Přednášky z JXT

1. XML – obecné informace 1
1.1. Základní charakteristiky 1
1.1.1. Co XML není 2
1.1.2. Ověření správnosti 2
1.1.3. Vývoj XML 2
1.1.4. Dvě hlavní oblasti použití 3
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 8
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. Literatura 16
2. Java a národní prostředí, Ant 18
2.1. Java a národní prostředí 18
2.1.1. Úvodní informace 18
2.1.2. Kódování 18
2.1.3. Čeština v programu 22
2.1.4. České výpisy na konzoli 26
2.1.5. Čeština v souborech 28
2.1.6. Převody mezi různými kódováními uvnitř programu 29
2.1.7. Třída Locale 30
2.1.8. Řazení řetězců 32
2.2. Ant – *Another Neat Tool* 34
2.2.1. Základní informace 34
2.2.2. Jak Ant získat 35
2.2.3. Použití 35
2.2.4. Použití `<property>` 38
2.2.5. Nastavování cest a opětovné použití nastavení 41
2.2.6. Nastavení jmen souborů 42
2.2.7. Kombinování `<path>` a `<fileset>` 43
2.2.8. Možnosti `<target>` 43
2.2.9. Když je něco špatně 48
2.2.10. Úkoly (tasks) 48
2.2.11. Přehled a použití často používaných úkolů 50
2.2.12. XJC pro JDK 1.6 57
2.2.13. Ukázka komplexního projektu použitelného v praxi 58
2.2.14. Doporučeno k přečtení: 60
3. Arrays, řazení, kolekce a genericita 61
3.1. Podpora práce s poli – třída Arrays 61
3.1.1. Možnosti třídy Arrays 61
3.2. Řazení objektů 62

3.2.1. Přirozené řazení (*natural ordering*) ... 63
3.2.2. Absolutní řazení (*total ordering*) ... 65
3.3. Kolekce a genericita – úvodní informace ... 67
3.4. Typové parametry a parametrizované typy ... 71
3.4.1. Použití žolíků – *unbounded wildcard* ... 73
3.4.2. Omezené využití žolíků – *bounded wildcard* ... 74
3.5. Rozhraní `Collection` ... 75
3.6. Rozhraní `List` ... 76
3.6.1. Implementace pomocí `ArrayList` ... 77
3.7. Zajištění algoritmů – třída `Collections` ... 81
3.8. Postupný průchod kolekcí ... 84
3.8.1. For-Each ... 84
3.8.2. Iterátory ... 85
3.9. Výhodnost jednotlivých seznamů ... 88
3.10. Ochrana proti nekonzistenci dat ... 89
3.11. Množiny – rozhraní Set ... 90
3.11.1. Práce s vlastní třídou v množině ... 92
3.11.2. Problémy objektů v hešovacích třídách ... 99
3.11.3. Použití `Collections` ... 105
3.11.4. Rozhraní SortedSet ... 105
3.11.5. Množinové operace a triky ... 106
3.12. Mapy – rozhraní Map ... 107
3.12.1. Třída TreeMap ... 110
3.13. Složitější datové struktury ... 112
4. Schémové jazyky DTD a XSD ... 115
4.1. Význam schémových jazyků (schémat) ... 115
4.2. DTD – *Document Type Definition* ... 115
4.2.1. Výhody a nevýhody DTD ... 115
4.2.2. Spojení DTD a XML ... 116
4.2.3. Prvky a struktura DTD ... 116
4.3. W3C XML Schema – WXS nebo XSD ... 127
4.3.1. Základní informace ... 127
4.3.2. Praktické použití XSD ... 128
4.3.3. Začátek XSD souboru ... 128
4.3.4. Výběr běžně použitelných základních typů ... 132
4.3.5. Jaké možnosti nám dávají základní typy ... 132
4.3.6. Možnosti restrikcí (*constraining facets*) ... 134
4.3.7. Složené (komplexní) datové typy ... 137
4.3.8. Atributy ... 140
4.3.9. Atribut je součástí koncového elementu ... 141
4.3.10. Prázdný element s atributy ... 141
4.3.11. Závěrečná definice kořenového elementu ... 142
4.3.12. Připojení schématu k dokumentu ... 142
4.3.13. Validace pomocí xerces ... 143
4.3.14. Jmenné prostory ... 143
4.3.15. Použití prázdné hodnoty ... 149
4.3.16. Práce s okrajovými bílými znaky ... 150
5. Java a XML, JAXB ... 153
5.1. Java a XML ... 153
5.1.1. Obecné vlastnosti parseru ... 153
5.1.2. Rozhraní parserů pro Javu ... 155
5.1.3. Základní dělení parserů ... 156
5.1.4. Přehled parserů ... 157

5.2. *Java Architecture for XML Binding* – základní informace ... 157
5.2.1. Podpora z Ant ... 158
5.3. Generování souborů z XSD ... 162
5.3.1. Princip bindingu ... 163
5.4. Čtení XML dokumentu ... 164
5.4.1. Výpočet celkové váhy ... 165
5.5. Čtení dokumentu včetně zpracování atributů ... 166
5.5.1. Výpočet celkové ceny ... 166
5.5.2. Všechny objekty v paměti ... 167
5.5.3. Problematické datové typy ... 167
5.6. Změna hodnot a vytvoření nových elementů ... 170
5.7. Zápis do XML dokumentu ... 170
5.8. Validace ... 173
5.8.1. Ukázka dvojí validace ... 174
5.9. Příprava kompletně nového dokumentu ... 175
5.9.1. Vše najednou ... 175
5.9.2. Příprava a následný zápis ... 177
6. XSLT a XPath ... 179
6.1. Zdroje ... 179
6.1.1. Tutoriály a další ... 179
6.1.2. Editory a další nástroje ... 179
6.2. Základní principy XSL ... 180
6.2.1. Důvody použití stylů ... 181
6.2.2. Dostupné XSLT procesory ... 182
6.2.3. Porovnání XSLT 2.0 a XQuery 1.0 ... 183
6.2.4. Základy XSLT stylů ... 183
6.3. XPath – dotazovací jazyk nad XML dokumentem ... 189
6.3.1. Abstraktní model dokumentu ... 189
6.3.2. Hierarchické vztahy mezi uzly – osy pohybu ... 193
6.3.3. Výrazy ... 195
6.3.4. Operátory ... 205
6.3.5. Funkce ... 206
6.3.6. Výčet všech funkcí a operátorů z XSLT 2.0 ... 208
7. XSLT 1 ... 217
7.1. Šablony ... 217
7.1.1. Režimy zpracování šablon ... 220
7.1.2. Priority šablon ... 221
7.1.3. Zabudované (implicitní) šablony ... 223
7.2. Tvorba výstupního dokumentu ... 225
7.2.1. Generování elementů a jejich obsahu ... 225
7.2.2. Generování hodnot atributů ... 228
7.2.3. Generování elementů s dynamickým jménem ... 229
7.2.4. Generování atributů s dynamickým jménem ... 230
7.2.5. Opakované vkládání skupiny atributů ... 231
7.2.6. Generování komentářů ... 233
7.2.7. Výstup vyhrazených znaků ... 233
7.2.8. Odstranění bílých znaků ze vstupního dokumentu ... 236
7.3. Formátování čísel ... 238
7.3.1. Formát pro automatické číslování ... 239
7.4. Podmíněné zpracování ... 241
7.4.1. Podmíněný příkaz ... 241
7.4.2. Podmíněný příkaz pomocí XPath výrazu ... 242
7.4.3. Podmíněné větvení ... 244

7.5. Iterativní zpracování ... 245
7.5.1. Ladicí výpisy pomocí cyklu ... 247
7.5.2. Cyklus pomocí posloupnosti ... 247
7.5.3. Cyklus pomocí posloupnosti řetězců a vnořený cyklus ... 248
7.5.4. Kdy vnořený cyklus nelze použít – změna kontextu ... 249
7.5.5. Cyklus pomocí XPath výrazu ... 252
7.5.6. Cyklus pomocí XPath výrazu a posloupnosti ... 254
7.6. Chybové zprávy ... 254
8. XSLT 2 ... 257
8.1. Proměnné a parametry ... 257
8.1.1. Proměnné ... 257
8.1.2. Parametry ... 264
8.1.3. Globální parametry ... 266
8.1.4. Typová kompatibilita proměnných a parametrů ... 267
8.2. Pojmenované šablony ... 268
8.2.1. Pojmenovaná šablona volaná z příkazové řádky ... 269
8.2.2. Pojmenovaná šablona a globální parametry ... 271
8.3. Definice vlastních funkcí ... 273
8.3.1. Funkce generující HTML bez jmenných prostorů ... 275
8.3.2. Funkce s více parametry ... 276
8.3.3. Složitější funkce ... 277
8.3.4. Funkce bez udaného typu návratové hodnoty nebo parametrů ... 278
9. XSLT 3 ... 281
9.1. Řazení uzlů ... 281
9.1.1. Řazení podle jednoho klíče ... 281
9.1.2. Řazení podle více klíčů ... 283
9.2. Seskupování uzlů ... 284
9.2.1. Základní použití ... 284
9.2.2. Řazení ve skupině ... 286
9.3. Styl uložený ve více souborech ... 287
9.3.1. Vložení stylu `<xsl:include>` ... 287
9.3.2. Import stylu `<xsl:import>` ... 289
9.3.3. Tipy pro návrh stylu ... 293
9.4. Práce s více soubory ... 293
9.4.1. Načítání více XML souborů ... 294
9.4.2. Generování více výstupních souborů ... 301
9.5. Regulární výrazy ... 305
9.6. Volání externích funkcí ... 308
9.6.1. Volání statických metod z JavaCore API ... 309
9.6.2. Volání instančních metod z JavaCore API ... 310
9.6.3. Volání metod vlastních Java tříd ... 312
9.6.4. Praktický příklad ... 314
9.7. Spouštění XSLT transformace z programu v Javě ... 318
9.7.1. Transformace s využitím JAXP a Java Core API Xalan ... 319
9.7.2. Transformace s využitím JAXP a Saxon ... 320
9.7.3. Přímé použití Saxon ... 322
10. SAX, StAX ... 325
10.1. SAX – *Simple API for XML* ... 325
10.1.1. Úvodní informace ... 325
10.1.2. Základní postup při zpracování ... 325
10.1.3. Zpracování parsovaného XML dokumentu ... 327
10.1.4. Zpracování složitějšího XML dokumentu ... 334
10.1.5. Problematika různého kódování ... 334

10.1.6. Nastavení vlastností parseru 335
10.1.7. Validace oproti DTD nebo XSD 336
10.1.8. Práce se jmennými prostory 339
10.2. Sun Java Streaming XML Parser (SJSXP) 340
10.2.1. Základní postup při zpracování 341
10.2.2. Přehled základních možností čtení 342
10.2.3. Zpracování atributů 343
10.2.4. Čtení na žádost 345
10.2.5. Práce se jmennými prostory 347
10.2.6. Zápis do XML dokumentu 349
10.2.7. Validace a transformace dokumentu 351
11. DOM – *Document Object Model* 354
11.1. Základní informace 354
11.2. Základní použití DOM 356
11.3. Zpracování parsovaného XML dokumentu 358
11.3.1. Výpočet celkové váhy 358
11.4. Metody rozhraní Node 359
11.4.1. Metody pro získání informace 359
11.4.2. Metoda pro pohyb nahoru (na rodiče) 360
11.4.3. Metody pro horizontální pohyb (na sourozence) 360
11.4.4. Metody pro pohyb dolů (na potomky) 360
11.4.5. Metody pro práci s atributy 360
11.5. Výpočet celkové ceny 361
11.6. Problém vkládaných elementů (odřádkování) 362
11.7. Práce se jmennými prostory 364
11.7.1. Jmenné prostory fakticky neuvažujeme 364
11.7.2. Se jmennými prostory pracujeme 365
11.8. Automatické odstranění komentářů 367
11.9. Všechny objekty v paměti 369
11.10. Průchod stromem dokumentu 370
11.11. Snadné odstranění „odřádkovávacích“ nodů 375
11.12. Zápis dokumentu 375
11.12.1. Ovlivnění práce transformační třídy 376
11.12.2. Ukázka zápisu do souboru 377
11.12.3. Problematika odřádkování a odsazování 379
11.13. Modifikace dokumentu 380
11.13.1. Změna hodnoty již existujícího elementu či atributu 380
11.13.2. Odstranění nodu 381
11.13.3. Vkládání nových nodů 381
11.13.4. Změna XML dokumentu a jeho zápis 382
11.14. Vytváření nového dokumentu 383
11.14.1. Klonování nodů 383
11.15. Validace nově vytvořeného nebo měněného dokumentu 385
12. Úvod do SVG 389
12.1. Úvodní informace 389
12.1.1. Struktura SVG dokumentu 390
12.2. Základní elementy SVG 392
12.2.1. Základní geometrické tvary (*basic shapes*) 392
12.2.2. Napojování a ukončování čar 394
12.2.3. Cesty (*path*) 396
12.2.4. Texty 399
12.2.5. Seskupování a společné nastavení vlastností 401
12.2.6. Transformace 401

12.2.7. Výplň – složitější možnosti ... 404
12.2.8. Animace ... 407
12.3. SVG a Java ... 410
12.3.1. Přímo použitelné nástroje z Batik ... 411
12.3.2. Programování pomocí knihoven Batiku ... 412
12.4. Generování SVG pomocí XSLT ... 413

Kapitola 1. XML – obecné informace

- hlavní informační zdroj/rozcestník v ČR – `www.kosek.cz`

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

- všespasitelná 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) označkovaný 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 označkovaného 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) „_ “(podtržítko) 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
  - datumy 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 uzavřená 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
  ```

```
<citat jazyk="čeština">
  Byli jsme a budem!
</citat>
<citat jazyk="angličtina">
  To be or not to be?
  <poznamkaPodCarou cislo="10">
    Shakespeare.
  </poznamkaPodCarou>
</citat>
```

Poznámka

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

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

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

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>
- předefinované 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

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

  - 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├í;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>Otylá</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>Otylá</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>Otylá</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 parsery
- 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\Ja-
va\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:bodu="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 nevadí, 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. Literatura

Kosek, J.: *XML pro každého*, GRADA, 2000

Marchal, B.: *XML v příkladech*, Computer Press, 2000

Harold, E.R. – Means, W.S.: *XML v kostce*, Computer Press, 2002

Kapitola 2. Java a národní prostředí, Ant

2.1. Java a národní prostředí

2.1.1. Úvodní informace

- 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 datumem a časem, měnou apod.
- používané zkratky:
  - i18n – i`nternationalizatio`n
    - psaní programu tak, aby se dal snadno přizpůsobit libovolnému národnímu prostředí
  - l10n – l`ocalizatio`n
    - převedení programu do dané jazykové mutace
- národní zváštnosti 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 a KIV/UUR

2.1.2. 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

2.1.2.1. 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 zkratek
  - 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, Ť, 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`

2.1.2.2. Značka bajtového pořadí

- UTF-x mohou pro identifikaci pořadí bajtů využívat počáteční **značku bajtového pořadí**
  - *initial byte order mark* (BOM), která je umístěna na samém začátku souboru

- pro tento účel definuje Unicode dva kódové body
  - `U+FEFF` = ZERO WIDTH NO-BREAK SPACE (*byte order mark*) (pevná mezera nulové délky)
  - `U+FFFE` = not a character code
- pro UTF-16 pro BE má tvar `FE FF` a pro LE pak `FF FE`
- je-li načtena správně, pak se pevná mezera nulové délky ze své podstaty nemůže zobrazit
  - při nesprávném načtení (záměna BE za LE nebo naopak) se opět nezobrazí, protože se jedná o neplatný znak
  - k nesprávnému načtení by ale principiálně nemělo dojít
- BOM se ale někdy nemusí nebo nesmí vyskytovat, což je dáno definicí charsetu

charset	BOM	hodnota	
UTF-8	volitelný	EF BB BF	
UTF-16	doporučený	FE FF nebo FF FE	
UTF-16LE	zakázaný	FF FE	na BOM se nebere zřetel, tj. i opačný FE FF musí být ignorován
UTF-16BE	zakázaný	FE FF	na BOM se nebere zřetel, tj. i opačný FF FE musí být ignorován
UTF-32	doporučený	00 00 FE FF nebo FF FE 00 00	
UTF-32LE	zakázaný	FF FE 00 00	na BOM se nebere zřetel
UTF-32BE	zakázaný	00 00 FE FF	na BOM se nebere zřetel

2.1.2.3. UTF-8

- bylo vytvořeno proto, aby se znaky Unicode daly zakódovat posloupností bajtů, protože s bajty umí pracovat každá aplikace a každý souborový systém
- vzniká v 1992
- staré a nepoužívané označení je UTF-2
- je to obecně rozšířený a přijímaný formát, popsaný např. v
  - RFC 2279
  - Amendment 2 v ISO 10646-1
  - Unicode Standard
  - např. řetězce v Javě v `.class` souborech jsou v UTF-8
- ze zmíněných sedmi charsetů Unicode se (v našich zeměpisných šířkách) často používá právě UTF-8
- výhody

- pro texty využívající jen znaky anglické abecedy je UTF-8 totožná s US-ASCII
  - využívá se jen jeden bajt na jeden znak
  - s US-ASCII umí pracovat každá aplikace
- pro akcentované znaky se využívají dva bajty

■ soubory kódované v UTF-16 zabírají pro většinu textů psaných latinkou (nejen angličtinou) zbytečně dvojnásobné místo

- zkoumáme-li četnost výskytu akcentovaných znaků v českém textu, zjistíme, že mají asi 10% výskyt

  „Například v tomto odstavci napsaném zcela prokazatelně češtinou s plným využitím akcentů je z celkových 129 písmen 114 písmen bez akcentů a jen 15 písmen s akcenty.“

- započítáme-li do předchozího příkladu i mezery, interpunkci a číslice (všechny jsou z US-ASCII) dostaneme celkově 164 znaků a z toho 15 akcentovaných

■ základní nevýhoda UTF-8 – znaky nemají stejnou délku, tzn. není možné skočit přímo na určitý znak („přeskoč prvních 20 znaků“)

- pravděpodobnost „omylu“ (považování poloviny znaku za celý znak) je omezena principem kódovacího schématu

znak Unicode	max. význ. bitů	bajty UTF-8
U+0000 až U+007F	7	`0xxx xxxx`
U+0080 až U+07FF	11	`110x xxxx 10yy yyyy`
U+0800 až U+FFFF	16	`1110 xxxx 10yy yyyy 10zz zzzz`
U+10000 až U+10FFFF	21	`1111 0xxx 10yy yyyy 10zz zzzz 10ss ssss`

■ princip čtení

- má-li bajt nastaveno MSB, pak počet jedničkových bitů za ním udává počet následujících bajtů znaku, které vždy začínají bity `10`
- „trefíme-li“ se náhodně doprostřed znaku, poznáme to podle začátku `10` a pak je nutné přeskočit všechny následující bajty začínající `10`

Poznámka

principiálně je možné zakódovat pomocí UTF-8 až 31 bitů

```
1111 110x 10yy yyyy 10zz zzzz 10ss ssss 10tt tttt 10uu uuuu
```

■ ale protože Unicode končí na významových 21 bitech, se tento způsob nevyužívá

■ nejširší znak v UTF-8 má tedy 4 bajty

■ u UTF-8 je z principu zbytečná značka bajtového pořadí (BOM)

- specifikace říká, že není ani vyžadována, ani doporučována, ale pokud je použita, nesmí vadit

- mnoho aplikací ale BOM vyžaduje a vytváří a bez BOM pracují chybně, např. nejsou schopné automaticky rozpoznat charset
- některé (např. SciTe) dokonce považují BOM za nezbytnou část a označení „UTF-8“ znamená „UTF-8 + BOM“ a pro UTF-8 bez BOM používají označení „UTF-8 Cookie“

■ praktický příklad

znak	Unicode	Unicode bitově	UTF-8 bitově	UTF-8 bajtově
D	U+0044	`0000 0000 0100 0100`	`0100 0100`	44
á	U+00E1	`0000 0000 1110 0001`	`1100 0011 1010 0001`	C3 A1
š	U+0161	`0000 0001 0110 0001`	`1100 0101 1010 0001`	C5 A1
a	U+0061	`0000 0000 0110 0001`	`0110 0001`	61
BOM	U+FEFF	`1111 1110 1111 1111`	`1110 1111 1011 1011 1011 1111`	EF BB BF
	U+FFFE	`1111 1111 1111 1110`	`1110 1111 1011 1111 1011 1110`	EF BF BE

■ jediná správná značka bajtového pořadí je v UTF-8 `EF BB BF`

- posloupnost `EF BF BE` vznikla zakódováním neexistujícího znaku, není tedy BOM a žádná aplikace ji nerozpozná

■ slovo „Dáša“ může tedy v UTF-8 vypadat

a. `44 C3 A1 C5 A1 61`

b. `EF BB BF 44 C3 A1 C5 A1 61`

ale nikdy jako

c. `EF BF BE 44 C3 A1 C5 A1 61`

Poznámka

Existuje též kódovací schéma UTF-7, které využívá pouze 7 bitů bajtu

■ je popsáno v RFC-2152

■ prakticky by se použilo, pokud by (zastaralý) přenosový kanál dovoloval přenášet jen sedmibitové bajty

■ slovo „Dáša“ v UTF-7 je: `44 2B 41 4F 45 42 59 51 2D 61`

což znamená, že každý akcentovaný znak využívá čtyři bajty

2.1.3. 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
pýknř└eskř└ýtaŘ =1

D:\zzz>type CestinaIdentifikatory.java
public class CestinaIdentifikatory {
  public static void main(String[] args) {
    int pýknř└eskř└ýtaŘ = 1;
    System.out.println("pýknř└eskř└ýtaŘ ="
                        + pýknř└eskř└ýtaŘ);
  }
}

D:\zzz>_
```

■ v `.class` souboru je použito `UTF-8`

- `.class` soubory jsou plně přenositelné na jiné platformy

2.1.3.1. 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─Ťkn├Ż─îesk├Ż─î├şta─Ź = 1;
         ^
UTF8.java:3: illegal character: \733
    int p─Ťkn├Ż─îesk├Ż─î├şta─Ź = 1;
             ^
UTF8.java:3: not a statement
    int p─Ťkn├Ż─îesk├Ż─î├şta─Ź = 1;
          ^
```

- možným – ale neelegantním – řešením je promocí externího programu pro změnu kódování převést soubor do implicitního kódování

2.1.3.1.1. 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
pýknř╚eskř╚ýtaŘ =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

2.1.3.1.2. Řešení prostředky JDK – `native2ascii`

- zdrojové soubory v Javě nemají možnost (narozdíl od souborů XML) defifovat 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\u00edta\u010d = 1;
    System.out.println("p\u011bkn\u00fd\u010cesk\u00fd\u010c\u00edta\u010d ="
                      + p\u011bkn\u00fd\u010cesk\u00fd\u010c\u00edta\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
pýknř└eskř└ýtaŘ =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>_
```

2.1.4. 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 iplicitním 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("Implicitni kodovani konzole: "
                  + oswDef.getEncoding());

    /* IBM852 je výstupní kódování češtiny v DOSovém okénku */
    OutputStreamWriter osw =
                  new OutputStreamWriter(System.out, "IBM852");
    System.out.println("Nastavene kodovani konzole:  "
                  + osw.getEncoding());

    PrintWriter p = new PrintWriter(osw);
    p.print("Příšerně žluťoučký kůň úpěl ďábelské ódy.\n");
    p.print("áčďéěíňóřšťúůýž\n");
    p.print("PŘÍŠERNĚ ŽLUŤOUČKÝ KŮŇ ÚPĚL ĎÁBELSKÉ ÓDY.\n");
    p.print("ÁČĎÉĚÍŇÓŘŠŤÚŮÝŽ\n");
    p.flush();
  }
}
```

překlad a spuštění

```
D:\zzz>javac NaKonzoli.java

D:\zzz>java NaKonzoli
Implicitni kodovani konzole: Cp1250
Nastavene kodovani konzole:  Cp852
Příšerně žluťoučký kůň úpěl ďábelské ódy.
áčďéěíňóřšťúůýž
PŘÍŠERNĚ ŽLUŤOUČKÝ KŮŇ ÚPĚL ĎÁBELSKÉ ÓDY.
ÁČĎÉĚÍŇÓŘŠŤÚŮÝŽ

D:\zzz>
```

- používají se zde „stará“ označení kódování (`Cp1250`, `Cp852`)

2.1.4.1. 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>_
```

2.1.5. 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 opíše 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
ď»żpublic class UTF8 {
  public static void main(String[] args) {
    int pĂ?knĂ˝ÄŚeskĂ˝ÄŚĂ-taÄŤ = 1;
    System.out.println("pĂ?knĂ˝ÄŚeskĂ˝ÄŚĂ-taÄŤ ="
                     + pĂ?knĂ˝ÄŚeskĂ˝ÄŚĂ-taÄŤ);
  }
}


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 žluťoučký kůň úpěl ďábelské ódy";
    String akcenty = "áčďéěíňóřšťúůýž";
    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 žluťoučký kůň úpěl ďábelské ódy
áčďéěíňóřšťúůýž
PŘÍŠERNĚ ŽLUŤOUČKÝ KŮŇ ÚPĚL ĎÁBELSKÉ ÓDY
ÁČĎÉĚÍŇÓŘŠŤÚŮÝŽ

D:\zzz>
```

2.1.6. 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 = "áčďéěíňóřšťúůýž";
    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
```

2.1.7. 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());
  }
}
```

vypíše 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 zkratek 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

2.1.8. Řazení řetězců

2.1.8.1. 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 "f" 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

2.1.8.2. 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 řadicí 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 řadicí 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
```

2.2. Ant – *Another Neat Tool*

2.2.1. 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)

2.2.2. Jak Ant získat

- `ant.apache.org` jako jeden `.zip` soubor, který se pouze rozbalí do `C:\Program Files\Java\ant` (leden 2009 verze 1.7.1.)
  - 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`

