „Jednotková cena“)
 - poslední sloupec „Cena“ je kompletně vypočítáván z údajů „Jednotková cena“ a „Váha“
 - stejně tak jsou vypočítávány buňky sumy vah a sumy ceny
 - tabulka bude vypadat takto:

Název	Jednotková cena	Váha	Cena
jablka	10	2,50	25,00
banány	25	2,00	50,00
grapefruity	19	0,75	14,25
švestky sušené	32	1,80	57,60
suma		7,05	146,85

- třetí příklad sloučí obě třídy datové vrstvy do jedné

13.9.1. Ukázka vzájemného provázání dat

- třída `DataVypocet` má místo jednoho dvourozměrného pole několik jednorozměrných polí představujících jednotlivé sloupce

- tato pole spolu vzájemně fyzicky nesouvisejí
 - ♦ logická souvislost (tj. spojení do jedné tabulky) se provede až ve třídě `ZmenaVypocetDatovyModel`
- třída je doplněna o čtyři statické metody:
 - ♦ `prepocetiCenu()` – vypočítá cenu zboží součinem jednotkové ceny a váhy
 - ♦ `sumaVah()` – vypočítá celkovou váhu zboží
 - ♦ `sumaCen()` – vypočítá celkovou cenu zboží
 - ♦ `pocatecniVypocetDat()` – provede všechny předchozí výpočty

```
public class DataVypocet {
    public static final int NAZEV = 0;
    public static final int JEDNOTKOVA_CENA = 1;
    public static final int VAHA = 2;
    public static final int CENA = 3;

    public static String[] zahlavi = {
        "Název", "Jednotková cena", "Váha", "Cena"
    };

    public static String[] nazev = {
        "jablka", "banány", "grapefruity", "švestky sušené", ""
    };

    // musi byt Object kvuli poslední sume
    public static Object[] jednotkovaCena = {
        new Integer(10), new Integer(25), new Integer(19),
        new Integer(32), "suma"
    };

    public static Double[] vaha = {
        new Double(2.5), new Double(2), new Double(0.75),
        new Double(1.8), new Double(0)
    };
}
```

```
public static Double[] cena = {
    new Double(0), new Double(0), new Double(0),
    new Double(0), new Double(0)
};
```

```
public static void prepocetiCenu(int radka) {
    cena[radka] =
        ((Integer) jednotkovaCena[radka])
        * ((Double) vaha[radka]);
}
```

```
public static void sumaVah() {
    double sumaV = 0.0;
    for (int i = 0; i < vaha.length - 1; i++) {
        sumaV += (Double) vaha[i];
    }
    vaha[vaha.length - 1] = new Double(sumaV);
}
```

```
public static void sumaCen() {
    double sumaC = 0.0;
    for (int i = 0; i < cena.length - 1; i++) {
        sumaC += (Double) cena[i];
    }
    cena[cena.length - 1] = new Double(sumaC);
}
```

```
// pocatecni vypocet cen a obou sum
public static void pocatecniVypocetDat() {
    for (int i = 0; i < cena.length - 1; i++) {
        prepocetiCenu(i);
    }
    sumaVah();
    sumaCen();
}
```

- třída `ZmenaVypocetDatovyModel` má oproti všem dosud uváděným příkladům výrazně změněnou metodu `getValueAt()`

- v této metodě se logicky spojují jednotlivé fyzicky nezávislé pole (tj. sloupce) dat
- podobná změna je i v metodě `setValueAt()`
 - ♦ zde je navíc volána metoda `fireTableCellUpdated()`, která zabezpečí, že po editaci některé buňky prezentační vrstva aktualizuje i buňku fyzicky nesouvisející (není na stejné řádce)

```
import javax.swing.table.*;
```

```
public class ZmenaVypocetDatovyModel
    extends AbstractTableModel {

    public int getRowCount() {
        return DataVypocet.nazev.length;
    }
}
```

```

    }

    public int getColumnCount() {
        return DataVypocet.zahlavi.length;
    }

    public Object getValueAt(int row, int column) {
        switch (column) {
            case DataVypocet.NAZEVI:
                return DataVypocet.nazev[row];

            case DataVypocet.JEDNOTKOVA_CENA:
                return DataVypocet.jednotkovaCena[row];

            case DataVypocet.VAHA:
                return DataVypocet.vaha[row];

            case DataVypocet.CENA:
                return DataVypocet.cena[row];
        }
        return null;
    }

    public String getColumnName(int column) {
        return DataVypocet.zahlavi[column];
    }

    public Class<?> getColumnClass(int column) {
        return getValueAt(0, column).getClass();
    }

    public boolean isCellEditable(int row, int column) {
        if (row == DataVypocet.nazev.length - 1) {
            return false;
        }
        if (column == DataVypocet.JEDNOTKOVA_CENA
            || column == DataVypocet.VAHA) {
            return true;
        }
        return false;
    }

    public void setValueAt(Object value, int row, int column) {
        switch (column) {
            case DataVypocet.JEDNOTKOVA_CENA:
                DataVypocet.jednotkovaCena[row] = (Integer) value;
                DataVypocet.prepocetiCenu(row);
                DataVypocet.sumaCen();
                this.fireTableCellUpdated(DataVypocet.cena.length - 1,
                    DataVypocet.CENA);

                break;

            case DataVypocet.VAHA:

```

```

                DataVypocet.vaha[row] = (Double) value;
                DataVypocet.sumaVah();
                this.fireTableCellUpdated(DataVypocet.vaha.length - 1,
                    DataVypocet.VAHA);

                DataVypocet.prepocetiCenu(row);
                DataVypocet.sumaCen();
                this.fireTableCellUpdated(DataVypocet.cena.length - 1,
                    DataVypocet.CENA);

                break;
        }
    }
}

```

■ **třída VypisDoubleVypocet, jako zobrazovač hodnot typu Double, je velmi podobná dříve používané třídě VypisVahy**

```

import java.awt.*;
import java.util.*;
import java.text.*;
import javax.swing.*;
import javax.swing.table.*;

public class VypisDoubleVypocet extends JLabel
    implements TableCellRenderer {
    static DecimalFormat df;
    static {
        NumberFormat nf = NumberFormat.getNumberInstance(
            new Locale("cs", "CZ"));
        df = (DecimalFormat) nf;
        df.applyPattern("#,##0.00");
    }
    VypisDoubleVypocet() {
        this.setHorizontalAlignment(JLabel.RIGHT);
        this.setFont(new Font("Dialog", Font.PLAIN, 12));
    }

    public Component getTableCellRendererComponent(JTable table,
        Object value, boolean isSelected, boolean hasFocus,
        int row, int column) {
        double vaha = ((Double) value).doubleValue();
        String s = df.format(vaha) + " ";
        this.setText(s);
        return this;
    }
}

```

■ **třída ZmenaZobrazeniVypocet je velmi podobná dříve uváděným třídám prezentační vrstvy**

Poznámka

Volání metody `DataVypocet.pocatecniVypocetDat()` by mělo být spíše v konstruktoru třídy `ZmenaVypocetDatovyModel`. Zde je uvedeno proto, aby mohl snadno navazovat další příklad.

```
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;

public class ZmenaZobrazeniVypocet extends JFrame {
    private JComponent nastaveniTabulky() {
        DataVypocet.pocatecniVypocetDat();
        JTable tabTB = new JTable(new ZmenaVypocetDatovyModel());
        tabTB.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
        tabTB.setDefaultRenderer(Double.class,
            new VypisDoubleVypocet());
        JScrollPane rolSP = new JScrollPane(tabTB);
        return rolSP;
    }

    private ZmenaZobrazeniVypocet() {
        super("ZmenaZobrazeniVypocet");
        this.add(nastaveniTabulky());
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(400, 180);
        this.setVisible(true);
    }

    public static void main(String[] args) {
        new ZmenaZobrazeniVypocet();
    }
}
```

13.9.2. Ukázka načítání dat ze souboru

■ všechny dosud uváděné příklady měly zobrazovaná data napevno nastavená ve třídě datové vrstvy

- to je nerealistický příklad, protože zobrazovaná data se nejčastěji načtou ze souboru

- ♦ soubor s názvem `jidlo.txt` má obsah

```
10;jablka;2.5
25;banány;2
19;grapefruit;0.75
32;švestky sušené;1.8
15;hrušky;1.5
22;pomeranče;3.2
40;fíky;0.5
```

- pro načítání stačí pouze dodat třídu `DataVypocetSoubor`, která údaje ze souboru načte
 - ♦ tak nahradí přednastavené hodnoty ve třídě `DataVypocet`
 - řádky ze souboru se parsují a jednotlivé údaje se ukládají do pomocných seznamů (`ArrayList`)
 - tím se nemusíme starat o velikost souboru
 - po načtení se seznamy převedou na pole metodou `toArray()`

- jejím parametrem je pole nulové velikosti typu převáděného pole – „trik“ z kolekce

- ♦ díky tomuto postupu není vůbec potřebná změna ve třídách `DataVypocet` a `ZmenaVypocetDatovyModel`