2.2.3. 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>
```

2.2.3.1. 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

2.2.4. 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>
```

2.2.4.1. 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
```

2.2.4.2. 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
```

2.2.4.3. 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>
```

2.2.5. 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>`

2.2.6. 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>`

2.2.7. 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" />
```

2.2.8. Možnosti `<target>`

2.2.8.1. Seznam `<target>`

- máme-li více `<target>`, můžeme si jejich seznam vypsat použitím příkazu:

```
-projecthelp
```

```
D:\xml\ant>ant -buildfile zavislosti.xml -projecthelp
Buildfile: zavislosti.xml

Main targets:

Other targets:
 druhy
 prvni
 treti
Default target: prvni
```

2.2.8.2. Help target

- mnohom jednodušší, než vypisovat targety pomocí předchozího příkladu je, když default target (často pojmenovaný Help) bude zajišťovat pouze výpis návodu k použití
  - po spuštění Antu (předpokládáme implicitní jméno `build.xml`) je pak tento návod vypsán

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Help" default="help">
  <target name="help">
    <echo message="Ukazka vyhodneho nastaveni default targetu"/>
    <echo message="  k dispozici jsou targety:"/>
    <echo message="    preklad  - preklada zdrojovy kod"/>
    <echo message="    spusteni - spousti vytvoreny program"/>
    <echo message="    mazani   - maze nepotrebne soubory"/>
  </target>

  <target name="preklad">
    <echo message="Prekladam"/>
  </target>

  <target name="spusteni">
    <echo message="Spoustim"/>
  </target>

  <target name="mazani">
    <echo message="Mazu"/>
  </target>
</project>
```

vypíše:

```
D:\xml\ant>ant
Buildfile: build.xml

help:
     [echo] Ukazka vyhodneho nastaveni default targetu
     [echo]   k dispozici jsou targety:
     [echo]     preklad  - preklada zdrojovy kod
     [echo]     spusteni - spousti vytvoreny program
     [echo]     mazani   - maze nepotrebne soubory
```

```
BUILD SUCCESSFUL
```

spuštění jednotlivých targetů je pak:

```
D:\xml\ant>ant preklad
Buildfile: build.xml

preklad:
     [echo] Prekladam

BUILD SUCCESSFUL
```

2.2.8.3. Vzájemné závislosti

- jednotlivé `<target>` mohou záviset pořadím provádění na jiných `<target>`
  - závislosti se mohou řetězit
- 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] treti

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
```

2.2.8.4. 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>
```

- vypíše:

```
D:\xml\ant>ant -buildfile if-unless3.xml
Buildfile: if-unless3.xml

druhy:

prvni:
     [echo] prvni = defaultni

BUILD SUCCESSFUL
```

2.2.9. 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.
...
```

2.2.10. Úkoly (tasks)

- jsou to jednotlivé příkazy v rámci jednoho `<target>`

Jsou tří 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

Visual Age for Java Tasks

2.2.11. Přehled a použití často používaných úkolů

2.2.11.1. `<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>
```

2.2.11.2. `<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>
```

2.2.11.3. <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

2.2.11.4. <move>

- stejné použití jako u `<copy>`, pouze soubory přesune

2.2.11.5. <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="." />
<!--       <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
```

2.2.11.6. `<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
```

2.2.11.7. `<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="./jiny-src"
             destdir="./doc"
             windowtitle="Projekt JXT"
             encoding="utf-8"
             docencoding="utf-8"
             charset="utf-8"
             author="true"
             private="true">
    </javadoc>
  </target>
</project>
```

- vypíše:

```
D:\xml\ant>ant -buildfile javadoc.xml
Buildfile: javadoc.xml

javadoc:
  [javadoc] Generating Javadoc
  [javadoc] Javadoc execution
  [javadoc] Creating destination directory: "D:\xml\ant\doc\"
  [javadoc] Loading source files for package jxt...
  [javadoc] Loading source files for package jxt.aplikacni...
  [javadoc] Loading source files for package jxt.datova...
  [javadoc] Constructing Javadoc information...
  [javadoc] Standard Doclet version 1.6.0_03
  [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
```

2.2.11.8. `<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>
```

2.2.11.9. 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
```

2.2.11.10. `<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>
```

2.2.12. 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`
  - nepříliš 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>
```

2.2.13. 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}">
  </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>
```

2.2.14. Doporučeno k přečtení:

ant\docs\ant_in_anger.html

Kapitola 3. Arrays, řazení, kolekce a genericita

3.1. Podpora práce s poli – třída Arrays

- pro ukládání více objektů stejného typu do paměti se používají datové struktury nazývané obecně **kontejnery**
  - kontejner má široký význam – označuje jakýkoliv objekt, který může uschovávat jiné objekty
    - **statické kontejnery** – jejich velikost je určena při jejich vzniku a je dále neměnná
      - návrhový vzor Přepravka
      - pole, podporované třídou `Arrays`
        - výhody – rychlost, primitivní datové prvky i objekty
        - nevýhody – pevná velikost, malá podpora přídavnými funkcemi, menší úroveň bezpečnosti (synchronizace apod.)
    - **dynamické kontejnery** – jejich velikost je přizpůsobena okamžitým potřebám
      - kolekce = třídy z balíku *Collection Framework* – `java.util`
        - výhody a nevýhody víceméně opačné
        - používají se více než pole
        - viz dále

3.1.1. 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 3.1. 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]);
  }
}
```

vypíše

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

3.2. Řazení objektů

Poznámka

Tyto informace jsou mj. nutné pro pochopení jednoho typu implementace kolekcí. Dále proto, že operace řazení se v kolekcích používá poměrně často.

- pro pole primitivních datových prvků jsou `sort()` a `binarySearch()` jednoduše použitelné
- totéž platí pro objekty knihovních 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!

3.2.1. 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 `int compareTo(Typ t)`
    - `compareTo()` vrací `int` s hodnotou 0, pokud jsou objekty stejné, zápornou, pokud je parametr `t` menší, a kladnou, pokud je parametr `t` 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 3.2. 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 ArraysPrirozeneRazeniObecnychObjektu {
  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, zena
[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

3.2.2. 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`
  - využívá se návrhový vzor **Příkaz** (*Command*)
    - tento NV zabalí metodu do objektu, což umožňuje dynamickou výměnu používaných metod za běhu programu
    - objekt je často typován na rozhraní, které má (nejčastěji) jednu metodu
    - reference na tento objekt se pak předává jako skutečný parametr nějaké metody, pro kterou metoda v Příkazu provádí pomocnou službu
- při řazení máme absolutní kontrolu nad procesem řazení, bez ohledu na případné přednastavení řazených objektů vyjádřené metodou `compareTo()` z přirozeného řazení

`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

Napíšeme-li metodu `compare()` nebo `compareTo()` podle zadání, bude se řadit vzestupně. Chceme-li sestupné řazení, **nikdy** to neřešíme obrácením znaménka návratové hodnoty! Pro `compareTo()` lze použít `reverseOrder()` z `java.util.Collections`.

Příklad 3.3. 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();
  }
}
```

vypíše:

```
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
```

3.2.2.1. 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

3.3. Kolekce a genericita – úvodní informace

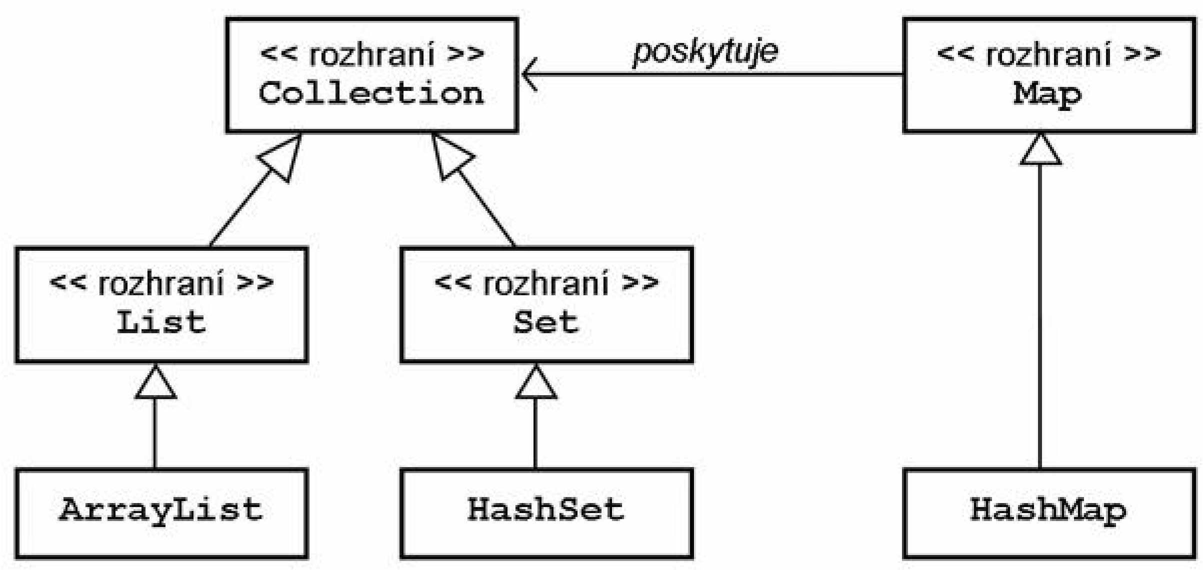
- objekty tříd z balíku `java.util` – *Java Collections Framework*
- slouží k uschovávání většího předem neznámého množství objektů libovolného typu

- „kolekce“ – dvě nejužívanější implementace – `ArrayList` a `HashSet` implementují rozhraní `java.util.Collection` (neplést si se třídou `java.util.Collections`)
- třetí významná implementace `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 kolekcí velmi rozsáhlá a obsahuje ve své úplnosti (v JDK 1.6) 13 rozhraní, 8 abstraktních tříd a 16 tříd
  - na knihovnu se lze dívat ze tří pohledů:
    - **rozhraní** – abstraktní datové typy (např. `Set` = množina), které můžeme použít
    - **implementace** – konkrétní implementace rozhraní, kterou použijeme dle požadovaného účelu (např. `HashSet` nebo `TreeSet`)
    - **algoritmy** – metody poskytující služby nad kolekcemi, např. `sort()`, `reverse()`, většinou shodně pojmenované pro více kolekcí
- pokud ale nevyžadujeme žádné speciality a nezáleží nám (zpočátku) na maximální možné efektivnosti programu, potřebujeme pouze čtyři rozhraní a tři implementační 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`ů

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) – podrobnosti viz dále
- zápis dříve (`TypovanaKolekce.java`):

```
ArrayList arNetypovany = new ArrayList();
arNetypovany.add(new Integer(1));
arNetypovany.add(new Integer(2));
arNetypovany.add("tri");
for (Object objekt: arNetypovany) {
  Integer i = (Integer) objekt;
  System.out.print(i.intValue() + ", ");
}
```

zápis od JDK 1.5:

```
ArrayList<Integer> arTypovany = new ArrayList<Integer>();
arTypovany.add(new Integer(1));
arTypovany.add(new Integer(2));
arTypovany.add("tri");     // chyba při kompilaci
for (Integer cislo : arTypovany) {
  System.out.print(cislo.intValue() + ", ");
}
```

Poznámka

Další vylepšení z JDK 1.5 viz později.

Poznámka

Konstrukce `for()` pro procházení přes všechny prvky kolekce viz též [skr-97].

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.

Poznámka

Nemáme-li skutečně vážný důvod, používáme referenční proměnnou typu rozhraní, nikoliv typu implementace. Zdůvodnění viz dříve. Toto typování je nutností ve formálních parametrech našich metod, protože metody z kolekcí vracejí zásadně typ rozhraní, nikoliv typ implementace – viz později u `CeleCislo`. Další výhoda – v budoucnu je možná snadná změna implementace.

```
List<Integer> seznam = new ArrayList<Integer>();
//    List<Integer> seznam = new LinkedList<Integer>();
seznam.add(new Integer(1));
seznam.add(new Integer(2));
for (Integer cislo : seznam) {
  System.out.print(cislo.intValue() + ", ");
}
```

Pokud typujeme na co hierarchicky nejvyšší rozhraní, je tímto způsobem možná i změna typu kolekce. Pouze se vytvoří jiná implementace kolekce (ale vkládání a výběr prvků a průchod kolekcí zůstanou naprosto nezměněny, protože tyto služby jsou popsány kontraktem rozhraní).

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

nebo později:

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

Příklad 3.4.

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

```
import java.util.*;

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

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

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

Vypíše:

```
List: [prvni, druhy, prvni]
Set:  [prvni, druhy]
Map:  {prvni=pivo, druhy=objekt}
```

3.4. Typové parametry a parametrizované typy

- podobně jako metody mají své (formální) parametry, mohou mít parametry i objektové datové typy (rozhraní a třídy)
  - tyto datové typy se pak nazývají **parametrizované typy** nebo synonymně **generické typy**
  - jsou zavedeny od JDK 1.5
- parametry třídy (rozhraní) stanovují typy objektů, které budou dále ve třídě využívány
  - označují se jako **typové parametry**
  - třída se uvedením typového parametru specializuje jen (zejména) na práci s tímto typem
    - překladač může provádět mnohem více kontrol
      - rozhoduje o přípustnosti skutečně použitého typu

- ♦ v přeloženém kódu již tyto typové informace nejsou
- jako typové parametry je možno použít pouze objektové typy (třídy nebo rozhraní) – nikoliv primitivní typy

■ typové parametry se uvádí ve špičatých závorkách za názvem třídy – např. `List<String>`

- v Java Core API je při deklaraci parametrizovaného typu je typový parametr často značen jako `<E>` („element“), např.:

```
public interface List<E>
```

- při použití je pak `E` nahrazeno skutečnou třídou nebo rozhraním – definujeme skutečný typ objektů

```
List<String> seznamRetezcu;
List<Integer> seznamCelychCisel;
List<Osoba> seznamOsob;
```

■ použití parametrizovaných typů je typické zejména v kolekcích, ale vyskytují se i jinde

- např. rozhraní `java.lang.Comparable<Typ>`

■ můžeme vytvářet svoje parametrizované typy

- v začátcích používání OOP asi nevyužijeme, protože všechny potřebné typy jsou již hotové
- `E` je použito jako typ `<E>`, jako formální parametr metody `vloz(E prvek)` i jako návratová hodnota `E vyber()`

```
public class Zasobnik<E> {
  List<E> prvky = new ArrayList<E>();

  public void vloz(E prvek) {
    prvky.add(0, prvek);
  }

  public boolean isPrazdny() {
    return (prvky.size() == 0);
  }

  public E vyber() {
    E vybirany = prvky.get(0);
    prvky.remove(0);
    return vybirany;
  }
}
```

- použití:

```
Zasobnik<Hruska> zas1;
Zasobnik<Integer> zas2;
```

- občas potřebujeme, aby typové parametry vyhovovaly daným omezením – např. aby implementovaly nějaké rozhraní

```
public class PorovnavaciZasobnik<E extends Comparable<E>> {
```

  - zde se i pro implementaci rozhraní používá klíčové slovo `extends`
  - do tohoto zásobníku by šly ukládat jen třídy, které implementují rozhraní `Comparable`, např.:

```
PorovnavaciZasobnik<Hruska> zas1;    // nelze
PorovnavaciZasobnik<Integer> zas2;
```

    - pro třídu `Hruska` hlásí překladač:

```
Bound mismatch: The type Hruska is not a valid substitute
for the bounded parameter <E extends Comparable<E>>
of the type PorovnavaciZasobnik<E>
```

3.4.1. Použití žolíků – *unbounded wildcard*

- slouží k řešení problémů způsobených omezeními dědičnosti parametrizovaných typů ve významu:
  - jakákoliv třída: `<?>`
  - třída implementující rozhraní `R`, resp. potomek `R`: `<? extends R>` – viz dále
- typový parametr `<?>` se používá pouze v deklaracích tříd a metod, např.:
  - v API: `boolean removeAll(Collection<?> c)`
- nelze použít pro deklaraci: `List<?> seznam = new ArrayList<?>();`
- v našich programech zásadně nepoužívat – ztrácíme výhody typování

Příklad 3.5.

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

  public static void tisk(List<?> seznam) {
    for (Object o : seznam) {
      if (o instanceof String) {
        System.out.println(o.toString());
      }
      if (o instanceof Integer) {
        System.out.println(((Integer) o).intValue());
      }
    }
  }
}
```

3.4.2. Omezené využití žolíků – *bounded wildcard*

- stejně jako běžnou třídu lze omezit i použití žolíku
  - třída implementující rozhraní `R`, resp. potomek `R`: `<? extends R>`
- používá se i ve formálních parametrech metod
  - v API: `boolean addAll(Collection<? extends T> c)`
- nelze použít pro deklaraci:

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

- jako bázový typ `T` lze samozřejmě použít i rozhraní
- v začátcích programování nepoužíváme

Příklad 3.6.

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

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

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

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

  public static void tisk(List<? extends Tisknutelny> seznam) {
    for (Tisknutelny tisknutelny : seznam) {
      tisknutelny.tiskni();
    }
  }
}
```

vypíše:

```
A
B potomek A
```

3.5. Rozhraní `Collection`

- základ seznamů a množin, definuje množství metod
  - formální parametr `E` má význam „typově libovolný **element** kolekce"
    - jak je to v knihovně kolekcí implementačně zařízeno, nás nemusí zajímat

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()`

3.6. 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

3.6.1. Implementace pomocí ArrayList

- nejpoužívanější implementace seznamu, na kterou přecházíme, přestávají-li nám stačit klasická pole
- kromě své velikosti, tj. aktuálního počtu prvků vraceného metodou size() má i kapacitu – důležité pro efektivitu implementace

Příklad 3.7.

```
import java.util.*;

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

  public static void main(String[] args) {
    List<String> sezn1 = new ArrayList<String>();
    System.out.println("sezn1 je prazdny: " + sezn1.isEmpty());
    sezn1.add("prvni");
    sezn1.add("druhy");
    sezn1.add("prvni");
    tiskni("sezn1", sezn1);

    System.out.println("\nPridavani a ubirani prvku");
    List<String> sezn2 = new ArrayList<String>(sezn1);
    sezn2.add("treti");
    tiskni("sezn2", sezn2);
    sezn2.remove("prvni");
    tiskni("sezn2", sezn2);
    sezn2.removeAll(sezn1);
    tiskni("sezn2", sezn2);
    sezn2.addAll(sezn1);
    tiskni("sezn2", sezn2);
    sezn2.retainAll(sezn1);
    tiskni("sezn2", sezn2);

    System.out.println("\nHledani prvku");
    List<String> sezn3 = new ArrayList<String>(sezn1);
    sezn3.add("ctvrty");
    System.out.println("sezn3 obsahuje 'paty': "
                     + sezn3.contains("paty"));
    System.out.println("sezn3 obsahuje sezn1: "
                     + sezn3.containsAll(sezn1));

    System.out.println("\nPrevod na pole sezn3");
    String[] poleRetezcu = (String[]) sezn3.toArray(new String[0]);
    System.out.println(Arrays.asList(poleRetezcu));
  }
}
```

Vypíše:

```
sezn1 je prazdny: true
sezn1 (3) : [0]=prvni, [1]=druhy, [2]=prvni,
```

```
Pridavani a ubirani prvku
sezn2 (4) : [0]=prvni, [1]=druhy, [2]=prvni, [3]=treti,
sezn2 (3) : [0]=druhy, [1]=prvni, [2]=treti,
sezn2 (1) : [0]=treti,
sezn2 (4) : [0]=treti, [1]=prvni, [2]=druhy, [3]=prvni,
sezn2 (3) : [0]=prvni, [1]=druhy, [2]=prvni,

Hledani prvku
sezn3 obsahuje 'paty': false
sezn3 obsahuje sezn1: true

Prevod na pole sezn3
[prvni, druhy, prvni, ctvrty]
```

Příklad 3.8.

Ve třídě `CeleCislo` je také překrytá metoda `toString()`, díky níž můžeme tisknout obsah celého seznamu najednou. Formálním parametrem metody `tiskni()` je objekt třídy `List`. To je nutné proto, že metoda `subList()` vrací rozhraní `List` nikoli implementaci `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). To je důvod, proč v kolekcích používáme téměř výhradně neměnné (*immutable*) objekty.

```
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 {
  public 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");
    List<CeleCislo> celySeznam = new ArrayList<CeleCislo>();
    for (int i = 0;  i < 5;  i++) {
      celySeznam.add(new CeleCislo(i + 10));
    }
    tiskni("celySeznam", celySeznam);
    System.out.println("Tisk celeho seznamu: " + celySeznam);

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

    celySeznam.get(3).setCislo(33);
```

```
    tiskni("celySeznam", celySeznam);
    tiskni("podSeznam ", podSeznam);
  }
}
```

Vypíše:

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

3.7. Zajištění algoritmů – třída `Collections`

- pro běžné pole existuje třída `Arrays` se svými statickými metodami, které slouží pro realizaci algoritmů nad poli – 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 – **přirozené řazení** (`compareTo()` patřící třídě řazených objektů) nebo **absolutní 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 3.9.

```
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);
  }
}
```

Vypíše 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 = 105, va = 26.1, dite,
vy = 172, va = 63.0, zena,
vy = 116, va = 80.5, obezni trpaslik,
vy = 186, va = 82.5, muz]
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]
```

3.8. 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. `List`) 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>();
  ```

3.8.1. For-Each

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

Příklad 3.10.

```
List<Integer> seznam = new ArrayList<Integer>();
seznam.add(new Integer(1));
seznam.add(new Integer(2));
for (Integer prvek: seznam) {
  System.out.print(prvek.intValue() + ", ");
}

System.out.println("\nBezne pole");
int[] pole = {5, 6, 7, 8, 9};
for (int hodnota : pole) {
  System.out.print(hodnota + ", ");
}
```

Vypíše:

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

3.8.2. Iterátory

- jedná se o návrhový vzor **Iterátor** (*Iterator*)
  - zprostředkuje jednoduchý a sekvenční přístup k objektům uloženým v nějaké složitější datové struktuře, přičemž implementace této struktury je uživateli skryta
- iterátory mají v Javě silnou podporu – jsou odvozeny od rozhraní `java.util.Iterator<Typ>` (nepoužívat „starý“ `Enumeration`)
  - dají se použít pro všechny třídy, které implementují rozhraní `java.lang.Iterable` – to jsou všechny třídy kolekcí a mnohé další
- objekty, které toto rozhraní implementují, vrací rozhraní kolekcí metodou `Iterator<Typ> iterator()`
- v porovnání s *For-Each* používáme jen ve speciálních případech
  - nejčastěji přeskakování (vynechání) některých prvků nebo odstranění prvků z kolekce během průchodu kolekcí
- 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 jednoprůchodový – po průchodu pozbývá funkčnost. Pro případný další průchod se musí znovu vygenerovat.

- 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 rozhraní (typy) kolekcí.

Příklad 3.11.

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) {
    List<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,
```

3.8.2.1. Změna kolekce při použití iterátoru

- během iterátor používání iterátoru se nesmí změnit procházená kolekce
  - v opačném případě je vyhozena výjimka `ConcurrentModificationException`
- jedinou možnou změnou kolekce je odstranění prvku, ale ne metodou kolekce, ale metodou `remove()` iterátoru
- je také třeba si uvědomit, že každé volání `next()` posunuje iterátor na další prvek kolekce
  - potřebujeme-li tedy s prvkem získaným pomocí `next()` pracovat opakovaně, je nutné vytvořit pomocnou referenční proměnnou

```
public class IteratorProblemy {
  public static void main(String[] args) {
    List<Hruska> kosHrusek = new ArrayList<Hruska>();
    for (int i = 0;  i < 11;  i++) {
      kosHrusek.add(new Hruska(i + 20));
    }

    for (Iterator<Hruska> it = kosHrusek.iterator();
                                  it.hasNext(); ) {
      Hruska h = it.next();
      System.out.print(h + ", ");
      h.tisk();
    }
//    it.next();    // neni pristupny - nelze udelat chybu
    System.out.println();

    Iterator<Hruska> it1 = kosHrusek.iterator();
//    Iterator<Hruska> it2 = kosHrusek.iterator();
    it1.next();
    it1.next();
    while (it1.hasNext()) {
      it1.next().tisk();
      it1.remove();
    }

    System.out.println("\nPo ubrani hrusek");
    Iterator<Hruska> it2 = kosHrusek.iterator();
    while (it2.hasNext()) {
      it2.next().tisk();
    }
    System.out.println();
  }
}
```

vypíše:

```
20, 20, 21, 21, 22, 22, 23, 23, 24, 24, 25, 25, 26, 26, 27, 27, 28, 28, 29, ►
29, 30, 30,
22, 23, 24, 25, 26, 27, 28, 29, 30,
Po ubrani hrusek
20, 21,
```

3.9. 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.

3.10. 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 3.12.

Výjimku vyvolá i akce s kolekcí v jednom a témž procesu. Není tedy pravda, že pro vyhození této výjimky musí být do kolekce souběžný přístup z více procesů.

```
import java.util.*;
public class IteratorZmenaKolekce {
  public static void main(String[] args) {
    List<Integer> kolekce = new ArrayList<Integer>();
    for (int i = 0;  i < 10;  i++) {
      kolekce.add(new Integer(i));
    }

//    Iterator<Integer> it = kolekce.iterator();
//    System.out.println(it.next());
//    kolekce.add(new Integer(20));
//    // zde vyhodi vyjimku
//    System.out.println(it.next());

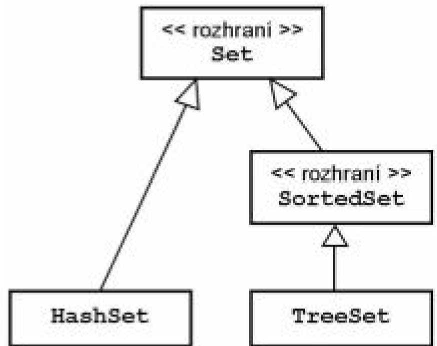
    for (Integer i : kolekce) {
      System.out.println(i);
      kolekce.add(new Integer(20));
    }
  }
}
```

Vypíše:

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

3.11. 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 v závislosti na použité implementaci definovánu určitou metodu zajišťující test shody
- pro implementaci se používá nejčastěji třída `java.util.HashSet`
  - unikátnost je zajištěna tím, že každý objekt vkládaný do množiny musí mít definovány metody `equals()` a `hashCode()`
  - výhodou neseřazené množiny oproti neseřazenému seznamu je větší rychlost vyhledání prvku (O1)
  - nevýhodou oproti seznamu je, že množina nezaručuje, že budou vložené prvky uchovávány v nějakém definovaném pořadí
- druhá konkurenční implementace je pomocí třídy `java.util.TreeSet` která implementuje `SortedSet`



- v `TreeSet` se průběžně udržují prvky seřazené (využívá stromovou strukturu)
  - unikátnost je zajištěna dvěma způsoby:
    1. každý objekt vkládaný do množiny musí implementovat rozhraní `Comparable` – přirozené řazení
    2. třídě `TreeSet` je v konstruktoru předán odkaz na existující `Comparator` – absolutní řazení
  - již v době vložení se vkládaný prvek zařadí do odpovídajícího pořadí vzhledem ke stávajícím prvkům
  - vkládání do `TreeSet` (i další operace) je pomalejší než do `HashSet`
    - výhoda `TreeSet` je v možnosti snadno získat nejmenší či největší prvek, případně podmnožinu původní kolekce
- protože `Set` je stejně jako `List` zděděno od `Collection`, obsahuje stejné metody, které již byly popsány u `Collection`
  - cokoliv, 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 3.13.

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> mnozina) {
    System.out.println(mnozina.getClass().getSimpleName());
    mnozina.add("treti");
    mnozina.add("druhy");
    mnozina.add("prvni");
    // pokus o vlozeni stejneho prvku
    if (mnozina.add("treti") == false) {
      System.out.println("'treti' podruhe nevlozen");
    }
    System.out.println(mnozina.size() + " " + mnozina);
    for (String s: mnozina) {
      System.out.print(s + ", ");
    }
    if (mnozina.contains("treti") == true) {
      System.out.println("\n'treti' je v mnozine");
    }
    mnozina.remove("treti");
    System.out.println(mnozina);
    mnozina.clear();
  }

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

Vypíše:

```
HashSet
'treti' podruhe nevlozen
3 [prvni, treti, druhy]
prvni, treti, druhy,
'treti' je v mnozine
[prvni, druhy]
TreeSet
'treti' podruhe nevlozen
3 [druhy, prvni, treti]
druhy, prvni, treti,
'treti' je v mnozine
[druhy, prvni]
```

3.11.1. Práce s vlastní třídou v množině

- předchozí příklad využíval skutečnosti, že třída `String` (stejně jako obalovací třídy) má správně naprogramované metody `compareTo()` (pro `TreeSet`) a/nebo `equals()` a `hashCode()` (pro `HashSet`)

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

Příklad 3.14.

Do množiny budeme ukládat již známý typ `Hruska`. Metoda `hashCode()` je zde jednoduchá. Využíváme přirozeného řazení, takže `Hruska` implementuje `Comparable`. 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` by došlo pouze k přetížení a program by pracoval chybně. Anotace `@Override` nás na tuto chybu upozorní.

Třída `Pocitadlo` umožňuje počítat, kolikrát byly volány metody `hashCode()`, `equals()` a `compareTo()`.

```
package kolekce;
import java.util.*;

class Pocitadlo {
  public static int e = 0;
  public static int h = 0;
  public static int c = 0;
}

class Hruska implements Comparable<Hruska> {
  private int cena;

  Hruska(int cena) { this.cena = cena; }

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

  @Override
//  public boolean equals(Hruska o) {  // chyba
  public boolean equals(Object o) {
    Pocitadlo.e++;
    System.out.print("" + Pocitadlo.e + "e ");

    if (o == this) {
      return true;
    }
    if (o instanceof Hruska == false) {
      return false;
    }
    boolean stejnaCena = (cena == ((Hruska) o).cena);
    return stejnaCena;
  }

  @Override
  public int hashCode() {
    Pocitadlo.h++;
    System.out.print("" + Pocitadlo.h + "h ");

    return cena;
  }

  public int compareTo(Hruska h) {
    Pocitadlo.c++;
    System.out.print("" + Pocitadlo.c + "c ");
```

```
    int cenaPom = h.cena;
    if (this.cena > cenaPom)
      return (+1);
    else if (this.cena == cenaPom)
      return (0);
    else
      return (-1);
  }
}

public class HruskyVMnozine {
  static void praceSHruskami(Set<Hruska> mnozina) {
    System.out.println("\n" + mnozina.getClass().getSimpleName());
    for (int i = 30;  i < 40;  i++) {
      mnozina.add(new Hruska(i));
    }
    mnozina.add(new Hruska(35));

    System.out.print("\npocet: " + mnozina.size() + " [");
    for (Hruska h: mnozina) {
      System.out.print(h + ", ");
    }
    System.out.println("]");
  }

  public static void main(String[] args) {
    praceSHruskami(new HashSet<Hruska>());
    System.out.println("e=" + Pocitadlo.e + ", h=" + Pocitadlo.h
                        + ", c=" + Pocitadlo.c);

    Pocitadlo.e = 0;
    Pocitadlo.h = 0;
    Pocitadlo.c = 0;
    praceSHruskami(new TreeSet<Hruska>());
    System.out.println("e=" + Pocitadlo.e + ", h=" + Pocitadlo.h
        + ", c=" + Pocitadlo.c);
  }
}
```

Vypíše:

```
HashSet
1h 2h 3h 4h 5h 6h 7h 8h 9h 10h 11h 1e
pocet: 10 [34, 35, 32, 33, 38, 39, 36, 37, 31, 30, ]
e=1, h=11, c=0

TreeSet
1c 2c 3c 4c 5c 6c 7c 8c 9c 10c 11c 12c 13c 14c 15c 16c 17c 18c 19c 20c 21c ►
22c 23c 24c 25c 26c 27c 28c 29c
pocet: 10 [30, 31, 32, 33, 34, 35, 36, 37, 38, 39, ]
e=0, h=0, c=29
```

Z výpisu pro `HashSet` je vidět, že metoda `compareTo()` není vůbec zapotřebí. Dále je vidět, že platí kontrakt mezi metodami `equals()` a `hashCode()`. Při pokusu o vložení dalšího prvku do množiny je nejprve volána `hashCode()` a v případě, že vrátí odlišnou hodnotu, od hodnot z již vložených objektů, není třeba volat `equals()`. V případě, že `hashCode()` vrátí stejnou hodnotu (dodatečně vkládaná `Hruska(35)`, `equals()` definitivně rozhodne, zda se tento objekt rovná (=nevloží se) či nerovná (=vloží se). Vložené prvky jsou v množině v „náhodném" pořadí.

Z výpisu pro `TreeSet`, je vidět, že veškeré porovnávání je provedeno pomocí `compareTo()`. Je také vidět, že vkládání do TreeSet je pomalejší, protože se zároveň vkládaný prvek zařazuje mezi již vložené. Vložené prvky jsou v množině v vzestupném pořadí.

Z ukázky vyplývá, že pokud připravujeme třídu, která má být vkládána do množiny, měla by mít implementovány metody `hashCode()`, `equals()` a `compareTo()`. Tím později neomezujeme použitou implementaci kolekce množiny.

3.11.1.1. Vkládaná třída neimplementuje `Comparable`

- pokud vkládaná třída neimplementuje `Comparable`, nelze v této podobě `TreeSet` použít
  - po spuštění je vyhozena výjimka `Hruska cannot be cast to java.lang.Comparable`
- situace se řeší tím, že připravíme komparátor pro absolutní řazení a jeho instanci předáme jako parametr konstruktoru `TreeSet`
  - `TreeSet` bude pak používat pro práci s vloženými objekty pouze absolutní řazení
  - komparátor se typicky připravuje jako anonymní třída (viz dále), která je přiřazena do statické konstanty, zde `PODLE_CENY`

```
public static final Comparator<Hruska> PODLE_CENY =
                           new Comparator<Hruska>() {
  public int compare(Hruska h1, Hruska h2) {
    Pocitadlo.c++;
    System.out.print("" + Pocitadlo.c + "c ");

    if (h1.getCena() > h2.getCena())
      return (+1);
    else if (h1.getCena() == h2.getCena())
      return (0);
    else
      return (-1);
  }
};
```

  - je ale možné připravit novou třídu `Porovnavac`
    - zde je navíc použit trik pro zjednodušení porovnání, kdy dvě ceny jednoduše odečteme a vrátíme rozdíl

```
final class Porovnavac implements Comparator<Hruska> {
   public int compare(Hruska h1, Hruska h2) {
    Pocitadlo.c++;
    System.out.print("" + Pocitadlo.c + "c ");
```

```
      return h1.getCena() - h2.getCena();
    }
  }
```

z výpisu je vidět, že program pracuje stejně jako v předchozím případě

```
TreeSet
1c 2c 3c 4c 5c 6c 7c 8c 9c 10c 11c 12c 13c 14c 15c 16c 17c 18c 19c 20c 21c ►
22c 23c 24c 25c 26c 27c 28c 29c
pocet: 10 [30, 31, 32, 33, 34, 35, 36, 37, 38, 39, ]
e=0, h=0, c=29
```

- metoda `praceSHruskami()` zůstala nezměněna

```
import java.util.*;

class Pocitadlo {
  public static int e = 0;
  public static int h = 0;
  public static int c = 0;
}

class Hruska {
  private int cena;

  Hruska(int cena) { this.cena = cena; }

  public int getCena() { return cena; }

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

final class Porovnavac implements Comparator<Hruska> {
  public int compare(Hruska h1, Hruska h2) {
    Pocitadlo.c++;
    System.out.print("" + Pocitadlo.c + "c ");

    return h1.getCena() - h2.getCena();
  }
}

public class HruskyComparator {

  public static final Comparator<Hruska> PODLE_CENY =
                            new Comparator<Hruska>() {
    public int compare(Hruska h1, Hruska h2) {
      Pocitadlo.c++;
      System.out.print("" + Pocitadlo.c + "c ");

      if (h1.getCena() > h2.getCena())
        return (+1);
      else if (h1.getCena() == h2.getCena())
```

```
        return (0);
      else
        return (-1);
    }
  };

  static void praceSHruskami(Set<Hruska> mnozina) {
    System.out.println("\n" + mnozina.getClass().getSimpleName());
    for (int i = 30;  i < 40;  i++) {
      mnozina.add(new Hruska(i));
    }
    mnozina.add(new Hruska(35));

    System.out.print("\npocet: " + mnozina.size() + " [");
    for (Hruska h: mnozina) {
      System.out.print(h + ", ");
    }
    System.out.println("]");
  }

  public static void main(String[] args) {
//    praceSHruskami(new TreeSet<Hruska>(new Porovnavac()));
    praceSHruskami(new TreeSet<Hruska>(PODLE_CENY));
    System.out.println("e=" + Pocitadlo.e + ", h=" + Pocitadlo.h
        + ", c=" + Pocitadlo.c);
  }
}
```

3.11.1.2. Vkládaná třída nepřekrývá `equals()` a `hashCode()`

- tyto dvě metody se dědí z `Object`, takže je možné i pro tuto třídu `Hruska` použít ihned `HashSet`

```
class Hruska {
  private int cena;

  Hruska(int cena) { this.cena = cena; }

  public int getCena() { return cena; }

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

    praceSHruskami(new HashSet<Hruska>());
```

protože je však `equals()` naprogramována v Object na nejpřísnější porovnání, kdy jsou objekty stejné, pokud se jedná o stejné instance, bude do množiny chybně vložena již existující `Hruska(35)`

```
HashSet
pocet: 11 [36, 35, 34, 30, 32, 31, 35, 39, 37, 33, 38, ]
```

3.11.2. Problémy objektů v hešovacích třídách

- hešování (*hashing*, **rozptylové tabulky**) je jednou ze základních programovacích technik – podrobně viz v KIV/PPA2 a KIV/PT
  - pomáhá výrazně urychlit vkládání a výběr do/z kolekcí
  - implementační podrobnosti nás zatím nezajímají – ty jsou již vyřešeny ve třídě `HashSet`
  - podstatné pro využití této techniky je to, že vkládané objekty by měly poskytovat službu „reprezentuj se unikátním celým číslem“
  - v Javě se očekává, že tato služba bude realizována pomocí metody `int hashCode()`
  - pokud metoda nevrací pro různé objekty unikátní čísla, program funguje, ovšem mnohem méně efektivně – implementační vysvětlení viz ve zmíněných předmětech
    - vysvětlení z rozhraní je to, že při stejných hešovacích kódech se musí pro ověření shodnosti následně volat metoda `equals()`
    - ve skutečnosti musíme současně překrýt metody `hashCode()` a `equals()`
- častou chybou je, že 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

3.11.2.1. Metoda `equals()`

- pět pravidel obecného kontraktu (opakování z dřívějška)

1. objekt se musí vždy rovnat sám sobě – reflexivnost

   `x.equals(x)` je vždy `true`

2. objekty se musí rovnat křížem – symetričnost

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

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`

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 3.15.

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;
  }
}
```

3.11.2.2. 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

3.11.2.3. Efektivní hashCode()

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

Návod z knihy **Java efektivně**:

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

Poznámka

Eclipse umožňuje vygenerovat metody `equals()` a `hashCode()`. Efektivita vygenerované `hashCode()` (liší se konstantami prvočísel) je maličko nižší než u výše uvedené `hashCode()`. Tzn. vyplatí se použít tu z Eclipse, která je zadarmo.

Příklad 3.16. Ukázka použití různých přístupů k hešování

Je použita třída `Osoba`, která má základní atributy typu `boolean`, `int`, `double` a `String`. Tato třída má připravenu 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()` připravenu 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 vracena 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;
  }
}

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

1. 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.

2. Na příkladu `NemennaPerfektniOsoba` je vidět, že pokud nechceme měnit stav vkládaného objektu, můžeme zvýšit rychlost až o jednu třetinu.

3. 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

3.11.3. 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)`

3.11.4. 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

```
  SortedSet<String> treeset = new TreeSet<String>();
  treeset.add("ahoj");
  treeset.add("akce");
  treeset.add("brzo");
  treeset.add("eso");
  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());
  }
}
```

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

Vypíše:

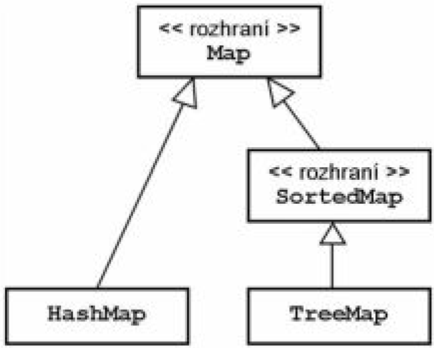
```
[3, 2] je podmnozinou [3, 2, 4, 1]
[3, 5, 2, 4, 1] je sjednocenim [3, 2, 4, 1] a [3, 5, 2]
[3, 2] je prunikem [3, 2, 4, 1] a [3, 5, 2]
[4, 1] je rozdilem [3, 2, 4, 1] a [3, 5, 2]
[5, 4, 1] je symetrickym rozdilem [3, 2, 4, 1] a [3, 5, 2]
```

3.12. 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 prázdnost
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 – nejpoužívanější 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`

```
public class ProchazeniMap {
  public static void main(String[] args) {
    Map<String, Integer> hm = new HashMap<String, Integer>();
    hm.put("prvni", 1);
    hm.put("druhy", 2);
    hm.put("treti", 3);

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

    // iterator
    for (Iterator<Map.Entry<String, Integer>>
           it = hm.entrySet().iterator();
           it.hasNext(); ) {
      Map.Entry<String, Integer> me = it.next();
//      it.remove();
      System.out.print(me.getKey() + "=" + me.getValue() + ", ");
    }
  }
}
```

Vypíše:

```
prvni=1, treti=3, druhy=2,
prvni=1, treti=3, druhy=2, ►
```

- 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 (stejné jméno, jiná váha)
2. vložit nový prvek se stejným klíčem (stejné jméno, jiná váha)
3. zavést vlastní třídu `Vaha`, jejíž datový prvek lze měnit

3.12.1. 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
- určitý komparátor pro absolutní řazení 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>`
- komparátor používaný v konkrétní `TreeMap` se dá zjistit metodou `Comparator<K> comparator()`
  - vrací buď objekt použitého komparátoru (absolutní řazení) nebo `null` (přirozené řazení)
- 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 3.17. 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);
    Map<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);
  }
}
```

Vypíše:

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

Příklad 3.18. 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}
```

3.13. Složitější datové struktury

- dlouhou dobu u kolekcí vystačíme s vkládáním datového typu `String`, obalových datových typů a hodnotových neměnitelných tříd
- je třeba si ale uvědomit, že kolekce nijak typ vkládaných objektů neomezují
  - do kolekce lze vložit jiná kolekce
  - tím lze vytvářet libovolně složité datové struktury
- typické příklady:
  - dvourozměrný seznam: `List<List<String>> dvourozmernySeznam;`

```
public static void dvourozmernySeznamRetezcu() {
  List<List<String>> dvourozmernySeznam;

  // postupne vytvareni
  dvourozmernySeznam = new ArrayList<List<String>>();

  for (int i = 0;  i < 2;  i++) {
    dvourozmernySeznam.add(new ArrayList<String>());
    List<String> pomSeznam = dvourozmernySeznam.get(i);
    for (int j = 0; j < 10; j++) {
      String pom = "" + (i + 1) + "-" + (j + 1);  // vzor 1-1
```

```
        pomSeznam.add(pom);
      }
    }

    // pruchod iteratorem
    for (List<String> pomSeznam : dvourozmernySeznam) {
      System.out.println("\nRadka:");
      for (String prvek : pomSeznam) {
        System.out.print(prvek + ", ");
      }
    }
  }
```

vypíše:

```
Radka:
1-1, 1-2, 1-3, 1-4, 1-5, 1-6, 1-7, 1-8, 1-9, 1-10,
Radka:
2-1, 2-2, 2-3, 2-4, 2-5, 2-6, 2-7, 2-8, 2-9, 2-10,
```

- mapa množin: `Map<String, Set<String>> mapaMnozin;`

množina je použita proto, že by se přítelkyně neměly opakovat a na jejich pořadí nezáleží

```
  public static void mapaMnozinRetezcu() {

    Map<String, Set<String>> mapaMnozin;

    // pomocna data
    String[] muzi = {"Karel", "Standa", "Honza"};
   String[] pritelkyne = {"Jana", "Hana", "Dana", "Anna", "Zina", "Dita"};
    Random r = new Random(2);

    // postupne vytvareni
    mapaMnozin = new HashMap<String, Set<String>>();

    for (int i = 0;  i < muzi.length;  i++) {
      Set<String> mnozinaPritelkyn = new HashSet<String>();

      int pocetPritelkyn = r.nextInt(pritelkyne.length) + 1;
      while (mnozinaPritelkyn.size() < pocetPritelkyn) {
        int poradi = r.nextInt(pritelkyne.length);
        mnozinaPritelkyn.add(pritelkyne[poradi]);
      }
      mapaMnozin.put(muzi[i], mnozinaPritelkyn);
    }

    // pruchod iteratorem
   for (Map.Entry<String, Set<String>> prvekMapy : mapaMnozin.entrySet()) {
     System.out.println("Muz: " + prvekMapy.getKey() + " a pritelkyne:");
      for (String prvekMnoziny : prvekMapy.getValue()) {
        System.out.print(prvekMnoziny + ", ");
      }
```

```
      System.out.println();
    }
  }
```

vypíše:

```
Muz: Standa a pritelkyne:
Dana,
Muz: Honza a pritelkyne:
Dana, Hana, Anna,
Muz: Karel a pritelkyne:
Jana, Zina, Dana, Hana, Anna,
```

Kapitola 4. Schémové jazyky DTD a XSD

4.1. 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*

4.2. 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

4.2.1. 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 zřejmé od samého počátku a proto rychle vznikaly další schémové jazyky
  - v současné době jsou používané 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

4.2.2. 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

4.2.3. 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"?>
```

4.2.3.1. 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

4.2.3.1.1. Prázdný element

- nejjednodušší – `EMPTY`

```
<!ELEMENT br EMPTY>
```

- v XML pak `<br></br>` nebo jen `<br/>`

Výstraha

Mezi značkami nesmí být odřádkováno – chybně:

```
<br>
</br>
```

4.2.3.1.2. 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 4.1.

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

<!DOCTYPE osoby SYSTEM "osoby.dtd">

<osoby>
  <osobaStrukturovaná>
    <jméno>
      <křestní>
        Bilbo
      </křestní>
      <příjmení>
        Pytlík
      </příjmení>
    </jméno>
    <pohlaví>
      <muž/>
    </pohlaví>
  </osobaStrukturovaná>

  <osobaStrukturovaná>
    <jméno>
      <příjmení>
        Roberts
      </příjmení>
      <křestní>
        Julie
      </křestní>
    </jméno>
    <pohlaví>
      <žena/>
    </pohlaví>
  </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 proděkani

    ```
    <!ELEMENT fakulta (proděkan, proděkan+)>
    ```

  - pokud je jídlo, pak se skládá z polévky a hlavního chodu

    ```
    <!ELEMENT jídlo (polévka, hlavníChod)?>
    ```

4.2.3.1.3. Konečný element

- obsahuje vlastní text

  ```
  <!ELEMENT polévka (#PCDATA)>
  ```

- v XML může být např.

  ```
  <polévka>gulášová</polévka>
  ```

4.2.3.1.4. 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 opakovač `*`

```
<!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>
```

4.2.3.2. 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

4.2.3.2.1. 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 (jedenTalíř | dvaTalíře | třiTalíře)>
```

- v XML může být např. :

```
<polévka množství="dvaTalíře">
```

4.2.3.2.2. 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 (jedenTalíř | dvaTalíře | třiTalíře)
                           "jedenTalíř">
```

- v XML může být např.:

```
<polévka>
```

ve smyslu

```
<polévka množství="jedenTalíř">
```

- 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

4.2.3.2.3. 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

4.2.3.3. 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

4.2.3.4. 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 4.2.

```
<?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>
      </pacient>
    </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)*>
```

4.3. W3C XML Schema – WXS nebo XSD

4.3.1. 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 možností 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

4.3.2. Praktické použití XSD

- z velké množiny možností bude popisována jen nejnutnější podmnožina pro datově orientované dokumenty
  - navíc bude používán jen jeden z více možných způsobů zápisu

4.3.3. 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 restrikcí 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 restrikce 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"/>
```

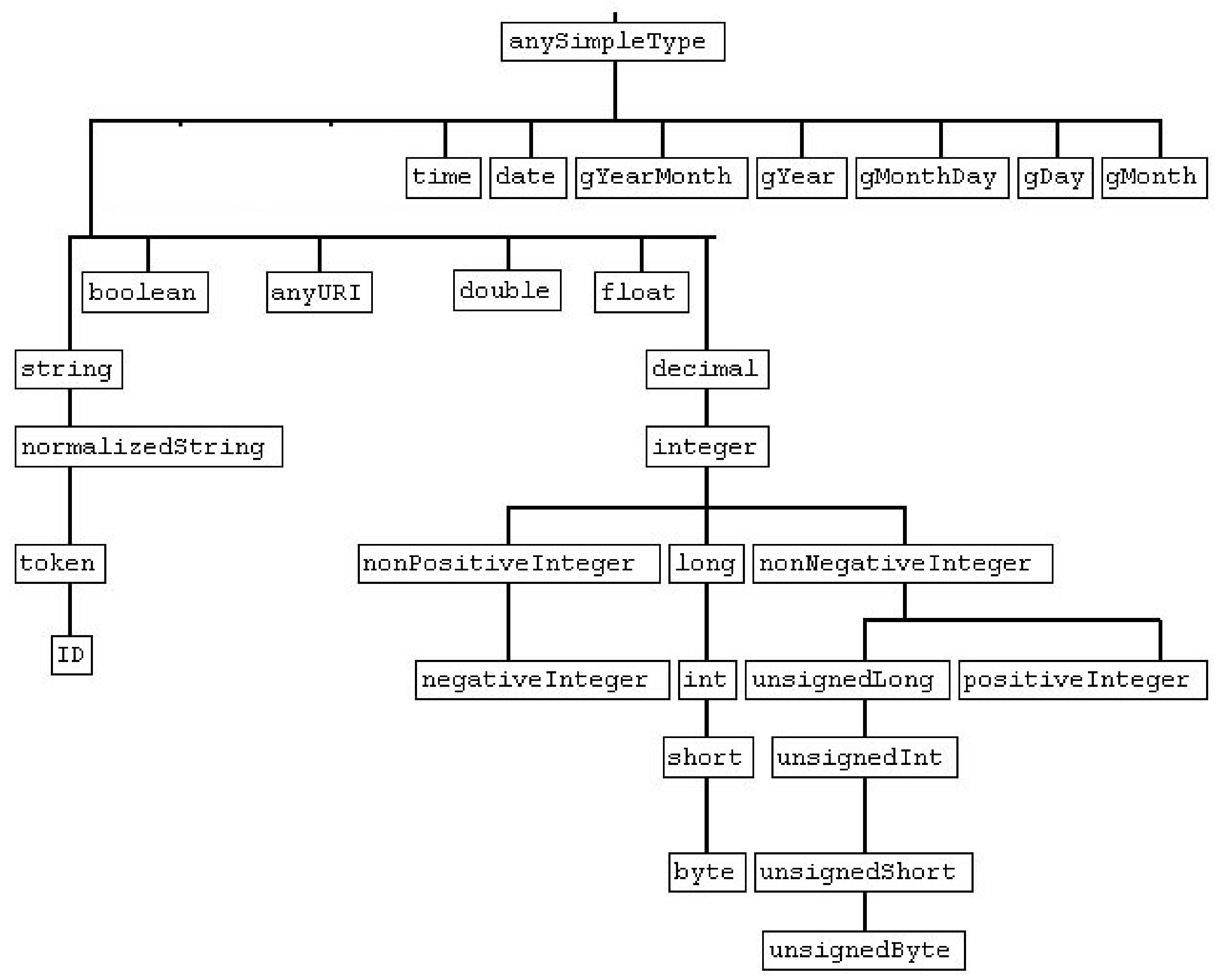
```
  </xs:complexType>