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.util.ArrayList;

public class DataVypocetSoubor {
    public static void nactiSoubor(String jmeno) {
        ArrayList<Object> jednotkovaCenaAr = new ArrayList<Object>();
        ArrayList<String> nazevAr = new ArrayList<String>();
        ArrayList<Double> vahaAr = new ArrayList<Double>();

        try {
            BufferedReader bfr = new BufferedReader(
                new FileReader(jmeno));
            String radka;
            while ((radka = bfr.readLine()) != null) {
                String[] polozky = radka.split(";");
                jednotkovaCenaAr.add(new Integer(polozky[0]));
                nazevAr.add(polozky[1]);
                vahaAr.add(new Double(polozky[2]));
            }
            bfr.close();
        } catch (Exception e) {
            e.printStackTrace();
        }

        // pridani radky pro sumy
        jednotkovaCenaAr.add("suma");
        nazevAr.add("");
        vahaAr.add(new Double(0));

        // prevod na pole
        DataVypocet.nazev = nazevAr.toArray(new String[0]);
        DataVypocet.jednotkovaCena = jednotkovaCenaAr.toArray(new Object[0]);
        DataVypocet.vaha = vahaAr.toArray(new Double[0]);

        // vytvoreni pole pro cenu
        DataVypocet.cena = new Double[DataVypocet.vaha.length];
    }
}
```

- v prezentační vrstvě je nutné zavolat metodu pro načtení ze souboru

Poznámka

Ve skutečném příkladě by se tato činnost řešila v přetíženém konstruktoru třídy `ZmenaVypocetDatovyModel`, kterému by se jako skutečný parametr předal název souboru.


```
public class ZmenaZobrazeniVypocet extends JFrame {
    private JComponent nastaveniTabulky() {
```

```

DataVypocetSoubor.nactiSoubor("jidlo.txt");
DataVypocet.pocatecniVypocetDat();
JTable tabTB = new JTable(new ZmenaVypocetDatovyModel());
tabTB.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
tabTB.setDefaultRenderer(Double.class,
                           new VypisDoubleVypocet());
JScrollPane rolSP = new JScrollPane(tabTB);
return rolSP;
}
...

```

- program zobrazí



Název	Jednotková cena	Váha	Cena
jablka	10	2,50	25,00
banány	25	2,00	50,00
grapefruity	19	0,75	14,25
švestky sušené	32	1,80	57,60
hrušky	15	1,50	22,50
pomeranče	22	3,20	70,40
řiky	40	0,50	20,00
suma		12,25	259,75

13.9.3. Data jsou součástí třídy implementující TableModel

- může se stát, že data nebudeme načítat ze souboru, ale potřebujeme je vytvořit programově v určité velikosti

- pak je vhodné mít tato data rovnou ve třídě implementující TableModel
 - ♦ zbavíme se jedné třídy datové vrstvy

- příklad připraví matici o zadaném počtu řádků a sloupců a naplní ji celými čísly

- v záhlaví budou čísla sloupců
- zobrazí se např.:



	1.	2.	3.	4.
11	12	13	14	
21	22	23	24	
31	32	33	34	
41	42	43	44	
51	52	53	54	

- datová vrstva

```

import javax.swing.table.*;

public class VypoctenoDatovyModel extends AbstractTableModel {
    private int nRadek;
    private int nSloupcu;
    private String[] zhlavi;
    private Integer[][] hodnoty;

```

```

    public VypoctenoDatovyModel(int nRadek, int nSloupcu) {
        this.nRadek = nRadek;
        this.nSloupcu = nSloupcu;
        hodnoty = new Integer[nRadek][nSloupcu];
        zhlavi = new String[nSloupcu];
        naplneniHodnot();
        naplneniZhlavi();
    }

```

```

    private void naplneniHodnot() {
        for (int i = 0; i < nRadek; i++) {
            for (int j = 0; j < nSloupcu; j++) {
                hodnoty[i][j] = (i + 1) * 10 + (j + 1);
            }
        }
    }

```

```

    private void naplneniZhlavi() {
        for (int j = 0; j < nSloupcu; j++) {
            zhlavi[j] = "" + (j + 1) + ".";
        }
    }

```

```

    public int getRowCount() {
        return nRadek;
    }

```

```

    public int getColumnCount() {
        return nSloupcu;
    }

```

```

    public Object getValueAt(int row, int column) {
        return hodnoty[row][column];
    }

```

```

    public String getColumnName(int column) {
        return zhlavi[column];
    }
}

```

- prezentační vrstva

```

public class VypoctenoZobrazeni extends JFrame {
    private JComponent nastaveniTabulky() {
        JTable tabTB = new JTable(new VypoctenoDatovyModel(5, 4));
        tabTB.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
        JScrollPane rolSP = new JScrollPane(tabTB);
        return rolSP;
    }
    ...
}

```