  <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 „matrjóška“ nebo „salámová kolečka“ (viz literatura)

4.3.4. Výběr běžně použitelných základních typů



4.3.5. Jaké možnosti nám dávají základní typy

a. datumy a časy

`dateTime`	datum a čas	`YYYY-MM-DD-THH:MM:SS`	2004-12-08T11:05:48
`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

(mezi datumem a časem je oddělovací znak velké `T`; u měsíce a dne jsou úvodní dvě pomlčky 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

4.3.6. Možnosti restrikcí (*constraining facets*)

Poznámka

Vždy se snažíme nalézt maximální možnou restrikci – 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: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

- opakovače mají běžný význam:

  `+` – jednou nebo vícekrát

  `*` – nula nebo vícekrát

  `?` – nula nebo jednou

  bez opakovače – právě jednou

4.3.7. 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

4.3.7.1. 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>
```

4.3.7.2. 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>
```

4.3.7.3. 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>
```

4.3.7.4. 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>
```

4.3.7.5. Prázdný element

- element je důležitý pouze svým výskytem nebo atributy (viz dále)

```
<xs:complexType name="svobodnyType">
</xs:complexType>
```

4.3.8. 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>
```

4.3.9. 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>
```

4.3.10. 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>
```

4.3.11. 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

4.3.12. 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:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="URIsouboru.xsd">
```

- všechny ostatní řetězce (kromě "`kořenovýElement`" a "`URIsouboru.xsd`") musí být přesně jako v ukázce
  - spleteme-li se, validace neproběhne úspěšně
- příklad pro výše uvedený `otec`

```
<otec
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="otec.xsd">
```

- příklad pro výše uvedený `obrazek` (má navíc atribut a je to prázdný element, což je hodně netypické)

```
<obrazek
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="obrazek.xsd" jmeno="pic001.jpg"/>
```

4.3.13. Validace pomocí xerces

```
xerces -v -s jmeno-souboru.xml
```

4.3.14. 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 jedněch 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>
```

4.3.14.1. 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>
```

4.3.14.2. XSD, které využívá značek z jiného XSD

- tento příklad je zesložitění předchozího případu
  - jedná se o situaci, kdy se v jednom XSD definují značky, které budou používány v jiném XSD
- soubor `zcu-schema.xsd`
  - oproti předchozímu případu jsou zde jen dvě změny

```
<xs:attribute name="zkratkaPracoviste" ...

<xs:element name="pocetKateder" ...
```

  - oba příkazy definují jména značek (atributu a elementu), které budou používána v jiném XSD

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="urn:x-herout:schemata:zcu.1.0"
           xmlns="urn:x-herout:schemata:zcu.1.0"
           elementFormDefault="qualified">

  <xs:simpleType name="zkratkaPracovisteType">
    <xs:restriction base="xs:string">
      <xs:pattern value="\p{Lu}{2,3}" />
    </xs:restriction>
  </xs:simpleType>

  <!-- pojmenovano pro pouziti v fav-schema.xsd -->
  <xs:attribute name="zkratkaPracoviste"
                type="zkratkaPracovisteType"/>

  <xs:simpleType name="pocetKatederType">
```

```
    <xs:restriction base="xs:nonNegativeInteger">
      <xs:minInclusive value="1" />
      <xs:maxExclusive value="20" />
      </xs:restriction>
    </xs:simpleType>

    <!-- pojmenovano pro pouziti v fav-schema.xsd -->
    <xs:element name="pocetKateder"
                type="pocetKatederType"/>

    <xs:complexType name="fakultaType">
      <xs:sequence>
        <xs:element name="pocetKateder" type="pocetKatederType"/>
      </xs:sequence>
      <xs:attribute name="zkratkaPracoviste"
                    type="zkratkaPracovisteType"
                    use="required"/>
    </xs:complexType>

    <xs:complexType name="zcuType">
      <xs:sequence>
        <xs:element name="fakulta" type="fakultaType"
                    minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>

    <xs:element name="zcu"
                type="zcuType"/>
</xs:schema>
```

- soubor `zcu.xml`
  - je zde použit implicitní jmenný prostor

```
<?xml version="1.0" encoding="utf-8" ?>
<zcu
  xmlns="urn:x-herout:schemata:zcu.1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:x-herout:schemata:zcu.1.0
                      zcu-schema.xsd">
  <fakulta zkratkaPracoviste="FAV">
    <pocetKateder>5</pocetKateder>
  </fakulta>
  <fakulta zkratkaPracoviste="FPE">
    <pocetKateder>18</pocetKateder>
  </fakulta>
</zcu>
```

- soubor `fav-schema.xsd`
  - na samém začátku již musí být definice jmenného prostoru `zcu`

```
xmlns:zcu="urn:x-herout:schemata:zcu.1.0"
```

- další příkaz importuje druhé XSD, takže bude možno používat jeho definované typy

```
<xs:import namespace="urn:x-herout:schemata:zcu.1.0"
           schemaLocation="zcu-schema.xsd"/>
```

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="urn:x-herout:schemata:fav.1.0"
           xmlns="urn:x-herout:schemata:fav.1.0"
           xmlns:zcu="urn:x-herout:schemata:zcu.1.0"
           elementFormDefault="qualified">

  <xs:import namespace="urn:x-herout:schemata:zcu.1.0"
             schemaLocation="zcu-schema.xsd"/>

  <xs:simpleType name="nazevType">
    <xs:restriction base="xs:string">
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="katedraType">
    <xs:sequence>
      <xs:element name="nazev" type="nazevType"/>
    </xs:sequence>
    <xs:attribute ref="zcu:zkratkaPracoviste"
                  use="required"/>
  </xs:complexType>

  <xs:complexType name="favType">
    <xs:sequence>
      <xs:element ref="zcu:pocetKateder"/>
      <xs:element name="katedra" type="katedraType"
                  minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="fav" type="favType"/>
</xs:schema>
```

- soubor `fav.xml`
  - pro značky z `fav-schema.xsd` je zde použit implicitní jmenný prostor
  - soubor `zcu-schema.xsd` není třeba připojovat, protože je již importován v souboru `fav-schema.xsd`

```
<?xml version="1.0" encoding="utf-8" ?>
<fav
  xmlns="urn:x-herout:schemata:fav.1.0"
  xmlns:zcu="urn:x-herout:schemata:zcu.1.0"
```

```
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:x-herout:schemata:fav.1.0
                      fav-schema.xsd">

  <zcu:pocetKateder>5</zcu:pocetKateder>
  <katedra zcu:zkratkaPracoviste="KIV">
    <nazev>Informatika a výpočetní technika</nazev>
  </katedra>
  <katedra zcu:zkratkaPracoviste="KKY">
    <nazev>Kybernetika</nazev>
  </katedra>
</fav>
```

4.3.15. 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 funkcni moznost
  <druhy/>
-->
</autori>
```

■ příklad zbytku jiného XML souboru

```
  <prvni>
     Müller
  </prvni>
  <druhy>
     Thurgau
  </druhy>
</autori>
```

4.3.16. 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 5. Java a XML, JAXB

5.1. 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

5.1.1. Obecné vlastnosti parseru

- čte XML ze souboru (nebo obecně ze vstupního proudu)
  - provádí nízkoúrovňovou 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 popsané jeho rozhraní (API), kterému pak vyhovuje určitý parser
- základní dělení rozhraní
  - proudové čtení
  - práce se stromovou reprezentací dokumentu

5.1.1.1. 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 nízkoúrovňové („přes ruku“) s množstvím pomocných proměnných nebo vlastní nadstavbou
  - prakticky jen pro čtení

5.1.1.2. 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 infoset 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

5.1.2. Rozhraní parserů pro Javu

5.1.2.1. 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 odstíněni

■ JAXP je součástí Java Core API 1.5

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

■ informace `http://java.sun.com/webservices/jaxp/`

5.1.2.2. 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* (JWSDP 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.]

5.1.3. Základní dělení parserů

1. podle způsobu zpracování XML dokumentu

   ### Proudové čtení

   - *push parsery* – SAX
     - čtení XML probíhá automaticky, generuje události, na které reagujeme
   - *pull parsery* – 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 infoset 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 infosetu
- 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

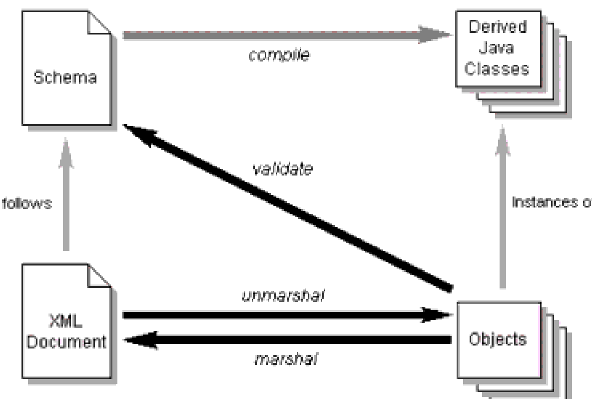
5.1.4. Přehled parserů

- 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, ...

5.2. *Java Architecture for XML Binding* – 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 (desítky MB)

- nástroje a API JAXB dovolují
  - jednorázově vygenerovat API našich tříd na základě XSD
  - načíst XML dokument do paměti do objektového modelu (*unmarshall*), 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 (*marshall*)
- 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.*;
```

Poznámka

Technologie JAXB ve spojení s moderními RAD (Eclipse, NetBeans, ...) výrazným způsobem zvyšuje produktivitu programátora. Neocenitelná je zejména kontextová nápověda. Výsledný program často funguje „na první pokus“.

5.2.1. Podpora z Ant

- pro práci je velmi vhodné využít Ant
  - soubor `build.xml` má pět cílů (*target*)
    - help – výpis návodu
    - generování – proběhne pouze jednou na začátku práce
    - ladění – spouští se po každé úpravě zdrojového kódu v Javě (*default target*), pokud používáme ladění z příkazové řádky
    - 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="help"
         basedir=".">

  <description>
    Univerzalni generacni, prekladaci a ladici
```

```
    predpis pro JAXB
    pro JDK 1.6 -- xjc.exe
    P.Herout, unor 2009
    celkovy popis v Help
  </description>

<!-- Nutno zmenit vzdy -->
  <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"/>

<!-- Menit jen pokud budeme Ant pouzivat i pro preklad
     a spousteni aplikace -->
  <property name="souborSFunkci_main()_ladeneAplikace"
           value="PripravaDat1"/>
  <property name="nazevBalikuLadeneAplikace"
           value=""/> <!-- pokud neni prazdny,
                        na konci musi byt tecka,
                        napr. "balik." -->

<!-- properties menene dle platformy -->
  <property name="kodovaniZdrojeXSD" value="utf-8"/>
  <property name="kodovaniZdrojeJava" value="utf-8"/>
  <!-- value="ISO-8859-2"
       value="windows-1250"
  -->

<!-- Od tohoto mista nic nemenit,
     pokud presne nevite, co delate -->
  <property environment="e"/>
  <property name="java"
           value="${e.JAVA_HOME}"/>

  <fileset id="XSDSoubory"
           dir=".">
    <include name="*.xsd" />
  </fileset>

  <property name="soubory" refid="XSDSoubory" />


  <target name="help">
     <echo message="Univerzalni generacni, prekladaci a ladici predpis pro ►
```

```
JAXB" />
     <echo message="Spusteni:" />
     <echo message="&gt;ant generovani" />
     <echo message="  -- ze souboru" />
     <echo message="       ${soubory}" />
     <echo message="       v kodovani: ${kodovaniZdrojeXSD}" />
     <echo message="     vygeneruje prislusne tridy v Jave, prelozi je" />
     <echo message="     a pripravi k nim JavaDoc dokumentaci" />
     <echo message="     a vysledek zabali i s dokumentaci do souboru:" />
     <echo message="       ${nazevBalikuGenerovanychTrid}.jar"/>
     <echo message="&gt;ant ladeni" />
     <echo message="  -- preklada a spusti ladenou aplikaci" />
     <echo message="&gt;ant distribuce" />
     <echo message="  -- vyrobi primo spustitelny .jar z aplikace" />
     <echo message="" />
  </target>

  <target name="generovani">
    <mkdir dir="${adresarGenerovanychSouboru}" />
    <!-- pomoci xjc.exe generuje soubory z .xsd souboru
         nelze nastavit vystupni kodovani - koduje do windows-1250
    -->
    <exec dir="." executable="${java}\bin\xjc">
      <arg line="*.xsd"/>
      <arg line="-d ${adresarGenerovanychSouboru}"/>
      <arg line="-p ${nazevBalikuGenerovanychTrid}"/>
    </exec>

    <!-- prelozi generovane soubory
         vstupni kodovani je windows-1250
    -->
    <mkdir dir="${adresarClassSouboru}" />
    <javac
       srcdir="${adresarGenerovanychSouboru}"
       destdir="${adresarClassSouboru}"
       debug="on">
    </javac>

    <!-- vyrobi dokumentaci k vygenerovanym souborum
         vstupni kodovani je windows-1250
    -->
    <mkdir dir="${adresarGenerovaneDokumentace}" />
    <javadoc
       sourcepath="${adresarGenerovanychSouboru}"
       destdir="${adresarGenerovaneDokumentace}"
       windowtitle="${nazevBalikuGenerovanychTrid}"
       docencoding="${kodovaniZdrojeXSD}"
       charset="${kodovaniZdrojeXSD}">
    </javadoc>

    <!-- vyrobi .jar z vygenerovanych souboru -->
    <jar
      destfile="${nazevBalikuGenerovanychTrid}.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="${adresarClassSouboru}"
      encoding="${kodovaniZdrojeJava}"
      includes="${souborSFunkci_main()_ladeneAplikace}.java"
      debug="on">
<!--      <compilerarg value="-Xlint" /> -->
    </javac>
    <java
      ►
classname="${nazevBalikuLadeneAplikace}${souborSFunkci_main()_ladeneAplikace}"
      fork="true">
    </java>
  </target>


  <target name="mazani">
    <delete dir="${adresarGenerovanychSouboru}" />
    <delete dir="${adresarGenerovaneDokumentace}" />
    <delete dir="${adresarClassSouboru}/${nazevBalikuGenerovanychTrid}" />
    <delete>
      <fileset dir="${adresarClassSouboru}">
        <include name="**/*.class"/>
        <exclude name="${nazevBalikuGenerovanychTrid}.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>
```

5.3. 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="nazevOvoceType">
    <xs:restriction base="xs:string">
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="nazevType">
    <xs:simpleContent>
      <xs:extension base="nazevOvoceType">
        <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`

5.3.1. 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)

5.4. 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
```

- nejlepší je přidat soubor `jidlobalik.jar` do Eclipse, kdy Eclipse pak poskytuje všechny výhody, zejména doplňování kódu

5.4.1. 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 odstiňuje 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

5.5. 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)

5.5.1. 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);
...
```

5.5.2. 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));
```

5.5.3. Problematické datové typy

- pokud je v XSD souboru použit datový typ odvozený od datumu a času, vygeneruje se odpovídající Java třída `XMLGregorianCalendar`
  - částečné problémy pak nastávají, chceme-li načtené hodnoty datumů a/nebo časů porovnávat programově s jinými
  - je možné připravit samostatnou instanci obsahující jen datum, jen čas nebo datum i čas
  - datum a čas dohromady se v XML souboru zapisují ve formátu `YYY-MM-DDTHH:MM:SS` (např. `2009-02-10T09:23:42`), tj. oddělovacím znakem je písmeno velké `T` bez mezer kolem něj
- je-li v XSD použit výčtový typ (`xs:enumeration`), je třeba dávat pozor na situaci, kdy jsou jednotlivé položky výčtového typu zapsány malými písmeny
  - tento výčtový typ se totiž překládá do Javovského `Enum`, ve kterém se nejčastěji pracuje jen s velkými písmeny
  - při práci s přečtenými hodnotami je pak nutné pracovat s metodou `value()` (nikoliv `toString()` nebo `name()`)

Soubor `casovyudaj.xml`

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

<casovyUdaj
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="casovyudaj-schema.xsd">

  <datum>2009-02-10</datum>
  <cas>09:23:42</cas>
  <datumACas>2009-02-10T09:23:42</datumACas>
  <den>Út</den>
</casovyUdaj>
```

Soubor `casovyudaj-schema.xml`

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:simpleType name="datumType">
    <xs:restriction base="xs:date">
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="casType">
    <xs:restriction base="xs:time">
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="datumACasType">
    <xs:restriction base="xs:dateTime">
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="denType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Po" />
      <xs:enumeration value="Út" />
      <xs:enumeration value="St" />
      <xs:enumeration value="Čt" />
      <xs:enumeration value="Pá" />
      <xs:enumeration value="So" />
      <xs:enumeration value="Ne" />
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="casovyUdajType">
    <xs:sequence>
      <xs:element name="datum" type="datumType"/>
      <xs:element name="cas" type="casType"/>
      <xs:element name="datumACas" type="datumACasType"/>
      <xs:element name="den" type="denType"/>
    </xs:sequence>
  </xs:complexType>
```

```
  <xs:element name="casovyUdaj" type="casovyUdajType"/>
</xs:schema>
```

Čtení tohoto XML souboru:

```
public class PraceSDatumemACasem {

  public static final String BALIK = "casovyudaj";
  public static final String VSTUP = "casovyudaj.xml";

  static void cteniAVypis() {
    try {
      JAXBContext jc = JAXBContext.newInstance(BALIK);
      Unmarshaller u = jc.createUnmarshaller();
      JAXBElement<?> element = (JAXBElement<?>) u.unmarshal(
                                  new File(VSTUP));

      CasovyUdajType casovyUdaj = (CasovyUdajType) element.getValue();

      XMLGregorianCalendar datum = casovyUdaj.getDatum();
      System.out.println("datum: " + datum);

      XMLGregorianCalendar cas = casovyUdaj.getCas();
      System.out.println("cas:   " + cas);

      XMLGregorianCalendar datumACas = casovyUdaj.getDatumACas();
      System.out.println("datum a cas: " + datumACas);

      DenType den = casovyUdaj.getDen();
      System.out.println("Enum.name():     " + den.name());     // chybne
      System.out.println("Enum.toString(): " + den.toString()); // chybne
      System.out.println("Enum.value():    " + den.value());    // spravne
    } catch (JAXBException e) {
        e.printStackTrace();
        System.exit(1);
    }
  }
```

vypíše:

```
datum: 2009-02-10
cas:   09:23:42
datum a cas: 2009-02-10T09:23:42
Enum.name():     ÚT
Enum.toString(): ÚT
Enum.value():    Út
```

Vytváření jednotlivých typů časových údajů např. pro vzájemné porovnávání metodou `compare()`:

```
  DatatypeFactory dtFactory = DatatypeFactory.newInstance();

  // datum
  XMLGregorianCalendar calDatum = dtFactory.newXMLGregorianCalendarDate(
```

```
            2009, 2, 10, DatatypeConstants.FIELD_UNDEFINED);

// cas
XMLGregorianCalendar calCas = dtFactory.newXMLGregorianCalendarTime(
            9, 20, 45, DatatypeConstants.FIELD_UNDEFINED);

// datum a cas
XMLGregorianCalendar calDatumACas = dtFactory.newXMLGregorianCalendar(
            2009, 2, 10, 9, 20, 45,
            DatatypeConstants.FIELD_UNDEFINED,
            DatatypeConstants.FIELD_UNDEFINED);
```

5.6. 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)

5.7. 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`, `Writter`, ...
  - 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
  - `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

Výstraha

Nepoužívat chybné:

`setProperty(Marshaller.JAXB_SCHEMA_LOCATION, SCHEMA);`

Příklad změní cenu u jablek a přidá další 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"));
        }
      }

      // vyrobeni noveho 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);

      // pridani noveho 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);
```

5.8. 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.

5.8.1. 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 vstupniho 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();

      // chybna zmena vahy grapefruitu
      for (OvoceType o: seznamOvoce) {
        String nazev = o.getNazev().getValue();
        if (nazev.equals("grapefruity") == true) {
          o.setVaha(-1.0);
        }
      }

      // validace objektu v pameti
      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'.]
```

5.9. Příprava kompletně nového dokumentu

5.9.1. Vše najednou

- 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();

      // vyrobeni noveho jidla - korenovy element
      JidloType jidlo = (JidloType) of.createJidloType();
      List<OvoceType> seznamOvoce = jidlo.getOvoce();

      // vyrobeni 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 &amp; &lt;</nazev>
        <vaha>0.9</vaha>
    </ovoce>
    <ovoce cislo="2">
        <nazev jednotkovaCena="42">ovoce 2 &amp; &lt;</nazev>
        <vaha>9.7</vaha>
    </ovoce>
    <ovoce cislo="3">
        <nazev jednotkovaCena="35">ovoce 3 &amp; &lt;</nazev>
        <vaha>7.8</vaha>
    </ovoce>
</jidlo>
```

5.9.2. Příprava a následný zápis

- příklad ukazuje práci s datovými typy pro datum a čas a také s výčtovým typem

```
  public static final String BALIK = "casovyudaj";
  public static final String SCHEMA = "casovyudaj-schema.xsd";
  public static final String VYSTUP = "novy-casovyudaj.xml";
  public static final String KODOVANI = "utf-8";

  static CasovyUdajType pripravaNoveho() {
    try {
      ObjectFactory of = new ObjectFactory();
      CasovyUdajType cut = of.createCasovyUdajType();
      DatatypeFactory dtFactory = DatatypeFactory.newInstance();

      // datum
      XMLGregorianCalendar calDatum = dtFactory.newXMLGregorianCalendarDate(
              2009, 2, 10, DatatypeConstants.FIELD_UNDEFINED);
      cut.setDatum(calDatum);

      // cas
      XMLGregorianCalendar calCas = dtFactory.newXMLGregorianCalendarTime(
              9, 20, 45, DatatypeConstants.FIELD_UNDEFINED);
      cut.setCas(calCas);

      // datum a cas
```

```
      XMLGregorianCalendar calDatumACas = dtFactory.newXMLGregorianCalendar(
          2009, 2, 10, 9, 20, 45,
          DatatypeConstants.FIELD_UNDEFINED,
          DatatypeConstants.FIELD_UNDEFINED);
      cut.setDatumACas(calDatumACas);

      // den - obe moznosti jsou spravne
//      cut.setDen(DenType.fromValue("Pá"));  // pouziti retezce
      cut.setDen(DenType.PÁ);                 // pouziti konstanty
      return cut;
    } catch (DatatypeConfigurationException e) {
        e.printStackTrace();
        System.exit(1);
    }
    return null;
  }

  static void zapis(CasovyUdajType cut) {
    ObjectFactory of = new ObjectFactory();
    JAXBElement<?> element = of.createCasovyUdaj(cut);

    try {
      JAXBContext jc = JAXBContext.newInstance(BALIK);
      Marshaller m = jc.createMarshaller();
      m.setProperty(Marshaller.JAXB_ENCODING, KODOVANI);
      m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
      m.setProperty(Marshaller.JAXB_NO_NAMESPACE_SCHEMA_LOCATION, SCHEMA);

      m.marshal(element, new FileOutputStream(VYSTUP));
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public static void main(String[] args) {
    CasovyUdajType cut = pripravaNoveho();
    zapis(cut);
  }
```

vygeneruje:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<casovyUdaj
   xsi:noNamespaceSchemaLocation="casovyudaj-schema.xsd"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <datum>2009-02-10</datum>
    <cas>09:20:45</cas>
    <datumACas>2009-02-10T09:20:45</datumACas>
    <den>Pá</den>
</casovyUdaj>
```

Kapitola 6. XSLT a XPath

6.1. Zdroje

6.1.1. Tutoriály a další

- `www.kosek.cz/xml/xslt`
  - XSLT v příkladech
  - velmi čtivé materiály týkající se ale XSLT 1.0

    Poznámka

    S laskavým svolením Jiřího Koska jsou podle tohoto zdroje připravovány tyto přednášky.
- `www.zvon.org/xxl/XSL-Ref/Tutorials`
  - XSLT 2.0 Tutorial postavený na komentovaných příkladech
  - velice často z celého světa odkazovaný zdroj, příklady z něj jsou např. v editoru oXygen
  - na stránkách `www.zvon.org` je mnoho dalších tutoriálů ohledně XML
- `www.saxonica.com/documentation`
  - užitečné při práci se Saxonem (XSLT procesor)
  - popis všeho, co Saxon podporuje
  - zejména je zde popis rozšíření Saxonu oproti standardu
- `www.dpawson.co.uk/xsl/rev2/rev2.html`
  - kvalitní stránky o problematice XSLT 2.0
  - na stránkách se dá najít i řada dalších užitečných rad, návodů a příkladů, např.:
    - `www.dpawson.co.uk/xsl/rev2/exampler2.html` – užitečné praktické příklady
- specifikace W3C
  - `www.w3.org/TR/xslt20/`
  - `www.w3.org/TR/xpath20/`
  - `www.w3.org/TR/xpath-functions/` – Functions and Operators

6.1.2. Editory a další nástroje

- `www.oxygenxml.com`
  - nejčastěji zmiňovaný editor pro vše týkající se XML a spol.

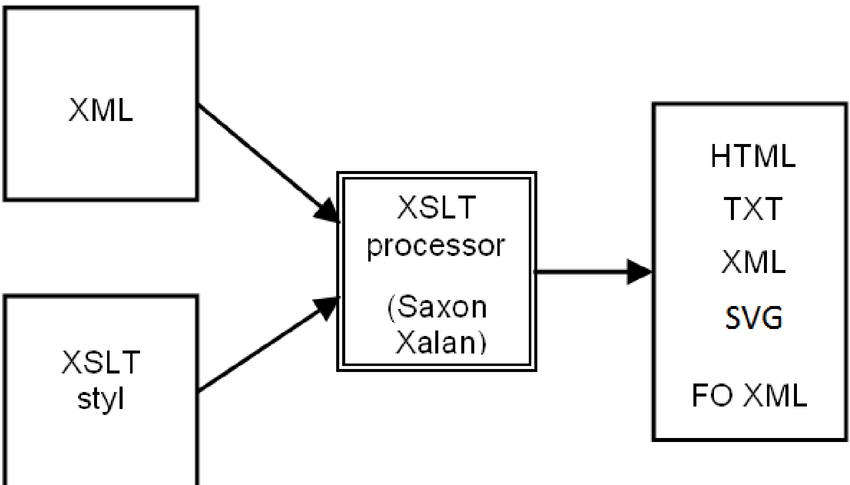
- má bezplatnou plně funkční 30 denní verzi

- nativní editory v NetBeans a .NET
- plug-in Orangevolt pro Eclipse
- `www.pspad.com`
  - univerzální editor, podporuje XSLT

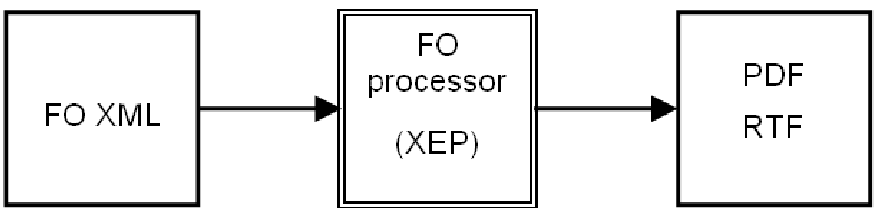
6.2. Základní principy XSL

- značky v XML popisují význam obsahu dokumentu nikoli jeho vzhled
  - většinou (s výjimkou B2B použití – viz dále) je ale někdy nutné zobrazit obsah XML a pak je forma vzhledu důležitá
  - řešení – vytvořit odděleně definici vzhledu jednotlivých elementů, tzv. *styl*, popsaný *stylovým jazykem*
    - pak se před vlastním zobrazením vytvoří z XML a stylu další dokument, který se (něčím) zobrazí
  - nejčastěji používané stylové jazyky:
    - kaskádové styly – CCS
    - *eXtensible Stylesheet Language* – XSL
- XSL prošel od roku 1996 značným vývojem a postupně se zjistilo, že slouží dvěma účelům

1. transformace dokumentů
   - *XSL Transformation* – XSLT
   - styly, jak provádět transformace nejčastěji do HTML, XML s jinou strukturou, TXT, SVG (grafika)
   - původní myšlenka popisu vzhledu je opuštěna
     - potřebujeme-li vzhled, zajistí se převodem do jiného jazyka
     - nejčastěji do HTML nebo do SVG
   - XSLT je standardem W3C od 1999
   - využití i pro B2B (*bussines to bussines*) aplikace, kdy se převádí XML v jedné struktuře do jiné struktury ale obsah zůstává (víceméně) stejný
   - transformace do TXT má význam, když potřebujeme jen čistě textový obsah XML dokumentu
     - lze ale použít i trik, kdy lze textový obsah „obalit“ dalším textem a tak vytvořit např. dávku SQL příkazů



2. přesný popis vzhledu dokumentů
   - *formátovací objekty* – XSL FO
   - velikost, styl písma, barvy, zarovnání odstavců, rozložení stránky, vícesloupcová sazba, atd.
   - XSL FO je dvoukroková transformace, v prvním kroku se využívá XSLT (viz výsledek z předchozího obrázku)



Poznámka

Další popis se bude týkat pouze XSLT.

6.2.1. Důvody použití stylů

- oddělení obsahu od vzhledu významně nabývá na důležitosti v případě, že se z jednoho obsahu generuje několik vzhledů
  - typický případ jsou tyto přednášky, kdy z jednoho XML zdroje se generuje
    - HTML pro přednášky
    - PDF pro tištěnou verzi
  - výhody
    - jen jeden zdrojový dokument
    - všechny změny a opravy se dělají pouze jednou
    - transformace jsou mechanická záležitost, kterou lze zautomatizovat

- významná výhoda je, pokud se transformace provádějí na žádost klienta
  - např. WWW server rozpozná typ klienta (mobilní telefon, PC s různými prohlížeči) a on-line mu připraví nejvhodnější formu dokumentu

- dalším důvodem je jednotnost stylu pro více XML dokumentů (příklad opět přednášky)
  - všechny jsou zformátovány stejně
  - při požadavku na změnu vzhledu se mění pouze jeden stylový soubor

6.2.2. Dostupné XSLT procesory

XSLT procesorů je značné množství

- Saxon (v 2011 ve verzi 9.3)
  - nejznámější a nejpoužívanější
  - nejrychlejší s nejlepší diagnostikou chyb a varování
  - vynikající dokumentace
  - napsaný prvotně v Javě, dostupný i v .NET
  - od verze 8 úplně implementuje XSLT 2.0 a XPath 2.0
  - dostupný jako volně šiřitelná verze Saxon-HE (*home edition*)
  - komerční verze jsou Saxon-PE (*professional edition*) a Saxon-EE (*enterprise edition*)
  - my používáme volně dostupnou a neměnnou verzi Saxon-B 9.1.0.8 (dřívější značení Saxonu)
    - má větší možnosti než Saxon-HE (např. příkaz `assign`)
- Xalan (v 2011 ve verzi 2.7.1 – vývoj je zastaven od 2009)
  - nejvýraznější výhoda – je součástí JDK od 1.4
  - napsaný prvotně v Javě, dostupný i v C++
  - úplně implementuje XSLT 1.0 a XPath 1.0
- dále existují XSLT procesory integrované ve webových prohlížečích
  - XSLT styl lze přímo připojit do XML dokumentu
  - transformace pak probíhá přímo v prohlížeči
    - není vhodné tuto možnost rozsáhle využívat
    - vhodné použití je pro jednoduché transformace kratších XML dokumentů
    - podrobně viz dále

6.2.3. Porovnání XSLT 2.0 a XQuery 1.0

- obě technologie slouží pro dotazování a transformace XML dat
- mnoho společných vlastností
  - založeny na XPath 2.0
  - sdílejí stejný datový model a stejnou sadu vestavěných funkcí
  - ze stejného XML dokumentu vytvoří (různými způsoby) stejný požadovaný výstup
- rozdíly
  - XSLT
    - XML dokumenty zpracováváme většinou celé nebo jejich podstatné části
      - proto XSLT procesor implicitně prochází celý XML dokument do hloubky
    - pracuje se převážně s texty (žádné komplikované výpočty)
    - XSLT není silně typovaný jazyk
    - XSLT je XML jazyk
    - výstupem jsou převážně XML dokumenty (nebo HTML)
  - XQuery
    - provádí dotazování nad XML dokumenty
      - to většinou znamená pohled jen na určité části XML dokumentů
    - pracuje s typovanými daty (složité výpočty možné)
    - XQuery je silně typovaný jazyk
    - XQuery není XML jazyk

6.2.4. Základy XSLT stylů

- budeme transformovat jednoduchý XML soubor `pozdrav-prvni.xml` (viz dále)
  - kódování `ISO-8859-2` je použito z pedagogických důvodů

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<pozdrav>ahoj programátoři</pozdrav>
```

6.2.4.1. Nutné části XSL souboru

soubor `prvni-styl.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

  <xsl:output method="html"
              encoding="windows-1250"
              indent="yes"/>

  <xsl:template match="/">
    <html>
      <body>
        <xsl:value-of select="."/>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

- XLST styl je uložen v souboru s příponou XSL (neplést s XLS – Excel)
- je to XML dokument
  - proto má standardní hlavičku `<?xml version="1.0" encoding="UTF-8"?>`
- kořenový element musí být `stylesheet`
- důsledně využívá jmenných prostorů
  - jmenný prostor pro řídicí značky musí být deklarován jako

    ```
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    ```

  - spleteme-li se v názvu, nebude XSLT procesor instrukce stylu vůbec zpracovávat
  - jeho používaný prefix je `xsl:`

    Poznámka

    Atributy nepoužívají prefix `xsl:`.
- povinný atribut kořenového elementu je `version="2.0"`
  - možné hodnoty jsou `"2.0"` nebo `"1.0"`
- dalším elementem je definice způsobu výstupu `xsl:output`
  - atribut `method` má čtyři základní hodnoty: `html`, `xhtml`, `xml` a `text`
  - druhý atribut je `encoding`, který stanovuje použité výstupní kódování
    - ♦ podporována jsou všechna standardní kódování – zde `windows-1250`
      - – setkáme se se třemi potenciálně různými kódováními

- vstupní XML soubor – zde `ISO-8859-2`
- transformační XSL soubor – zde `UTF-8`
- výstupní soubor – zde `windows-1250`

♦ použijeme-li `us-ascii` dostaneme (viz příklad dále) akcentované znaky zapsané v závislosti na hodnotě atributu `method`

– HTML

```
ahoj program&aacute;to&#345;i
```

– XML

```
ahoj program&#225;to&#345;i
```

● element `xsl:output` může mít další atributy (celkem 21), z nichž mnohé mají význam v závislosti na hodnotě atributu `method` (např. `standalone`)

♦ zde je uveden `indent="yes"` s významem odsazování výstupu

■ posledním elementem je `xsl:template`, který popisuje výkonou část stylu

● tento element se v XSLT stylech vyskytuje opakovaně

● při výběru částí dokumentu se používá dotazovací jazyk XPath (viz dále)

● obsahuje atribut `match`, což je XPath výraz identifikující uzly (elementy a atributy) (ekvivalent „nody“ z terminologie DOM)

♦ hodnota `/` znamená kořenový uzel

♦ protože každý XML dokument musí mít právě jeden kořenový uzel, lze tento styl použít pro jakýkoli XML dokument

● vnořený element `<xsl:value-of select="."/>` posílá celou textovou hodnotu vybraného elementu z XML dokumentu na výstup

♦ to, jaký element je vybrán, určuje opět XPath výraz, který je uveden v atributu `select`

– v tomto případě má význam „obsah aktuálního uzlu“

■ v XSL dokumentu se míchají dvě sady značek

● řídicí

♦ jsou to příkazy pro XSLT procesor a mají prefix `xsl:`

♦ element `<xsl:value-of select="."/>` zkopíruje na výstup hodnotu aktuálního uzlu

● značky výsledného dokumentu

♦ zde HTML značky `<html>` a `<body>`

- ♦ tyto elementy se kopírují 1:1 na výstup, protože jsou z jiného jmenného prostoru než `xsl:`

6.2.4.2. Provedení transformace

- pro provedení transformace potřebujeme vstupní XML dokument (`pozdrav-prvni.xml`)

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<pozdrav>ahoj programátoři</pozdrav>
```

- z příkazové řádky se transformace spustí

```
saxon -o výstupní_soubor dokument.xml styl.xsl
```

Varování

Parametr `-o výstupní_soubor` musí být uveden jako první!

v našem případě:

```
saxon -o prvni.html pozdrav-prvni.xml prvni-styl.xsl
```

- výsledkem je soubor `prvni.html`

```
<html>
   <body>ahoj programátoři</body>
</html>
```

- hodnota atributu `method` elementu `xsl:output` významně ovlivňuje výstup, např. pro hodnoty:

  - "xml"

```
<?xml version="1.0" encoding="windows-1250"?>
<html>
   <body>ahoj programátoři</body>
</html>
```

    - ♦ pro nastavení atributu `indent="no"`, dostaneme

```
<?xml version="1.0" encoding="windows-1250"?><html><body>ahoj ►
programátoři</body></html>
```

  - "xhtml"

```
<?xml version="1.0" encoding="windows-1250"?><html>
   <body>ahoj programátoři</body>
</html>
```

  - "text"

```
ahoj programátoři
```

    - zde se do výstupu nekopírují značky

- pozor na skutečnost, že transformace není typu proudový (*stream*) vstup -> výstup
  - ve skutečnosti se celý XML dokument načte do paměti jako strom (viz DOM později), aby bylo možné provádět XPath dotazy
    - nelze použít na extrémně velké XML dokumenty
  - strom z paměti se serializátorem dle pravidel uvedených ve stylovém XSL souboru zapisuje do výstupního souboru

Poznámka

Pro spouštění transformace je nejvhodnější použít nějaký editor, který umí transformaci spouštět přímo.

6.2.4.3. Připojení stylu ke XML souboru

- XML soubor nemusíme pro jeho zobrazení vždy transformovat pomocí XSLT procesoru
- je možné připojit transformační styl pomocí instrukce pro zpracování

```
<?xml-stylesheet type="text/xsl" href="pouzity_styl.xsl"?>
```

  - atribut `type` musí mít hodnotu `"text/xsl"` nikoliv `"text/plain"`
- doplněný předchozí příklad – soubor `pozdrav-a-styl.xml` (obsah souboru `prvni-styl.xsl` zůstal nezměněn):

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="prvni-styl.xsl"?>
<pozdrav>ahoj programátoři</pozdrav>
```

- v HTML prohlížeči se zobrazí:

```
ahoj programátoři
```

- tento způsob lze doporučit jen pro velmi specifické použití, protože nevýhody jsou:
  - pokud je XSL soubor nesprávně, v HTML výsledku nic nevidíme
    - prakticky je tedy před finálním připojením nutné XSL soubor odladit dříve uvedeným postupem (např. z příkazové řádky)
  - styl je v XML dokumentu uveden „natvrdo“
    - při požadavku použít jiný styl je nutno měnit XML dokument

6.2.4.4. Skutečný styl pro HTML soubor

- korektně napsaný HTML soubor má na svém začátku další doplňující informace (včetně uvedení kódování)

- další informace uvedené v elementu <meta>
- obojí lze snadno v XSLT stylu zadat
- XSLT styl korektni-styl-html.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

  <xsl:output method="html"
              encoding="UTF-8"
              indent="yes"
              doctype-public="-//W3C//DTD HTML 4.01//EN"
              doctype-system="http://www.w3.org/TR/html4/strict.dtd"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>Korektní HTML</title>
      </head>
      <body>
        <xsl:value-of select="."/>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

  - klíčový je element <head> – způsobí vypsání elementu <meta>

- po transformaci dostaneme:

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Korektní HTML</title>
   </head>
   <body>ahoj programátoři</body>
</html>
```

6.2.4.5. Minimální korektní styl

- liší se od předchozího prázdným elementem <head></head>

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">
```

```
  <xsl:output method="html"
              encoding="UTF-8"
              indent="yes"/>

  <xsl:template match="/">
    <html>
      <head></head>
      <body>
        <xsl:value-of select="."/>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

6.3. XPath – dotazovací jazyk nad XML dokumentem

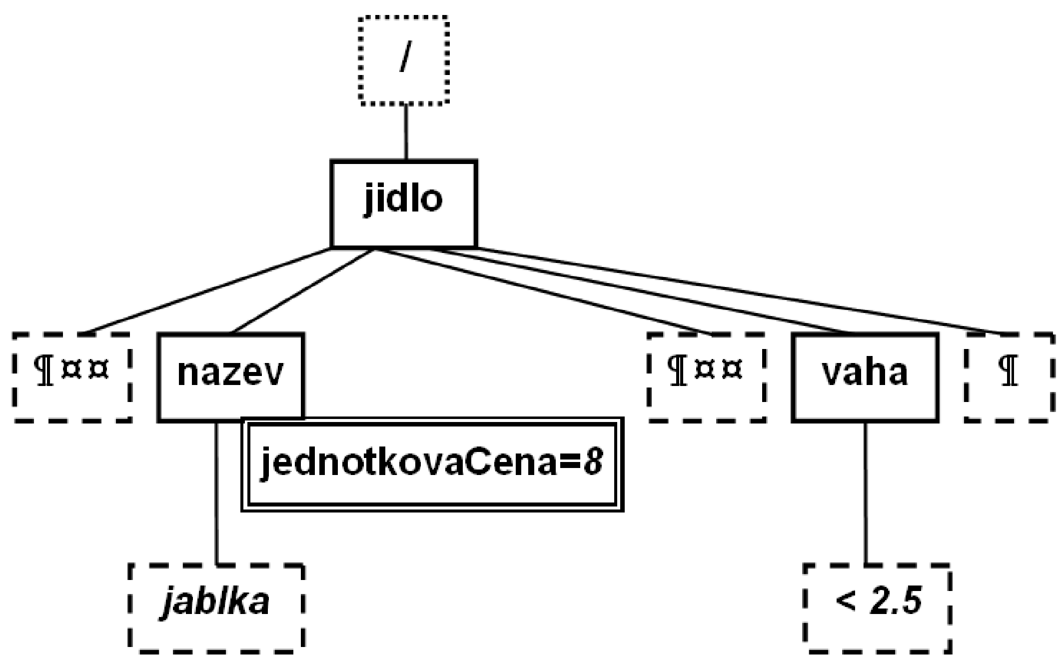
- XPath umožňuje zapisovat výrazy, jejichž výsledkem je nejčastěji množina uzlů
  - kromě toho XPath poskytuje více než 150 funkcí pro nejrůznější operace
    - ty většinou využijeme až při vytváření výstupu v XSLT (viz dále)
- XPath je nyní ve verzi 2.0
  - `www.w3.org/TR/xpath20/`

6.3.1. Abstraktní model dokumentu

- XSLT využívající XPath pracuje nad modelem XML dokumentu
  - ten se vytvoří celý v paměti (problém s velkými XML dokumenty)
  - model je velmi podobný DOM (viz později)
- model vychází ze specifikace XML Infoset
  - `www.w3.org/TR/xml-infoset`
- celý dokument je reprezentován stromovou strukturou
  - obsahuje uzly mnoha různých typů
- XML dokument `jidlo-entita.xml` (u elementu `vaha` je použita entita < ve smyslu „menší než“, tj. < 2.5)

```
<jidlo>
  <nazev jednotkovaCena="8">jablka</nazev>
  <vaha>&lt; 2.5</vaha>
</jidlo>
```

- má následující stromovou reprezentaci
  - zápis `¶¤¤` znamená odřádkování (¶) a dvě mezery (¤¤)



- na obrázku jsou zakresleny různé uzly, které mají typ, název a obsah
- používané typy uzlů
  - **kořenový uzel**
    - není to kořenový element zobrazovaného XML!
    - je to dodatečně přidaný uzel, jehož název je vždy `/` (podobnost s adresářovou strukturou)
    - jeho dítětem je kořenový element (zde `jidlo`) a dalším případným dítětem by byly komentáře uvedené před kořenovým elementem (zde nejsou)
  - **element**
    - každý element, který se vyskytuje v XML
      - zde `jidlo`, `nazev`, `vaha`
    - hierarchie je stejná jako v XML
    - název uzlu je název elementu
    - dítětem mohou být
      - elementy – `nazev` je dítětem `jidlo`
      - textové uzly – textový uzel s obsahem `jablka` je dítětem `nazev`
      - komentáře

  - k uzlu mohou být připojeny
    - atributy – `jednotkovaCena` je atributem uzlu `nazev`
    - jmenné prostory
  - obsah uzlu se skládá z textu, který uzel obsahuje
    - např. zde `nazev` má obsah `jablka`
    - pro nekoncové elementy je obsahem sloučený obsah všech uzlů, které jsou potomky
      - např. zde `jidlo` má obsah `¶¤¤jablka¶¤¤< 2.5¶`
- **atribut**
  - je připojen k odpovídajícímu elementu
  - atribut není dítě elementu, ale paradoxně se element chápe jako rodič atributu!
  - je to vždy koncový uzel
  - název je název atributu – zde `jednotkovaCena`
  - obsah je hodnota atributu – zde `8`
- **textový uzel**
  - je to vždy koncový uzel
  - uzel nemá název (což je rozdíl od DOM, kde je název vždy `#text`)
  - obsahem je textový obsah XML elementů
    - odřádkování je vždy převedeno na jeden znak <LF> (`
`)
    - pokud se v XML objeví entity, jsou již nahrazeny skutečným textem
      - zde entita `<` je nahrazena znakem `<`
  - pozor na skutečnost, že samostatné textové uzly jsou také všechny uzly zajišťující odřádkování a formátování
    - je ale možné nastavit XSLT procesor tak, aby je ignoroval
- **komentář** (v příkladu není uveden)
  - je to vždy koncový uzel
  - uzel nemá název
  - obsahem je text mezi komentářovými závorkami
    - např. pro XML komentář `<!--poznámka-->`

      bude obsah `poznámka`

- **jmenný prostor** (v příkladu není uveden)
  - je připojen k odpovídajícímu elementu
  - není dítě elementu, ale element se chápe jako jeho rodič
  - je to vždy koncový uzel
  - název je prefix jmenného prostoru
  - obsah je URI jmenného prostoru
  - deklarace jmenného prostoru se dědí, takže je tento uzel přítomný i na potomcích rodičovského elementu
- **instrukce pro zpracování** (v příkladu není uvedena)
  - je to vždy koncový uzel
  - název je cíl instrukce
  - obsah je obsah instrukce uvedený za cílem instrukce a bílými znaky až do znaku pro konec instrukce `?>`
    - např. dříve uvedená instrukce pro zpracování

      ```
      <?xml-stylesheet type="text/xsl" href="prvni-styl.xsl"?>
      ```

      má název: `xml-stylesheet`

      a obsah: `type="text/xsl" href="prvni-styl.xsl"`

6.3.1.1. Výpis stromové struktury XML dokumentu

- stromovou strukturu XML dokumentu lze vypsat v textové podobě použitím speciálního stylu

  `http://www.cranesoftwrights.com/resources/showtree/showtree-20000610.xsl`

  ### Poznámka

  Při použití XSLT 2.0 je třeba v něm nejdříve opravit tři malé chyby ;-)

- pro XML dokument `jidlo-entita.xml`

  ```
  <jidlo>
    <nazev jednotkovaCena="8">jablka</nazev>
    <vaha>&lt; 2.5</vaha>
  </jidlo>
  ```

  dostaneme po příkazu:

  ```
  saxon jidlo-entita.xml showtree-opraveno.xsl
  ```

  výpis

```
SHOWTREE Stylesheet - http://www.CraneSoftwrights.com/resources/
Processor: SAXON 9.1.0.3 from Saxonica
1  Element 'jidlo':
1.1  Text (jidlo): {
  }
1.2  Element 'nazev' (jidlo):
1.2.A  Attribute 'jednotkovaCena': {8}
1.2.1  Text (jidlo,nazev): {jablka}
1.3  Text (jidlo): {
  }
1.4  Element 'vaha' (jidlo):
1.4.1  Text (jidlo,vaha): {< 2.5}
1.5  Text (jidlo): {
}
```

- z výpisu je zřejmé, že dříve uvedený obrázek stromové struktury odpovídá skutečnosti

6.3.2. Hierarchické vztahy mezi uzly – osy pohybu

Varování

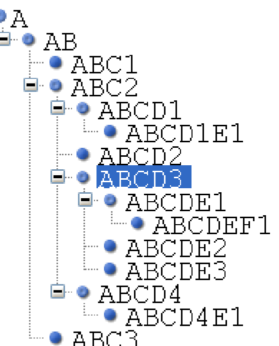
Zvládnutí následující terminologie je nutnou podmínkou pro používání XPath.

- mezi uzly lze přecházet po různých osách (*axises*)
  - XPath definuje celkem 13 os, z nichž se běžně používá až 12 (13. je `namespace`)
  - při používání os (pohybu po nich) vždy záleží na tom, ve kterém uzlu se nacházíme
- XML dokument `ukazka-os.xml`

```
<A>
  <AB>
    <ABC1>
    </ABC1>
    <ABC2>
      <ABCD1>
        <ABCD1E1>
        </ABCD1E1>
      </ABCD1>
      <ABCD2>
      </ABCD2>
      <ABCD3>
        <ABCDE1>
          <ABCDEF1>
          </ABCDEF1>
        </ABCDE1>
        <ABCDE2>
        </ABCDE2>
        <ABCDE3>
        </ABCDE3>
      </ABCD3>
      <ABCD4>
```

```
        <ABCD4E1>
        </ABCD4E1>
      </ABCD4>
    </ABC2>
    <ABC3>
    </ABC3>
  </AB>
</A>
```

se strukturou a vybraným elementem ABCD3



- vztahy uzlu ABCD3 (aktuálního uzlu) k ostatním uzlům:
  - self (aktuální)
    - označuje sebe sama

      ABCD3
  - parent (rodič)
    - jediný rodič

      ABC2
  - ancestor (předci)
    - všichni předci včetně rodiče v pořadí od rodiče výše

      ABC2, AB, A
  - ancestor-or-self (aktuální a předci)
    - aktuální a všichni jeho předci v pořadí od aktuálního výše

      ABCD3, ABC2, AB, A
  - child (děti)
    - všechny přímo vnořené uzly v pořadí, jak jsou v XML

      ABCDE1, ABCDE2, ABCDE3
  - descendant (potomci)
    - všichni potomci v pořadí, jak jsou v XML

`ABCDE1`, `ABCDEF1`, `ABCDE2`, `ABCDE3`

- `descendant-or-self` (aktuální a potomci)
  - aktuální a všichni jeho potomci v pořadí od aktuálního, jak jsou v XML

    `ABCD3`, `ABCDE1`, `ABCDEF1`, `ABCDE2`, `ABCDE3`
- `preceding-sibling` (předchozí sourozenci)
  - všechny v XML předcházející uzly, které jsou sourozencem aktuálního, v opačném pořadí než v XML

    `ABCD2`, `ABCD1`
- `preceding` (předchůdci)
  - všechny v XML předcházející uzly s výjimkou předků, v opačném pořadí než v XML

    `ABCD2`, `ABCD1E1`, `ABCD1`
- `following-sibling` (následní sourozenci)
  - všechny v XML následující uzly, které jsou sourozencem aktuálního, v pořadí jak jsou v XML

    `ABCD4`
- `following` (následníci)
  - všechny v XML následující uzly s výjimkou potomků, v pořadí jak jsou v XML

    `ABCD4`, `ABCD4E1`
- `attribute` (atributy)
  - atributy aktuálního uzlu

    zde žádné nejsou

6.3.3. Výrazy

- XPath výraz ve verzi 1.0 může vracet následující typy hodnot:
  - logickou hodnotu (`true` nebo `false`)
  - číslo (reálné nebo celočíselné)
  - řetězec
  - uzel z hierarchického stromu
- od verze 2.0 byla zavedena univerzální datová struktura, která všechny zmíněné typy zastřešuje
  - název je **posloupnost** (*sequence*)
  - její výskyt si nemusíme dlouho vůbec uvědomovat

- ♦ atomická hodnota posloupnosti je posloupnost o délce jedna neboli obsahující jeden prvek
- ♦ pak se posloupnost chová, jako kdyby byla nahrazena svým jediným prvkem
- ♦ např. vrací-li XPath výraz posloupnost, která obsahuje pouze jedno celé číslo, pracujeme s tímto výsledkem automaticky jako s celým číslem, bez toho, že bychom toto číslo museli před použitím z posloupnosti nějak získat

- prvky posloupnosti mohou být již zmíněné logické hodnoty, čísla, řetězce a uzly
  - ♦ je důležité, že obsah posloupnosti nemusí být typově jednotný
  - ♦ v jedné posloupnosti se mohou nalézat libovolné kombinace těchto datových typů
- posloupnost je uspořádaná – lze z ní lze získat prvky v pořadí, tj. pomocí indexu
  - ♦ první prvek v pořadí má index 1, poslední `last()`
- prvky posloupnosti mohou být duplicitní, tj. jejich hodnoty se mohou opakovat
- posloupnost je výsledek XPath výrazu
- skutečná (víceprvková) posloupnost vzniká nejpřirozeněji jako výběr uzlů ze stromu dokumentu
  - ♦ je ale možné posloupnost vytvořit uměle
- prázdná posloupnost se zapíše pomocí prázdných kulatých závorek `()`

■ v XSLT se XPath výrazy používají pro:

- výběr uzlů, které chceme zpracovat – vrací posloupnost vyhovujících uzlů

```
<xsl:apply-templates select="/jidlo/nazev"/>
```

- vytvoření vzoru pro porovnávání uzlu se šablonou – vrací `true` nebo `false`

```
<xsl:template match="nazev">
```

- vytvoření textu vkládaného do výstupního dokumentu (`pohyb-po-osach.xsl`)

```
<xsl:value-of select="following-sibling::vaha * attribute::jednotkovaCena"/>
```

  - ♦ zde je vidět, že součástí XPath výrazu mohou být i operátory – zde operátor násobení

6.3.3.1. Výrazy pro výběr části dokumentu pomocí cesty

■ příkazy pro výběr uzlů ze stromu dokumentu mají velký stupeň volnosti a výsledek záleží na mnoha okolnostech

■ první důležitou částí je výběr počátku místa hledání

- je třeba si uvědomit, že celý strom je v paměti, tzn naráz přístupný, takže můžeme začínat z jeho libovolného uzlu

■ možné počátky místa hledání

- **od kořene stromu**
  - použijeme absolutní cestu začínající `/`
  - tento způsob je možné použít téměř vždy, ale je málo častý
- **od aktuálního uzlu**
  - ten je určen z XSLT, například:
    - v šabloně je to uzel, pro který byla šablona aktivována
    - v cyklu je to právě zpracovávaný uzel
  - použijeme relativní cestu, která NEzačíná `/`
  - tento způsob se používá nejčastěji
    - je to přirozené zpracování dokumentu
    - je nejrychlejší (nejvýkonnější)
- **z libovolného přesně určeného místa**
  - je nutné použít speciální funkce XPath, které provedou přesun kamkoliv ve stromu, např. `id()`
  - pokud počátek stanovíme tímto způsobem, smí se dále používat jen relativní cesta

- cesta se může skládat z několika částí
  - mezi sebou se oddělují znakem `/` (jako v souborovém systému)
  - každá část cesty se může skládat z následujících komponent
    - identifikátoru osy
    - testu uzlu
    - predikátů

6.3.3.1.1. Identifikátory osy

- využívají se názvy, které byly uvedeny u hierarchického stromu, za nimiž následují dvě dvojtečky (`::`)
  - vrací se vždy posloupnost všech vyhovujících uzlů (může být prázdný)
    - uzly v posloupnosti jsou číslovány od 1 (nikoliv od 0)
- pohyby směrem
  - vzhůru – na předky
    - `parent::`, `ancestor::`, `ancestor-or-self::`
  - dolů – na potomky
    - `child::`, `descendant::`, `descendant-or-self::`

- doleva – na předcházející sourozence a jejich potomky
  - `preceding-sibling::`, `preceding::`
- doprava – na následující sourozence a jejich potomky
  - `following-sibling::`, `following::`
- speciální – na sebe a na atributy
  - `self::` a `attribute::`

Poznámka

Opačné uspořádání uzlů na osách nahoru a doleva má vliv jen na číslování pořadí uzlů. Výsledná množina je vždy uspořádána ve stejném pořadí jako v XML. Číslo tak má význam „vzdálenost od aktuálního uzlu“, kde `1` je nejblíže, a `last()` je nejdále.

`last()`	...	2	1	`self::`

- `last()` je funkce XPath, která vrací index posledního uzlu v posloupnosti

6.3.3.1.2. Testy uzlu

- z posloupnosti uzlů vrácených ze zvolené osy lze dále vybírat testem na název uzlu nebo jeho typ

  pro XML dokument `jedno-jidlo.xml`:

```
<jidlo>
  <nazev jednotkovaCena="8">jablka</nazev>
  <vaha>2.5</vaha>
</jidlo>
```

- nejčastější výběr je **názvem uzlu** (jménem elementu či atributu)
  - `/jidlo/child::nazev` – jdeme po ose `child::` a testujeme, zda je na ní uzel `nazev`

    vrací: `/jidlo[1]/nazev[1]`
  - `/jidlo/child::nazev/attribute::jednotkovaCena`

    vrací: `/jidlo[1]/nazev[1]/@jednotkovaCena`
    - `@jednotkovaCena` znamená „atribut `jednotkovaCena`“ – viz dále zkrácené zápisy
- často se používá zástupný znak `*` ve smyslu „libovolný název“
- pro výběr speciálních uzlů (mimo elementy a atributy) lze použít testy **typu uzlu**:
  - `comment()` – vybere všechny komentáře
  - `text()` – vybere všechny textové uzly
    - `/jidlo/child::*/child::text()`

vrací:

```
/jidlo[1]/nazev[1]/text()[1]
/jidlo[1]/vaha[1]/text()[1]
```

- `node()` – vybere všechny uzly bez ohledu na typ
  - ♦ `/jidlo/child::node()`

    vrací (textové uzly jsou odřádkování – viz výše):

    ```
    /jidlo[1]/text()[1]
    /jidlo[1]/nazev[1]
    /jidlo[1]/text()[2]
    /jidlo[1]/vaha[1]
    /jidlo[1]/text()[3]
    ```

6.3.3.1.3. Používané zkratky

- protože jsou zápisy pomocí identifikace osy a testu uzlů dlouhé, jsou pro nejčastější způsoby definovány zkratky
  - samotný název uzlu je považován za pohyb po ose dětí (`child::` není nutno psát)
    - ♦ `/jidlo/nazev`

      vrací: `/jidlo[1]/nazev[1]`
  - `attribute::` lze nahradit znakem `@`
    - ♦ `/jidlo/nazev/@jednotkovaCena`

      vrací: `/jidlo[1]/nazev[1]/@jednotkovaCena`
  - na aktuální uzel se lze odkázat tečkou `.` místo `self::node()`
  - na rodičovský uzel se lze odkázat dvojtečkou `..` místo `parent::node()`
    - ♦ `/jidlo/nazev/text()/../@jednotkovaCena`

      vrací: `/jidlo[1]/nazev[1]/@jednotkovaCena`
  - pro prohledávání potomků do libovolné hloubky lze použít místo:

    ```
    /descendant-or-self::node()/child::nazev
    ```

    zkrácený zápis

    ```
    //nazev
    ```

Varování

Tuto možnost je dobré nezneužívat. Prohledává a testuje všechny uzly, což může u dlouhých dokumentů velmi zdržovat.

6.3.3.1.4. Predikáty

- v každé části cesty lze použít jednoho nebo více predikátů – snižují počet vyhovujících uzlů
- zapisují se do hranatých závorek `[ ]`
- vyhodnocují se pro každý uzel, který dosud vyhovoval cestě
  - pokud je pro uzel predikát pravdivý, uzel zůstává v posloupnosti uzlů, jinak je vyřazen
- následující ukázky budou platné pro XML dokument `jidlo-4-ovoce.xml`

```
<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>
```

- výraz použitý v predikátu může vracet různé datové typy, podle kterých se pak vyhodnotí logická hodnota predikátu
  - **číselná hodnota** – číslo se chápe jako pozice uzlu v posloupnosti na dané ose
    - `/jidlo/ovoce[2]`

      vrací: `/jidlo[1]/ovoce[2]`
    - `/jidlo/ovoce[last()]` (funkce `last()` vrací číslo posledního uzlu v posloupnosti – viz později)

      vrací: `/jidlo[1]/ovoce[4]`
  - **množina uzlů** – výraz v predikátu je XPath výraz, který vrací posloupnost uzlů

    je-li posloupnost neprázdná, má predikát hodnotu `true`, jinak `false`
    - `/jidlo/ovoce/nazev[@jednotkovaCena]` – všechny uzly `nazev`, které mají atribut `jednotkovaCena`

      vrací:

```
/jidlo[1]/ovoce[1]/nazev[1]
/jidlo[1]/ovoce[2]/nazev[1]
/jidlo[1]/ovoce[3]/nazev[1]
/jidlo[1]/ovoce[4]/nazev[1]
```

- ♦ v XPath výrazu lze použít logické podmínky

  `/jidlo/ovoce/nazev[@jednotkovaCena < 20]` – všechny uzly `nazev`, které mají atribut `jednotkovaCena` s hodnotou menší než `20`

  vrací:

  ```
  /jidlo[1]/ovoce[1]/nazev[1]
  /jidlo[1]/ovoce[3]/nazev[1]
  ```

- ♦ pozor při zápisu konstantních hodnot řetězců

  – čísla lze zapsat jako `20` nebo `'20'`

  – řetězce je nutné vložit do apostrofů: `'jablka'` – bez apostrofů mají význam „název uzlu“

  `/jidlo/ovoce[nazev='jablka']` – všechny uzly `nazev`, které mají hodnotu `jablka`

  vrací:

  ```
  /jidlo[1]/ovoce[1]
  ```

- ♦ predikáty lze psát za sebe – pak se vyhodnocují zleva doprava – prakticky to znamená logický součin

  `/jidlo/ovoce[vaha <= 2][vaha > 1.0]` – všechny uzly `vaha`, které mají obsah v intervalu (1; 2>

  vrací:

  ```
  /jidlo[1]/ovoce[2]
  /jidlo[1]/ovoce[4]
  ```

- **množina uzlů a číselná hodnota** – z posloupnosti se vybere uzel s daným číslem

  - ♦ `/jidlo/ovoce[vaha <= 2][vaha > 1.0][1]` – z předchozí posloupnosti vyber první uzel

    vrací: `/jidlo[1]/ovoce[2]`

  - ♦ na pořadí predikátů u čísel záleží

    `/jidlo/ovoce[1][vaha <= 2][vaha > 1.0]` – vyber první uzel ovoce, jehož vaha je v intervalu (1; 2>

    taková podmínka není splněna, proto je hodnota predikátu `false`, což znamená, že je vrácena prázdná posloupnost

6.3.3.2. Závěrečný přehled možností a praktické pokusy

- pro praktické pokusy je vhodné nalézt nástroj, který umožní interaktivní praktické pokusy
  - nejvhodnější je editor oXygen, který však není bezplatný
  - v tutoriálu `http://zvon.org/xxl/XPathTutorial/General_cze/examples.html` lze provádět přímo pokusy s uvedenými příklady po zvolení „Otevřít příklad v XLabu".
    - je zde množství krátkých názorných příkladů
  - lze provést pokusy s `http://www.codinghorror.com/xpath/4.0/` popsaným na `http://www.codinghorror.com/blog/archives/000158.html`
    - nepodařilo se mi jej plně vyzkoušet
    - na přednastavený XML dokument funguje jen v MSIE
    - nelze zvolit vlastní XML dokument
  - použitelné výsledky (pod MSIE) dává XPath Expression Builder 3.0 dostupný na Portále jako `xpathexpr.zip`
    - rozbalte jej, v MSIE spusťte soubor `xpath.htm`
    - v řádce `XML Instance:` zvolte příslušný XML soubor
    - v řádce `XPath Expression` zadávejte XPath výraz
- následující ukázky budou platné pro XML dokument `jidlo-4-ovoce.xml`

```
<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>
```

- přehled nejčastěji používaných možností XPath výrazů
  - `/jidlo`

vybere kořenový element `jidlo`

- `/jidlo/ovoce`

  vybere všechny elementy `ovoce`, které jsou dětmi kořenového elementu `jidlo`

- `./nazev`

  vybere všechny elementy `nazev`, které jsou dětmi aktuálního uzlu

- `*`

  vybere všechny elementy, které jsou dětmi aktuálního uzlu

- `.`

  vybere aktuální uzel

- `..`

  vybere rodičovský uzel aktuálního uzlu

- `ovoce/*`

  vybere všechny elementy, které jsou dětmi elementu `ovoce`, který je dítětem aktuálního uzlu

- `text()`

  vybere všechny textové uzly, které jsou dětmi aktuálního uzlu

- `//text()`

  vybere všechny textové uzly v celém dokumentu

- `@jednotkovaCena`

  vybere atribut `@jednotkovaCena` aktuálního uzlu

- `//@jednotkovaCena`

  vybere všechny atributy `@jednotkovaCena` v celém dokumentu

- `ovoce/@cislo`

  vybere atribut `@cislo` elementu ovoce, který je dítětem aktuálního uzlu

- `@*`

  vybere všechny atributy aktuálního uzlu

- `//*[@*]`

  vybere všechny elementy dokumentu, které mají alespoň jeden atribut

- `ovoce[2]`

  vybere druhý element `ovoce`, který je dítětem aktuálního uzlu

- `ovoce[last()]]`

  vybere poslední element `ovoce`, který je dítětem aktuálního uzlu

- `*/vaha`

  vybere všechny elementy `vaha`, které jsou dětmi dětí aktuálního uzlu

- `/jidlo/ovoce[2]/vaha`

  vybere element `vaha` druhého elementu `ovoce` od kořenového uzlu

- `ovoce//text()`

  vybere všechny textové uzly elementu `ovoce`, který je dítětem aktuálního uzlu

- `//ovoce[vaha >= 2]/nazev`

  vybere všechny elementy `nazev`, které jsou dítětem elementu `ovoce`, za předpokladu, že element `vaha` má hodnotu 2 a více

  ### Varování

  Tento výraz (a další podobné následující výrazy) je nutné zapsat do XSLT souboru s nahrazením < za < a > za > např.: //ovoce[vaha >= 2]/nazev

- `/jidlo/ovoce[nazev[@jednotkovaCena >= 20]][1]`

  vybere první element `ovoce` ze všech elementů `ovoce`, jejichž `jednotkovaCena` >= 20

- `/jidlo/ovoce[1][nazev[@jednotkovaCena >= 20]]`

  vybere první element `ovoce`, který je v pořadí v XML dokumentu, za předpokladu, že jeho `jednotkovaCena` >= 20 (zde není splněno)

- `/jidlo/ovoce[nazev[text()='jablka']]/vaha` nebo

  `/jidlo/ovoce[nazev='jablka']/vaha`

  vybere element `vaha` takového elementu `ovoce`, jehož nazev je `jablka`

- `self::vaha`

  vybere aktuální uzel za předpokladu, že se jmenuje `vaha`

- `/jidlo/ovoce[nazev='grapefruity']/following::*[1]`

  vybere element `ovoce`, který je následující za elementem `ovoce` s názvem `grapefruity` (zde švestky sušené)

- `/jidlo/ovoce[nazev='grapefruity']/preceding-sibling::*[1]`

  vybere element `ovoce`, který je bezprostředně předcházející elementu `ovoce` s názvem `grapefruity` (zde banány)

- `/jidlo/ovoce[nazev='grapefruity']/preceding-sibling::*[last()]`

vybere element `ovoce`, který je prvním sourozencem v pořadí v XML dokumentu předcházející elementu `ovoce` s názvem `grapefruity` (zde jablka)

- `/jidlo/ovoce[nazev='grapefruity']/preceding::*[1]`

  vybere element `vaha`, který je bezprostředně předcházející elementu `ovoce` s názvem `grapefruity` (není to sourozenec, ale vnořený element ovoce banány)

6.3.4. Operátory

- tato část slouží jako ucelený přehled některých již použitých operátorů v XPath výrazech

6.3.4.1. Operátory pro práci s posloupností uzlů

- sjednocení – znak `|` (svislítko) – slouží ke spojení více XPath výrazů principiálně i zcela odlišných (není to `or`)

  - `/jidlo/ovoce[nazev='grapefruity'] | /jidlo/ovoce[nazev='jablka']`

    vrátí posloupnost dvou elementů `ovoce`

6.3.4.2. Logické operátory

- logický součet `or`

  - `/jidlo/ovoce[nazev='grapefruity' or nazev='jablka']`

    vrátí posloupnost dvou elementů `ovoce`, které se jmenují buď `grapefruity` nebo `jablka`

- logický součin `and`

  - `/jidlo/ovoce[nazev='grapefruity' and vaha<=1]`

    vrátí element `ovoce`, které se jmenuje `grapefruity` a váží méně než 1 kg

- negace – pomocí funkce `not()`

  - `/jidlo/ovoce[not(nazev='grapefruity')]`

    vrátí posloupnost elementů `ovoce`, které se nejmenují `grapefruity`

6.3.4.3. Relační operátory

- mají zápis a smyl známý z jiných jazyků

  `=` (porovnání), `!=`, `>`, `>=`, `<`, `<=`

Varování

Pozor při zápisu znaku `<` v XSLT souborech, které jsou ve skutečnosti XML dokumenty. Znak `<` je nutno zapisovat pomocí entity `<` Je vhodné zapisovat i znak `>` pomocí entity `>`

- tyto operátory mohou porovnávat i posloupnosti

- při porovnání posloupností je výraz vyhodnocován tak, že daná relace musí platit pro každý z prvků obou posloupností
- `(1, 2, 3) > (5, 6, 7)` vrací `false`, protože každé číslo z levé posloupnosti je menší než kterékoliv číslo z pravé posloupnosti
- `(1, 2, 3) > (5, 2, 7)` vrací `true`, protože 3 z levé posloupnosti je větší než 2 z pravé posloupnosti

■ pro zaručené porovnávání jen jednotlivých hodnot použijeme operátory `eq, ne, gt, ge, lt, le`

- Pozor – nejsou to přepisy pomocí entit – není před nimi `&` a za nimi `;`
- `1 ne 5` vrací `true` protože 1 se nerovná 5

6.3.4.4. Matematické operátory

■ opět mají zápis a smysl známý z jiných jazyků (+ – *)

■ pro dělení (reálné) je nutné použít `div` a pro zbytek po dělení `mod`.

6.3.5. Funkce

■ funkce se nejčastěji používají v predikátech

■ další použití je v případech, kdy XPath má vracet něco jiného než posloupnost uzlů, např. `sum()`, `count()`

- ty využijeme až při použití v XSLT

Poznámka

Následující přehled je stručný přehled nejpoužívanějších funkcí. V XPath 2.0 je k dispozici přes 150 funkcí. Podrobně viz `http://www.w3.org/TR/xpath-functions/`. Výčet všech funkcí bude uveden dále.

6.3.5.1. Funkce pro práci s uzly

■ `číslo last()` – vrací pozici posledního uzlu v posloupnosti

■ `číslo position()` – vrací pozici aktuálního uzlu v posloupnosti

■ `číslo count(posloupnost uzlů)` – vrací počet uzlů v posloupnosti

■ `řetězec local-name()` – vrací název aktuálního uzlu (bez případného prefixu)

■ `řetězec name()` – vrací název aktuálního uzlu

■ `řetězec namespace-uri()` – vrací URI adresu jmenného prostoru (ne prefix!) aktuálního uzlu

- poslední tři funkce mohou mít jako skutečný parametr posloupnost uzlů – pak se vrací jméno prvního uzlu

6.3.5.2. Řetězcové funkce

- konstantní hodnoty řetězce se uzavírají do apostrofů: `'ahoj'`
- `řetězec string(objekt libovolného typu)` – převede libovolný objekt na řetězec
  - většinou se nepoužívá, protože XPath konvertuje typy na string automaticky
- `řetězec concat(řetězec1, řetězec2, ...)` – spojí řetězce do jednoho
- `boolean starts-with(řetězec, hledaný řetězec)` – test, zda řetězec začíná hledaným řetězcem
- `boolean contains(řetězec, hledaný řetězec)` – test, zda řetězec obsahuje hledaný řetězec
- `řetězec substring-before(řetězec, hledaný řetězec)` – vrací podřetězec před nalezeným řetězcem
  - `substring-before('12345', '34')` vrátí `'12'`
- `řetězec substring-after(řetězec, hledaný řetězec)` – vrací podřetězec za nalezeným řetězcem
  - `substring-after('12345', '34')` vrátí `'5'`
- `řetězec substring(řetězec, index začátku, počet znaků)` – vrací podřetězec
  - první znak v řetězci má index 1
  - pokud není `počet znaků` uveden, vrací se celý zbytek řetězce
- `číslo string-length(řetězec)` – vrací počet znaků řetězce
  - `//*[string-length(name(.))=4]` – vrací všechny uzly jejichž název je čtyřznakový
  - pokud je funkce zavolána bez parametrů, převede aktuální uzel na řetězec, ze kterého je vrácena délka
- `řetězec normalize-space(řetězec)` – odstraní z řetězce přebytečné bílé znaky
  - odřádkování a tabulátory převede na mezery, počáteční a koncové mezery se odříznou, vícenásobné mezery uvnitř jsou nahrazeny jednou
- `řetězec translate(řetězec, původní znaky, náhrada znaků)` – všechny znaky prvního řetězce, které jsou obsaženy ve druhém parametru funkce jsou nahrazeny znaky na stejné pozici ze třetího parametru; není-li tam, je znak vynechán
  - `translate('12 300,-', ', -', '.')` vrátí `12300.`
  - `translate('ahoj', 'a', 'A')` vrátí `Ahoj`

6.3.5.3. Logické funkce

- `boolean boolean(objekt libovolného typu)` – převede objekt na booleovskou hodnotu

- číslo – 0 je `false`, ostatní `true`
- posloupnost uzlů – prázdná je `false`, neprázdná `true`
- řetězec – prázdný je `false`, neprázdný `true`

- `boolean not(boolean)` – negace
- `true true()` – vrací hodnotu `true`
- `false false()` – vrací hodnotu `false`

6.3.5.4. Funkce pro práci s čísly

- `číslo number(objekt libovolného typu)` – převede objekt na číslo
  - boolean – `false` je 0, `true` je 1
  - řetězec – číslo se převádí, jinak vrací `'NaN'`
  - posloupnost uzlů – převedou se na řetězec a ten se dále převádí
  - volání bez parametru – převede aktuální uzel
- `číslo sum(posloupnost uzlů)` – vrací součet hodnot uložených v posloupnosti uzlů
  - z každého uzlu se vezme řetězec, ten se převede na číslo a čísla se sečtou
- `číslo floor(číslo)` – zaokrouhlení na celé číslo dolů
- `číslo ceiling(číslo)` – zaokrouhlení na celé číslo nahoru
- `číslo round(číslo)` – zaokrouhlení na nejbližší celé číslo

6.3.5.5. Funkce pro práci s datumem a časem

- používají datový typ `xs:duration`
- úplný formát je `Pn1Yn2Mn3DTn4Hn5Mn6S`, kde `P` je úvodní znak, `T` je znak oddělující datum od času a `n1` až `n6` jsou počty roků, měsíců, dnů, hodin, minut, sekund
- používají se i zkrácené formáty, např.: `Pn1Yn2M` nebo `PTn4Hn5M`
- dále jsou používány datové typy `xs:dateTime()`, `xs:date()` a `xs:time()` – viz XSD schémata
- `xs:date('2010-05-02') - xs:date('2010-04-01')` vrací `P31D`

6.3.6. Výčet všech funkcí a operátorů z XSLT 2.0

- tento přehled je užitečný tehdy, hledáme-li nějakou funkcionalitu a nevíme, zda je již poskytována
  - jména funkcí jsou velmi významová, navíc jsou přehledně členěna do sekcí, takže vyhledat funkci, která by mohla splňovat naše požadavky, je snadné
  - dalším krokem by pak bylo, podívat se na podrobný popis této funkce na:

`http://www.w3.org/TR/xpath-functions/`

- uváděné funkce lze rozdělit do dvou skupin podle uvedeného prefixu
  - první skupina má prefix `fn:` (např. funkce `fn:abs()`) a takto označené funkce používáme dle běžných zvyklostí
    - prefix `fn:` není běžně používán
  - druhá skupina má prefix `op:` (např. `op:date-equal()`) a jsou to **operátory**
    - jejich definice vypadá přesně jako definice funkcí (těch s prefixem `fn:`), což je zpočátku značně matoucí
      - smysl je, že ukazují, jaké operace (relační, aritmetické, ...) jsou předdefinovány pro speciální datové typy
    - použití je z prvního pohledu kryptické – např. jméno `date-equal` nelze jako řetězec nikde použít
    - je třeba se podívat do popisu a tam zjistíte, jaké **náhradní operátory** lze místo nich použít
    - uvedeném příkladě lze použít operátory `eq` nebo `ne` nebo `le` nebo `ge`
    - např. porovnání dvou datumů je:

```
xs:date('2010-06-30') lt xs:date('2010-06-29')
```

    - výraz vrací `false`, protože datum `2010-06-30` je větší než `2010-06-29`
- *Functions and Operators on Numerics*
  - *Operators on Numeric Values*
    - `op:numeric-add()`
    - `op:numeric-subtract()`
    - `op:numeric-multiply()`
    - `op:numeric-divide()`
    - `op:numeric-integer-divide()`
    - `op:numeric-mod()`
    - `op:numeric-unary-plus()`
    - `op:numeric-unary-minus()`
  - *Comparison Operators on Numeric Values*
    - `op:numeric-equal()`
    - `op:numeric-less-than()`
    - `op:numeric-greater-than()`

- *Functions on Numeric Values*
  - `fn:abs()`
  - `fn:ceiling()`
  - `fn:floor()`
  - `fn:round()`
  - `fn:round-half-to-even()`

- *Functions on Strings*
  - *Functions to Assemble and Disassemble Strings*
    - `fn:codepoints-to-string()`
    - `fn:string-to-codepoints()`
  - *Equality and Comparison of Strings*
    - `fn:compare()`
    - `fn:codepoint-equal()`
  - *Functions on String Values*
    - `fn:concat()`
    - `fn:string-join()`
    - `fn:substring()`
    - `fn:string-length()`
    - `fn:normalize-space()`
    - `fn:normalize-unicode()`
    - `fn:upper-case()`
    - `fn:lower-case()`
    - `fn:translate()`
    - `fn:encode-for-uri()`
    - `fn:iri-to-uri()`
    - `fn:escape-html-uri()`
  - *Functions Based on Substring Matching*
    - `fn:contains()`
    - `fn:starts-with()`

- fn:ends-with()
- fn:substring-before()
- fn:substring-after()

- *String Functions that Use Pattern Matching*
  - fn:matches()
  - fn:replace()
  - fn:tokenize()

- *Functions on anyURI*
  - fn:resolve-uri()
- *Functions and Operators on Boolean Values*
  - *Additional Boolean Constructor Functions*
    - fn:true()
    - fn:false()
  - *Operators on Boolean Values*
    - op:boolean-equal()
    - op:boolean-less-than()
    - op:boolean-greater-than()
  - *Functions on Boolean Values*
    - fn:not()
- *Functions and Operators on Durations, Dates and Times*
  - *Two Totally Ordered Subtypes of Duration*
    - xs:yearMonthDuration()
    - xs:dayTimeDuration()
  - *Comparison Operators on Duration, Date and Time Values*
    - op:yearMonthDuration-less-than()
    - op:yearMonthDuration-greater-than()
    - op:dayTimeDuration-less-than()
    - op:dayTimeDuration-greater-than()
    - op:duration-equal()

- op:dateTime-equal()
- op:dateTime-less-than()
- op:dateTime-greater-than()
- op:date-equal()
- op:date-less-than()
- op:date-greater-than()
- op:time-equal()
- op:time-less-than()
- op:time-greater-than()
- op:gYearMonth-equal()
- op:gYear-equal()
- op:gMonthDay-equal()
- op:gMonth-equal()
- op:gDay-equal()

- *Component Extraction Functions on Durations, Dates and Times*
  - fn:years-from-duration()
  - fn:months-from-duration()
  - fn:days-from-duration()
  - fn:hours-from-duration()
  - fn:minutes-from-duration()
  - fn:seconds-from-duration()
  - fn:year-from-dateTime()
  - fn:month-from-dateTime()
  - fn:day-from-dateTime()
  - fn:hours-from-dateTime()
  - fn:minutes-from-dateTime()
  - fn:seconds-from-dateTime()
  - fn:timezone-from-dateTime()
  - fn:year-from-date()

- ♦ `fn:month-from-date()`
- ♦ `fn:day-from-date()`
- ♦ `fn:timezone-from-date()`
- ♦ `fn:hours-from-time()`
- ♦ `fn:minutes-from-time()`
- ♦ `fn:seconds-from-time()`
- ♦ `fn:timezone-from-time()`

- *Arithmetic Operators on Durations*
  - ♦ `op:add-yearMonthDurations()`
  - ♦ `op:subtract-yearMonthDurations()`
  - ♦ `op:multiply-yearMonthDuration()`
  - ♦ `op:divide-yearMonthDuration()`
  - ♦ `op:divide-yearMonthDuration-by-yearMonthDuration()`
  - ♦ `op:add-dayTimeDurations()`
  - ♦ `op:subtract-dayTimeDurations()`
  - ♦ `op:multiply-dayTimeDuration()`
  - ♦ `op:divide-dayTimeDuration()`
  - ♦ `op:divide-dayTimeDuration-by-dayTimeDuration()`
- *Timezone Adjustment Functions on Dates and Time Values*
  - ♦ `fn:adjust-dateTime-to-timezone()`
  - ♦ `fn:adjust-date-to-timezone()`
  - ♦ `fn:adjust-time-to-timezone()`
- *Arithmetic Operators on Durations, Dates and Times*
  - ♦ `op:subtract-dateTimes()`
  - ♦ `op:subtract-dates()`
  - ♦ `op:subtract-times()`
  - ♦ `op:add-yearMonthDuration-to-dateTime()`
  - ♦ `op:add-dayTimeDuration-to-dateTime()`
  - ♦ `op:subtract-yearMonthDuration-from-dateTime()`

- ♦ op:subtract-dayTimeDuration-from-dateTime()
- ♦ op:add-yearMonthDuration-to-date()
- ♦ op:add-dayTimeDuration-to-date()
- ♦ op:subtract-yearMonthDuration-from-date()
- ♦ op:subtract-dayTimeDuration-from-date()
- ♦ op:add-dayTimeDuration-to-time()
- ♦ op:subtract-dayTimeDuration-from-time()

- *Functions Related to QNames*
  - *Additional Constructor Functions for QNames*
    - fn:resolve-QName()
    - fn:QName()
  - *Functions and Operators Related to QNames*
    - op:QName-equal()
    - fn:prefix-from-QName()
    - fn:local-name-from-QName()
    - fn:namespace-uri-from-QName()
    - fn:namespace-uri-for-prefix()
    - fn:in-scope-prefixes()
- *Operators on base64Binary and hexBinary*
  - *Comparisons of base64Binary and hexBinary Values*
    - op:hexBinary-equal()
    - op:base64Binary-equal()
- *Operators on NOTATION*
  - op:NOTATION-equal()
- *Functions and Operators on Nodes*
  - fn:name()
  - fn:local-name()
  - fn:namespace-uri()
  - fn:number()

- `fn:lang()`
- `op:is-same-node()`
- `op:node-before()`
- `op:node-after()`
- `fn:root()`

■ *Functions and Operators on Sequences*

- *General Functions and Operators on Sequences*
  - `fn:boolean()`
  - `op:concatenate()`
  - `fn:index-of()`
  - `fn:empty()`
  - `fn:exists()`
  - `fn:distinct-values()`
  - `fn:insert-before()`
  - `fn:remove()`
  - `fn:reverse()`
  - `fn:subsequence()`
  - `fn:unordered()`
- *Functions That Test the Cardinality of Sequences*
  - `fn:zero-or-one()`
  - `fn:one-or-more()`
  - `fn:exactly-one()`
- *Equals, Union, Intersection and Except*
  - `fn:deep-equal()`
  - `op:union()`
  - `op:intersect()`
  - `op:except()`
- *Aggregate Functions*
  - `fn:count()`

- `fn:avg()`
    - `fn:max()`
    - `fn:min()`
    - `fn:sum()`
  - *Functions and Operators that Generate Sequences*
    - `op:to()`
    - `fn:id()`
    - `fn:idref()`
    - `fn:doc()`
    - `fn:doc-available()`
    - `fn:collection()`
- *Context Functions*
  - `fn:position()`
  - `fn:last()`
  - `fn:current-dateTime()`
  - `fn:current-date()`
  - `fn:current-time()`
  - `fn:implicit-timezone()`
  - `fn:default-collation()`
  - `fn:static-base-uri()`

Kapitola 7. XSLT 1

7.1. Šablony

- jsou základem každého stylu
- obecný tvar je:

```
<xsl:template match="XPath vzor">
  tělo šablony
</xsl:template>
```

- `XPath vzor` je takový výraz, který používá jen osy pro přechod nebo `//`
  - nepoužívají se predikáty
- tělo šablony definuje, jak se budou části XML dokumentu zpracovávat
  - v těle šablony se používají
    - další instrukce XSLT, nejčastěji:
      - `<xsl:apply-templates>` – zpracování další šablony
      - `<xsl:value-of>` – výstup textového obsahu, který je vytvořen z parametru XPath výrazu
      - `<xsl:text>` – výstup textového obsahu (viz dále)
    - elementy výsledného dokumentu (nejčastěji HTML nebo XML)
  - XSLT procesor na začátku práce načte celý XML dokument do paměti do stromové reprezentace (viz dříve)
    - tento strom prochází od kořene v pořadí, v jakém jsou elementy v XML (procházení do hloubky)
    - je-li nalezena šablona odpovídající uzlu ve stromu, začne se obsah uzlu podle uvedených příkazů vypisovat na výstup
    - další potomci uzlu vybraného šablonou nejsou zpracováváni
      - chceme-li je zpracovat, musíme použít `<xsl:apply-templates>`
        - toto neplatí pro koncové uzly bez šablony, které jsou vypisovány na výstup
      - je zodpovědností šablony zpracovat zvolený uzel a všechny jeho potomky
    - chceme-li vložit textový obsah nějakého elementu nebo jeho atributů či podelementů na výstup, použijeme `<xsl:value-of>`
    - pro vyvolávání šablon není podstatné fyzické umístění jednotlivých šablon v souboru XSLT stylu
- pro XML dokument `jidlo-2-ovoce.xml`

```
<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>
</jidlo>
```

- XSLT styl `sablony-zaklad.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

    <xsl:output method="html"
        encoding="UTF-8"/>

  <xsl:template match="/">
    <html>
      <head></head>
      <body>
        <xsl:apply-templates select="jidlo/ovoce"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="ovoce">
    <xsl:value-of select="@cislo"/>
    <xsl:text>. </xsl:text>
    <xsl:apply-templates select="nazev"/>
    <!--   <xsl:apply-templates/>   -->
  </xsl:template>

  <xsl:template match="nazev">
    <xsl:value-of select="."/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="@jednotkovaCena"/>
    <xsl:text>, </xsl:text>
  </xsl:template>
</xsl:stylesheet>
```

- vypíše:

```
<html>
  <head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=UTF-8">
  </head>
```

```
   <body>1. jablka 10, 2. banány 25, </body>
</html>
```

- šablona `match="/"`
  - zajistí výpis HTML značek na začátku a na konci
  - pak je volána šablona pro `select="jidlo/ovoce"`
- šablona `match="ovoce"`
  - atribut `cislo` je vypsán pomocí `<xsl:value-of select="@cislo"/>`
  - oddělovací tečku a mezeru lze bez odřádkování zapsat pomocí `<xsl:text>. </xsl:text>`
  - pak je volána šablona pro `select="nazev"`
- šablona `match="nazev"`
  - vypíše obsah elementu `nazev` pomocí `<xsl:value-of select="."/>`
  - všechny další příkazy jsou již známé
- pokud XSLT styl změníme v šabloně `<xsl:template match="ovoce">` na

```
<!-- <xsl:apply-templates select="nazev"/>  -->
     <xsl:apply-templates/>
```

- vypíše se:

```
<html>
   <body>1.
      jablka 10,
      2.5
      2.
      banány 25,
      2

   </body>
</html>
```

- protože
  - `<xsl:apply-templates/>` má význam `<xsl:apply-templates select="node()"/>` a znamená hledání šablon pro všechny děti aktuálního uzlu (zde `nazev` a `vaha` a textové uzly odřádkování)
    - hledání se netýká atributů, které nejsou považovány za děti
  - XSLT procesory mají v soběuloženo několik zabudovaných (implicitních) šablon – viz též dále
    - ty např. na výstup kopírují obsah veškerých textových uzlů, které nebyly zpracovány žádnou definovanou šablonou

- stejný výstup

```
<html>
   <body>1. jablka 10, 2. banány 25, </body>
</html>
```

dostaneme jednodušeji při použití jen jedné šablony `sablony-zaklad-jedna.xsl`:

```
  <xsl:template match="ovoce">
    <xsl:value-of select="@cislo"/>
    <xsl:text>. </xsl:text>
    <xsl:value-of select="nazev"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="nazev/@jednotkovaCena"/>
    <xsl:text>, </xsl:text>
  </xsl:template>
```

7.1.1. Režimy zpracování šablon

- při běžném použití zpracuje jedna šablona konkrétní uzel právě jednou
- občas požadujeme opakované zpracování jiným způsobem
  - tiskne se nejdříve stručná informace (obsah) a pak detailní
- každá šablona může mít pomocí atributu `mode` definován režim

```
<xsl:template match="výraz" mode="název režimu">
```

- tento `název režimu` se pak použije při volání šablony

```
<xsl:apply-templates select="jiný výraz" mode="název režimu"/>
```

- pro XML dokument `jidlo-2-ovoce.xml`

```
<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>
</jidlo>
```

- XSLT styl `sablony-mode.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
      version="2.0">

    <xsl:output method="html"
        encoding="UTF-8"/>

  <xsl:template match="/">
    <html>
      <body>
        <p>Na skladě jsou:
        <xsl:apply-templates select="jidlo/ovoce" mode="seznam"/>
        </p>
        <p>s následujícími parametry:<br/>
          <xsl:apply-templates select="jidlo/ovoce" mode="detail"/>
        </p>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="ovoce" mode="seznam">
    <xsl:value-of select="nazev"/>
    <xsl:text>, </xsl:text>
  </xsl:template>

  <xsl:template match="ovoce" mode="detail">
    <xsl:value-of select="nazev"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="vaha"/>
    <xsl:text> kg po </xsl:text>
    <xsl:value-of select="nazev/@jednotkovaCena"/>
    <xsl:text> Kč/kg</xsl:text>
    <br/>
  </xsl:template>
</xsl:stylesheet>
```

- vypíše:

```
<html>
   <body>
      <p>Na skladě jsou:
         jablka, banány,
      </p>
      <p>s následujícími parametry:
         jablka 2.5 kg po 10 Kč/kg<br>banány 2 kg po 25 Kč/kg<br></p>
   </body>
</html>
```

7.1.2. Priority šablon

- pokud připravujeme rozsáhlejší styl, může se stát, že jeden uzel může být zpracováván pomocí více šablon

- typicky se to stává, pokud je výraz v `match=""` příliš obecný, např. `*` nebo `@*`

- tento konflikt lze řešit tím, že každá šablona má svoji prioritu
  - tu lze explicitně nastavit pomocí atributu `priority="celé číslo od 1"`
  - při nalezení dvou (či více) vyhovujících šablon se vybere ta s vyšší prioritou
  - každá šablona má svoji implicitní prioritu za jasně definovaných pravidel (viz `www.w3.org/TR/xslt20/`)
    - platí jednoduché pravidlo – čím obecnější šablona, tím nižší priorita
- nejsme-li si s prioritami jisti, je vhodné je explicitně určit (a dodržet přitom stejné pravidlo)
- pro XML dokument `jidlo-2-ovoce.xml`

```
<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>
</jidlo>
```

- XSLT styl `sablony-priority.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

    <xsl:output method="html"
        encoding="UTF-8"/>

  <xsl:template match="/">
    <html>
      <body>
        <xsl:apply-templates select="jidlo/ovoce"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="ovoce" priority="2">
    <xsl:text>druh: </xsl:text>
    <xsl:value-of select="nazev"/>
    <xsl:text>, </xsl:text>
  </xsl:template>

  <xsl:template match="*" priority="1">
    <xsl:value-of select="."/>
```

```
  </xsl:template>
</xsl:stylesheet>
```

- vypíše:

```
<html>
   <body>druh: jablka, druh: banány, </body>
</html>
```

- při změně na `<xsl:template match="*" priority="3">` vypíše:

```
<html>
   <body>
      jablka
      2.5

      banány
      2

   </body>
</html>
```

7.1.3. Zabudované (implicitní) šablony

- již bylo zmíněno, že pokud po zavolání `<xsl:apply-templates>` nenalezne XSLT procesor ve stylu žádnou vyhovující šablonu, použije některou ze zabudovaných šablon
  - jedná se o tyto šablony:

```
<!-- Šablona pro postupné zpracování celého dokumentu -->
<xsl:template match="*|/">
  <xsl:apply-templates/>
</xsl:template>

<!-- Kopírování textových uzlů -->
<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>

<!-- Ignorování komentářů a instrukcí pro zpracování -->
<xsl:template match="comment()|processing-instruction()"/>
```

  - u šablony `<xsl:template match="text()|@*">` je výraz `"text()|@*"` proto, že atributy nejsou považovány za děti a musíme proto o ně žádat zvlášť
- pokud do svého stylu některou šablonu zkopírujeme a předefinujeme, bude platit ta naše
  - předefinování často znamená, že je šablona prázdná, např.:

```
<!-- zákaz kopírování textových uzlů -->
<xsl:template match="text()" >
</xsl:template>
```

- pokud toto provedeme u textových uzlů, musíme zajistit výpis jejich hodnot v našich upravených šablonách

■ pro XML dokument `jidlo-2-ovoce.xml`

```
<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>
</jidlo>
```

■ XSLT styl `zabudovana-sablona.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

    <xsl:output method="html"
        encoding="UTF-8"/>

  <xsl:template match="/">
    <html>
      <body>
        <xsl:apply-templates select="jidlo/ovoce"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="nazev">
    <xsl:value-of select="."/>
  </xsl:template>

  <xsl:template match="vaha">
    <xsl:value-of select="."/>
  </xsl:template>

<!-- zákaz kopírování textových uzlů
  <xsl:template match="text()" >
  </xsl:template>
 -->
</xsl:stylesheet>
```

- vypíše:

```
<html>
   <body>
      jablka
      2.5

      banány
      2

   </body>
</html>
```

- při změně na:

```
  <!-- zákaz kopírování textových uzlů -->
  <xsl:template match="text()" >
  </xsl:template>
```

- vypíše:

```
<html>
   <body>jablka2.5banány2</body>
</html>
```

7.2. Tvorba výstupního dokumentu

- XSLT umožňuje vytvořit HTML, XHTML i XML výstup (TXT výstupu se tato problematika týká jen okrajově)
  - je nutné umět vytvořit všechny části těchto dokumentů, tj. značky, jejich obsah, atributy, komentáře apod.

7.2.1. Generování elementů a jejich obsahu

- je to ten nejjednodušší případ
- všechen text v šablonách, který není uvnitř instrukcí `<xsl:>` je kopírován na výstup
  - kopírují se i úvodní mezery, koncové mezery a odřádkování
  - nechceme-li kopírovat tyto bílé znaky, lze požadovaný text uzavřít do

    ```
    <xsl:text>ahoj</xsl:text>
    ```

    - na výstup se pak zkopíruje jen `ahoj` bez jakýchkoliv bílých znaků před a po
  - potřebujeme-li odřádkovat, použije se jen znak <LF> zapsaný pomocí číselné znakové entity `
`

    ```
    <xsl:text>ahoj&#xA;nazdar</xsl:text>
    ```

- generujeme-li texty na základě XPath výrazů, použijeme `<xsl:value-of select="XPath výraz"/>`
  - to dosud nejčastěji jen zkopírovalo obsah elementu či atributu

    ```
    <xsl:value-of select="vaha"/>
    ```

  - je ale možné použít libovolný XPath výraz (často složený z volání mnoha funkcí) (`xpath-nasobeni.xsl`)

    ```
    <xsl:template match="ovoce">
      <xsl:text>Cena: </xsl:text>
      <xsl:value-of select="vaha * nazev/@jednotkovaCena"/>
    </xsl:template>
    ```

    vypíše: `Cena: 25`

7.2.1.1. Výpisy posloupností

- v mnoha případech je užitečná možnost získat na výstupu nějakou posloupnost (čísel, řetězců, ...), která není součástí původního XML dokumentu

Poznámka

Praktické použití posloupností bude ukázáno při použití skupin příkazem `<xsl:for-each-group>`.

7.2.1.1.1. Základní použití

- v příkazu `<xsl:value-of>` se zapíše jako hodnota `select=""` požadovaná posloupnost
  - jednotlivé položky (čísla nebo řetězce nebo i nody) jsou odděleny čárkami
    - ve výpisu pak budou jednotlivé položky odděleny jednou mezerou
  - v případě čísel nebo řetězců se tyto vypíší přesně tak jak jsou uvedeny na výstup
  - v případě nodů se vypíše jejich obsah
- pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `posloupnost-zaklad.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

    <xsl:output method="html"
        encoding="UTF-8"/>

  <xsl:template match="/">
    <html>
      <body>
        <xsl:apply-templates select="jidlo"/>
      </body>
    </html>
  </xsl:template>
```

```
  <xsl:template match="jidlo">
    <xsl:value-of select="1,2,'Ahoj'"/>
    <xsl:value-of select="'x','y','z',''"/>
    <xsl:value-of select="ovoce[2]/nazev,ovoce[1]/nazev"/>
  </xsl:template>
</xsl:stylesheet>
```

- vypíše:

```
<html>
   <body>1 2 Ahojx y z banány jablka</body>
</html>
```

  - všimněte si prázdného řetězce v `select="'x','y','z',''"`, který způsobí výpis mezery mezi `z banány`

- toto základní použití nemá příliš velký smysl pro posloupnosti čísel a řetězců, protože stejný výstup lze dostat i jednodušeji, např. pomocí: (`posloupnost-zaklad-jednoduse.xsl`)

```
  <xsl:template match="jidlo">
    <xsl:text>1 2 Ahojx y z </xsl:text>
    <xsl:value-of select="ovoce[2]/nazev,ovoce[1]/nazev"/>
  </xsl:template>
```

- význam pro nody už je zřejmý, neboť v posloupnosti nodů si můžeme pomocí indexů libovolně zvolit pořadí
  - zde je druhý node vypsán jako první

7.2.1.1.2. Operátor `to` – průběžné číslování

- je-li posloupnost číselná, lze ji vygenerovat mnohem snadněji než výčtem použitím operátoru `to`
  - operátor `to` nelze použít pro řetězce
- změněný XSLT styl `posloupnost-to.xsl`

```
  <xsl:template match="jidlo">
    <xsl:value-of select="1 to 5"/>
<!--  <xsl:value-of select="'a' to 'd'"/>  NELZE -->
  </xsl:template>
```

- vypíše:

```
1 2 3 4 5
```

7.2.1.1.3. Funkce `reverse()` – otočení pořadí

- posloupnost (popsanou výčtem nebo pomocí operátoru `to`) lze vypsat v opačném pořadí použitím funkce `reverse()`

- tato možnost je užitečná zejména pro výpis nodů v obráceném pořadí

- změněný XSLT styl `posloupnost-reverse.xsl`

```
<xsl:template match="jidlo">
  <xsl:value-of select="reverse(1 to 5)"/>
  <xsl:value-of select="reverse(ovoce/nazev)"/>
</xsl:template>
```

- vypíše:

```
5 4 3 2 1banány jablka
```

7.2.2. Generování hodnot atributů

- instrukci `<xsl:value-of select="XPath výraz"/>` nelze použít při generování obsahu atributů výstupního dokumentu
- v tomto případě XPath výraz vkládáme do složených závorek `{ }` a použijeme jej v hodnotě atributu
  - takto lze vytvořit hodnotu libovolného atributu výstupního dokumentu
  - lze tak vytvořit i několik atributů v XSLT stylu – viz dále
- pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `obsah-atributu.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

    <xsl:output method="html"
        encoding="UTF-8"/>

  <xsl:template match="/">
    <html>
      <body>
        <xsl:apply-templates select="jidlo/ovoce"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="ovoce">
    <p><a name="navesti{@cislo}">
    <xsl:value-of select="nazev"/>
    </a></p>
  </xsl:template>

</xsl:stylesheet>
```

- vypíše:

```
<html>
   <body>
      <p><a name="navesti1">jablka</a></p>
      <p><a name="navesti2">banány</a></p>
   </body>
</html>
```

7.2.3. Generování elementů s dynamickým jménem

- předchozí dvě možnosti stačí pro generování většiny běžných HTML dokumentů
  - v nich existuje pevná (tj. předem známá) množina elementů a atributů
  - ty lze staticky zapsat do XSLT stylu
- v komplikovaných stylech můžeme generovat i název elementu
- to lze provést XSLT instrukcí `<xsl:element name="výraz">`
  - `name` představuje jméno vytvářeného elementu a `výraz` může být libovolný text doplněný XPath výrazem
  - jako obsah elementu se pak použijí již známé konstrukce
- pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `jmeno-elementu.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

    <xsl:output method="html"
        encoding="UTF-8"/>

  <xsl:template match="/">
    <html>
      <body>
        <xsl:apply-templates select="jidlo/ovoce"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="ovoce">
    <xsl:element name="h{@cislo}">
      <xsl:value-of select="nazev"/>
      <hr/>
    </xsl:element>
  </xsl:template>

</xsl:stylesheet>
```

- vypíše (Pozor: Struktura HTML dokumentu není logická – příklad je jen ukázka generování dynamicky pojmenovaných elementů.):

```
<html>
   <body>
      <h1>jablka
         <hr>
      </h1>
      <h2>banány
         <hr>
      </h2>
   </body>
</html>
```

7.2.4. Generování atributů s dynamickým jménem

- v konstantních i v dynamicky generovaných jménech elementů lze vytvářet i atributy s dynamickým obsahem
  - použijeme `<xsl:attribute name="jméno">hodnota</xsl:attribute>`
    - tato instrukce se musí objevit bezprostředně za instrukcí `<xsl:element>` nebo za počáteční značkou (viz dále)
    - `name` je jméno atributu
    - `hodnota` je hodnota atributu a může být i generována pomocí XPath výrazu
- pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `dynamicky-atribut.xsl`

```
...

  <xsl:template match="ovoce">
    <xsl:element name="h{@cislo}">
      <xsl:attribute name="id">
        <xsl:value-of select="generate-id()"/>
      </xsl:attribute>
      <xsl:value-of select="nazev"/>
      <hr/>
    </xsl:element>
  </xsl:template>

</xsl:stylesheet>
```

  - v šabloně je použita XPath funkce `generate-id()`, která vygeneruje unikátní ID
- vypíše:

```
<html>
   <body>
      <h1 id="d1e3">jablka
         <hr>
      </h1>
```

```
      <h2 id="d1e12">banány
         <hr>
      </h2>
   </body>
</html>
```

- trochu praktičtější případ, který ukazuje generování proměnného atributu u pevně dané značky
  - je vytvářen element typu `<a href="">`
- pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `atribut-href.xsl`

```
...

  <xsl:template match="ovoce">
    <p><a>
      <xsl:attribute name="href">
        <xsl:text>http://www.heroutovy-sady.cz/</xsl:text>
        <xsl:value-of select="nazev"/>
        <xsl:text>/index.html</xsl:text>
      </xsl:attribute>
      <xsl:value-of select="nazev"/>
    </a></p>
  </xsl:template>

</xsl:stylesheet>
```

- vypíše:

```
<html>
   <body>
      <p><a ►
href="http://www.heroutovy-sady.cz/jablka/index.html">jablka</a></p>
      <p><a ►
href="http://www.heroutovy-sady.cz/ban%C3%A1ny/index.html">banány</a></p>
   </body>
</html>
```

7.2.5. Opakované vkládání skupiny atributů

- často se vyskytne požadavek opakovaně vkládat stejnou skupinu atributů na různá místa výstupního dokumentu
- používá se instrukce `<xsl:attribute-set name="název">` pro vytvoření skupiny
- pro přidání této skupiny k libovolnému elementu se jako atribut použije instrukce `xsl:use-attribute-sets="název"`
  - hodnotou `xsl:use-attribute-sets="název"` může být i několik názvů skupin atributů oddělených mezerou
    - pak se použijí všechny atributy

- atribut `xsl:use-attribute-sets="název"` lze použít i u elementů XSLT jako `<xsl:element>` a `<xsl:copy>`
- pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `skupina-atributu.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

  <xsl:output method="html" encoding="UTF-8"/>

  <xsl:template match="/">
    <html>
      <body>
        <table border="1">
          <xsl:apply-templates select="jidlo/ovoce"/>
        </table>
      </body>
    </html>
  </xsl:template>

  <xsl:attribute-set name="stred">
    <xsl:attribute name="align">center</xsl:attribute>
    <xsl:attribute name="valign">middle</xsl:attribute>
  </xsl:attribute-set>

  <xsl:attribute-set name="pozadi">
    <xsl:attribute name="bgcolor">blue</xsl:attribute>
  </xsl:attribute-set>

  <xsl:template match="ovoce">
    <tr>
      <td xsl:use-attribute-sets="stred pozadi">
        <xsl:value-of select="@cislo"/>
      </td>
      <td bgcolor="yellow">
        <xsl:value-of select="nazev"/>
      </td>
      <td xsl:use-attribute-sets="stred">
        <xsl:value-of select="vaha"/>
      </td>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```

- vypíše:

```
<html>
   <body>
      <table border="1">
         <tr>
```

```
            <td align="center" valign="middle" bgcolor="blue">1</td>
            <td bgcolor="yellow">jablka</td>
            <td align="center" valign="middle">2.5</td>
         </tr>
         <tr>
            <td align="center" valign="middle" bgcolor="blue">2</td>
            <td bgcolor="yellow">banány</td>
            <td align="center" valign="middle">2</td>
         </tr>
      </table>
   </body>
</html>
```

7.2.6. Generování komentářů

- používá se instrukce <xsl:comment>text komentáře</xsl:comment>
  - text komentáře může být i generovaný pomocí XPath
- pro známý XML dokument jidlo-2-ovoce.xml XSLT styl komentare.xsl

```
...

  <xsl:template match="ovoce">
    <xsl:comment><xsl:value-of select="nazev"/> došly</xsl:comment>
  </xsl:template>

</xsl:stylesheet>
```

- vypíše:

```
<html>
   <body>
      <!--jablka došly-->
      <!--banány došly-->
   </body>
</html>
```

7.2.7. Výstup vyhrazených znaků

7.2.7.1. Znaky < nebo > nebo &

- ve speciálních situacích potřebujeme do výstupu vložit přímo znaky < nebo > nebo &
- protože XSLT styl je XML dokument, nemůžeme je zapsat přímo, ale jen jako jejich znakové entity < nebo > nebo &
- toto funguje dle očekávání pouze při výstupu do textu (znaky-lt-gt.xsl)

```
<xsl:output method="text" encoding="UTF-8"/>

<xsl:template match="/">
  <xsl:text>1 &lt; 5 &gt; 3 &amp; </xsl:text>
</xsl:template>
```

vypíše:

```
1 < 5 > 3 &
```

- při výstupu do HTML nebo XML jsou ale znakové entity na výstupu ponechány (`znaky-lt-gt.xsl`)

```
<xsl:output method="html" encoding="UTF-8"/>

<xsl:template match="/">
  <xsl:text>1 &lt; 5 &gt; 3 &amp; </xsl:text>
</xsl:template>
```

vypíše:

```
1 &lt; 5 &gt; 3 &amp;
```

- toto chování XSLT procesoru lze pro instrukce `<xsl:text>` a `<xsl:value-of>` vypnout
  - použijeme atribut těchto elementů `disable-output-escaping="yes"`
    - ten pro danou instrukci náhradu vypne
    - Pozor: Je to nebezpečná možnost, kdy výstup nemusí být syntakticky správný HTML či XML dokument.
- XSLT styl `znaky-lt-gt.xsl`

```
<xsl:output method="html" encoding="UTF-8"/>

<xsl:template match="/">
 <xsl:text disable-output-escaping="yes">1 &lt; 5 &gt; 3 &amp; </xsl:text>
</xsl:template>
```

vypíše

```
1 < 5 > 3 &
```

7.2.7.2. Entita ` `

- tato entita se používá velice často v tabulkách HTML, aby byly jinak prázdné buňky správně orámovány
- problém je v tom, že ` ` není jako entita pro XSLT implicitně definována
  - nelze tedy napsat: `<xsl:text> </xsl:text>`
- lze použít zápis (`nbsp.xsl`)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="2.0">

  <xsl:output method="html" encoding="UTF-8"/>

  <xsl:template match="/">
    <html>
      <body>
        <table border="1">
          <tr>
            <td>plná</td>
            <td>
<xsl:text disable-output-escaping="yes">&amp;nbsp;</xsl:text>
            </td>
            <td>plná</td>
          </tr>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

vypíše

```
<html>
   <body>
      <table border="1">
         <tr>
            <td>plná</td>
            <td> </td>
            <td>plná</td>
         </tr>
      </table>
   </body>
</html>
```

- elegantnější způsob je využití číselné znakové entity ` `
  - to je kód nedělitelné mezery, kterou XSTL procesor převede na ` `
  - problém je, že se toto číslo (magická konstanta) špatně pamatuje
  - používá se proto trik s definováním této entity
- XSLT styl `nbsp-definice.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE xsl:stylesheet[
<!ENTITY nbsp " ">
]>
```

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="2.0">

  <xsl:output method="html" encoding="UTF-8"/>

  <xsl:template match="/">
    <html>
      <body>
        <table border="1">
          <tr>
            <td>plná</td>
            <td> </td>
            <td>plná</td>
          </tr>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

vypíše

```
<html>
   <body>
      <table border="1">
         <tr>
            <td>plná</td>
            <td> </td>
            <td>plná</td>
         </tr>
      </table>
   </body>
</html>
```

7.2.8. Odstranění bílých znaků ze vstupního dokumentu

- vstupní XML dokument obsahuje často odřádkování a odsazování elementů
  - to po načtení občas způsobuje problémy s nechtěnými textovými uzly, které obsahují jen bílé znaky
- XML dokument `jidlo-2-ovoce.xml`

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

- zpracován XSLT stylem `strip-space.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

  <xsl:output method="html"
        encoding="UTF-8"/>

  <xsl:template match="/">
    <html>
      <body>
         <xsl:value-of select="."/>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

- vypíše

```
<html>
   <body>

      jablka
      2.5


      banány
      2


   </body>
</html>
```

- přebytečné textové uzly jen s bílými znaky lze při načítání automaticky odstranit
  - použije se instrukce `<xsl:strip-space elements="seznam elementů"/>`
  - pak se odstraní textové uzly, jejichž rodiče jsou uvedeny v `seznam elementů`
    - v seznamu se nejčastěji používá jen `elements="*"` ve významu „všechny elementy“
    - pokud chceme uvést jen některé konkrétní elementy, pak jejich výčet v seznamu oddělíme mezerou

```
<xsl:strip-space elements="jidlo ovoce"/>
```

- po přidání instrukce `<xsl:strip-space elements="*"/>` do stylu `strip-space.xsl`

```
...
  <xsl:output method="html"
      encoding="UTF-8"/>

  <xsl:strip-space elements="*"/>
...
  <xsl:template match="/">
...
```

- vypíše

```
<html>
   <body>jablka2.5banány2</body>
</html>
```

7.3. Formátování čísel

- potřebujeme-li vypisovat čísla dle českých zvyklostí (desetinný oddělovač je čárka, oddělovač řádů pak mezera), využijeme funkci `format-number()`
  - formát vypisovaného čísla je určen pomocí formátovacího řetězce, který se předává jako druhý parametr
  - jako třetí parametr se předává jméno stylu, ve kterém jsou určeny významy speciálních znaků
    - významy speciálních znaků se určí pomocí instrukce `<xsl:decimal-format>`, přičemž pro naše podmínky stačí změnit tři znaky:
      - `decimal-separator` – oddělovač desetinné části – znak `,`
      - `grouping-separator` – oddělovač řádů – pevná mezera s kódem ` `
      - `zero-digit` – nevýznamová nula – zde použit znak `0`
- pro libovolný XML dokument XSLT styl `formatovani.xsl`
  - formát `' ###,00 Kč'` stanoví, že:
    - řády budou odděleny po třech číslicích (`###`) znakem s kódem ` `
    - za desetinnou částí budou max. dvě číslice případně doplněné nevýznamovými nulami (`00`) – delší desetinná část se zaokrouhlí
    - za každé číslo se vypíše řetězec `" Kč"`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="2.0">

  <xsl:output method="html"
```

```
    encoding="UTF-8"/>

  <xsl:decimal-format name="cesky-styl"
     decimal-separator=","
     grouping-separator=" "
     zero-digit="0"/>

  <xsl:template match="/">
    <html>
      <body>
        <xsl:value-of
          select="format-number(-123456789.912,
                               ' ###,00 Kč',
                               'cesky-styl')"/>
        <xsl:text>&#xA;</xsl:text>
        <xsl:value-of
          select="format-number(6789.1,
                               ' ###,00 Kč',
                               'cesky-styl')"/>
        <xsl:text>&#xA;</xsl:text>
        <xsl:value-of
          select="format-number(89.,
                               ' ###,00 Kč',
                               'cesky-styl')"/>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

- vypíše:

```
<html>
   <body>-123 456 789,91 Kč
      6 789,10 Kč
      89,00 Kč
   </body>
</html>
```

7.3.1. Formát pro automatické číslování

- speciální případ formátování čísla poskytuje instrukce `<xsl:number>`
  - ta se používá pro automatické číslování např. kapitol, obrázků, apod.
- instrukce má mnoho způsobů použití, pro základní účely stačí:

```
<xsl:number value="výraz" format="formát výpisu"/>
```

  - `výraz` je XPath výraz, který musí poskytnout číselnou hodnotu
  - `formát výpisu` má tyto možnosti:

- ♦ `1` – arabské číslice
- ♦ `001` – arabské číslice na tři místa s nevýznamovými nulami
- ♦ `I` – římské číslice velkými písmeny
- ♦ `i` – římské číslice malými písmeny
- ♦ `A` – velká písmena (základ soustavy je 26)
- ♦ `a` – malá písmena (základ soustavy je 26)
- ve formátu mohou být další doplňující znaky, které se vypíší tak, jak jsou – typicky tečka a mezera za číslem

■ pro libovolný XML dokument XSLT styl `auto-cislovani.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="2.0">

  <xsl:output method="text"
    encoding="UTF-8"/>

  <xsl:template match="/">
    <xsl:number value="5" format="1. "/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:number value="15" format="001"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:number value="4" format="I -"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:number value="8" format="i"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:number value="5" format="* A"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:number value="27" format="a"/>
  </xsl:template>

</xsl:stylesheet>
```

■ vypíše:

```
5.
015
IV -
viii
* E
aa
```

7.4. Podmíněné zpracování

- podmíněné zpracování lze řešit pomocí predikátů v XPath výrazech
- další možností jsou větvení pomocí XSLT instrukcí

7.4.1. Podmíněný příkaz

- používá se instrukce `<xsl:if test="boolean výraz">`
  - má-li `boolean výraz` hodnotu `true` provedou se vnořené příkazy
- pozor na skutečnost, že se jedná o neúplnou podmínku – příkaz nemá část „else“
  - toto se řeší použitím `<xsl:choose>`
- pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `if.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="2.0">

  <xsl:output method="html" encoding="UTF-8"/>

  <xsl:template match="/">
    <html>
      <body>
        <table border="1">
          <xsl:apply-templates select="//ovoce"/>
        </table>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="ovoce">
    <xsl:if test="position() mod 2 = 0">
      <tr bgcolor="yellow">
        <xsl:apply-templates select="nazev | vaha"/>
      </tr>
    </xsl:if>
    <xsl:if test="position() mod 2 = 1">
      <tr>
        <xsl:apply-templates select="nazev | vaha"/>
      </tr>
    </xsl:if>
  </xsl:template>

  <xsl:template match="nazev">
    <td>
      <xsl:apply-templates/>
    </td>
  </xsl:template>
```

```
  <xsl:template match="vaha">
    <td align="center">
      <xsl:apply-templates/>
    </td>
  </xsl:template>

</xsl:stylesheet>
```

- generování každé řádky (`<xsl:template match="ovoce">`) a každé položky řádky (`<xsl:template match="nazev">` a `<xsl:template match="vaha">`) je zde v samostatné šabloně
  - ♦ styl je přehlednější a jednotlivé sloupce mohou být vypsány rozdílně (váha je centrovaná)
  - ♦ ovšem položky budou v řádce v tom pořadí, v jakém byly v XML dokumentu
    - – nezáleží na pořadí `nazev` a `vaha` v `<xsl:apply-templates select="nazev | vaha"/>`
- položky řádky (`nazev` a `vaha`) pak pro výpis hodnoty využívají implicitní šablonu
- pro identifikaci sudé řádky se používá výraz "`position() mod 2 = 0`", což je další užitečná funkce XPath udávající pozici uzlu v seznamu

■ vypíše tabulku ovocí, kde každá sudá řádka je podbarvena žlutě:

```
<html>
   <body>
      <table border="1">
         <tr>
            <td>jablka</td>
            <td align="center">2.5</td>
         </tr>
         <tr bgcolor="yellow">
            <td>banány</td>
            <td align="center">2</td>
         </tr>
      </table>
   </body>
</html>
```

7.4.2. Podmíněný příkaz pomocí XPath výrazu

■ XPath 2.0 přináší možnost úplné podmínky

- díky ní lze snadněji řešit příklady podobné předchozímu
- úplná podmínka má syntaxi: `if (podmínka) then výraz-true else výraz-false`
- podmínka musí být vždy úplná – nestačí jen `if then`

■ pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `xpath-if-else.xsl`

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

```
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

    <xsl:output method="html" encoding="UTF-8"/>

    <xsl:template match="/">
      <html>
        <body>
          <table border="1">
            <xsl:apply-templates select="//ovoce"/>
          </table>
        </body>
      </html>
    </xsl:template>

    <xsl:template match="/">
      <html>
        <body>
          <table border="1">
            <xsl:apply-templates select="//ovoce"/>
          </table>
        </body>
      </html>
    </xsl:template>

    <xsl:template match="ovoce">
        <tr bgcolor="{if (position() mod 2 = 0)
                      then 'yellow' else 'white'}">
          <xsl:apply-templates select="vaha | nazev"/>
        </tr>
    </xsl:template>

    <xsl:template match="*">
      <td>
        <xsl:apply-templates/>
      </td>
    </xsl:template>

  </xsl:stylesheet>
```

- oproti předchozímu příkladu je zde několik změn:
  - atribut `bgcolor` je vždy nastaven (buď na `yellow` nebo na `white`)
  - pro výpis hodnoty názvu a váhy je použita společná šablona, což ovšem brání vypsat jednotlivé sloupce rozdílně

■ vypíše tabulku ovocí, kde každá sudá řádka je podbarvena žlutě:

```
<html>
   <body>
      <table border="1">
         <tr bgcolor="white">
            <td>jablka</td>
```

```
            <td>2.5</td>
         </tr>
         <tr bgcolor="yellow">
            <td>banány</td>
            <td>2</td>
         </tr>
      </table>
   </body>
</html>
```

7.4.3. Podmíněné větvení

- umožňuje rozdělit zpracování do několika větví
  - není to *switch*, známý z programovacích jazyků, který testuje jen jednou na začátku
    - testy musí být uvedeny u každé větve
    - analogie příkazů else-if z programovacích jazyků
- zápis je:

```
<xsl:choose>
  <xsl:when test="podmínka">
    příkazy
  </xsl:when>
  <xsl:when test="podmínka">
    příkazy
  </xsl:when>
  ...
  <xsl:otherwise>
    příkazy
  </xsl:otherwise>
</xsl:choose>
```

- postupně se procházejí jednotlivé větve a pokud vyhovuje podmínka, pak se provedou
  - po provedení jakékoliv větve se `<xsl:choose>` opouští
  - nevyhovuje-li podmínka u žádné z větví `<xsl:when>`, provede se větev `<xsl:otherwise>`
- tím lze snadno nahradit chybějící konstrukci úplné podmínky if-else, kdy se uvede pouze jedna větev `<xsl:when>`

```
<xsl:choose>
  <xsl:when test="podmínka">
    příkazy
  </xsl:when>
  <xsl:otherwise>
    příkazy
  </xsl:otherwise>
</xsl:choose>
```

■ pro známý XML dokument `jidlo-2-ovoce.xml` trochu elegantnější (bez dvojitého testu) XSLT styl `choose.xsl`

```
...
  <xsl:template match="ovoce">
    <xsl:choose>
      <xsl:when test="position() mod 2 = 0">
        <tr bgcolor="yellow">
          <xsl:apply-templates select="vaha | nazev"/>
        </tr>
      </xsl:when>
      <xsl:otherwise>
      <tr>
        <xsl:apply-templates select="nazev | vaha"/>
      </tr>
    </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

...
</xsl:stylesheet>
```

■ vypíše zcela stejnou tabulku ovocí, jako předchozí příklad

7.5. Iterativní zpracování

■ nad každým seznamem uzlů (tzn. v libovolné šabloně) lze provést zpracování cyklem `<xsl:for-each select="výraz">`

■ obsah elementu `<xsl:for-each>` tvoří šablonu, která se zpracovává pro každý uzel seznamu (tj. prvek posloupnosti `výraz`)

- uzly ke zpracování se vybírají známým `select`
- pořadí uzlů je stejné, jaké je v XML dokumentu
- uvnitř `<xsl:for-each>` je aktuální uzel právě zpracovávaný uzel

■ je třeba si uvědomit, že cyklus a šablona se chovají podobně, ale nejsou stejné!

■ pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `for-each-sloupce.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="2.0">

  <xsl:output method="html" encoding="UTF-8"/>

  <xsl:template match="/">
    <html>
      <body>
        <table border="1">
```

```
          <xsl:apply-templates select="jidlo"/>
        </table>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="jidlo">
    <tr>
      <xsl:for-each select="ovoce">
        <td>
          <xsl:value-of select="@cislo"/>
        </td>
      </xsl:for-each>
    </tr>
    <tr>
      <xsl:for-each select="ovoce">
        <td>
          <xsl:value-of select="nazev"/>
        </td>
      </xsl:for-each>
    </tr>
    <tr>
      <xsl:for-each select="ovoce">
        <td>
          <xsl:value-of select="vaha"/>
        </td>
      </xsl:for-each>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```

- vypíše tabulku ovocí, kde každé ovoce je v jednom sloupci, nikoliv v řádce:

```
<html>
   <body>
      <table border="1">
         <tr>
            <td>1</td>
            <td>2</td>
         </tr>
         <tr>
            <td>jablka</td>
            <td>banány</td>
         </tr>
         <tr>
            <td>2.5</td>
            <td>2</td>
         </tr>
      </table>
   </body>
</html>
```

7.5.1. Ladicí výpisy pomocí cyklu

- pokud ladíme styl, mohou se nám hodit ladicí výpisy např. seznamu uzlů, který je předán do šablony
- pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `ladici-vypisy.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="2.0">

  <xsl:output method="text" encoding="UTF-8"/>

  <xsl:template match="/">
    <xsl:for-each select="/jidlo/ovoce">
      <xsl:value-of select="name()"/>
      <xsl:text>[</xsl:text>
      <xsl:value-of select="position()"/>
     <xsl:text>]&#xA;</xsl:text>    <!-- odřádkování -->
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

  - příkaz `<xsl:for-each>` můžeme přesunout např. do jakékoliv šablony
    - pak by měl nejspíše (podle okolností) tvar `<xsl:for-each select="*">`
  - později bude uvedeno elegantnější řešení
- vypíše:

```
ovoce[1]
ovoce[2]
```

7.5.2. Cyklus pomocí posloupnosti

- instrukce `<xsl:for-each>` nemusí iterovat jen přes seznam nodů
  - je možné iterovat i přes posloupnost
- pro libovolný XML dokument XSLT styl `for-each-posloupnost.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="2.0">

  <xsl:output method="html" encoding="UTF-8"/>

  <xsl:template match="/">
    <html>
      <body>
        <table border="1">
```

```
          <tr>
            <xsl:for-each select="1 to 5">
                <td>
                  <xsl:value-of select="."/>
                </td>
            </xsl:for-each>
          </tr>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

- vypíše tabulku s pěti očíslovanými sloupci:

```
<html>
   <body>
      <table border="1">
         <tr>
            <td>1</td>
            <td>2</td>
            <td>3</td>
            <td>4</td>
            <td>5</td>
         </tr>
      </table>
   </body>
</html>
```

7.5.3. Cyklus pomocí posloupnosti řetězců a vnořený cyklus

- instrukce `<xsl:for-each>` může být zanořena do jiného cyklu
- posloupností nemusí být jen celá čísla, ale např. i posloupnost řetězců
- pro libovolný XML dokument XSLT styl `for-each-posloupnost2.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="2.0">

  <xsl:output method="html" encoding="UTF-8"/>

  <xsl:template match="/">
    <html>
      <body>
        <table border="1">
        <xsl:for-each select="'Po','Ut','St'">
          <tr>
```

```
            <!-- nazev dne -->
            <td><xsl:value-of select="."/></td>
            <xsl:for-each select="1 to 2">
              <td>
                <xsl:value-of select="."/>
              </td>
            </xsl:for-each>
          </tr>
        </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

- vypíše tabulku se třemi řádkami, kde řádka začíná názvem dne a pak následují dva očíslované sloupce:

```
<html>
   <body>
      <table border="1">
         <tr>
            <td>Po</td>
            <td>1</td>
            <td>2</td>
         </tr>
         <tr>
            <td>Ut</td>
            <td>1</td>
            <td>2</td>
         </tr>
         <tr>
            <td>St</td>
            <td>1</td>
            <td>2</td>
         </tr>
      </table>
   </body>
</html>
```

7.5.4. Kdy vnořený cyklus nelze použít – změna kontextu

- s kombinací šablon a cyklů je třeba zacházet opatrně, protože nejsou dovoleny všechny kombinace
  - ve `<xsl:for-each>`, kde iterujeme přes posloupnost, nesmí být jiný `<xsl:for-each>` (nebo instrukce `<xsl:apply-templates>`) s atributem select do XML souboru
  - ani nepřímo ve volané šabloně!
- ve skutečnosti lze vnořovat libovolně, ale je potřeba dát pozor na změnu kontextu
  - místo, kde se právě ve stromu dokumentu nalézáme je jeden kontext a druhý kontext může být v právě procházené posloupnosti

- kontext lze převést (předat) pomocí proměnné, což bude ukázáno na příkladě následujícím po ukázce problému.

- pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `for-each-posloupnost-select-chyba.xsl` podobný předchozímu příkladu
  - měl by ke každému dni z posloupnosti vypsat názvy všech ovocí
  - uvnitř cyklu `<xsl:for-each select="'Po','Út','St'">` je kontext přepnut na posloupnost dní
    - pokud se ve vnořeném cyklu `<xsl:for-each select=" jidlo/ovoce">` odvoláváme relativní cestou na XML dokument, není to možné
    - takto zapsaná relativní cesta se váže ke kontextu posloupnosti dní, nikoliv ke XML dokumentu

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="2.0">

  <xsl:output method="html" encoding="UTF-8"/>

  <xsl:template match="/">  <!-- řádka, kde je hlášena chyba -->
    <html>
      <head></head>
      <body>
        <table border="1">
          <xsl:for-each select="'Po','Út','St'">
            <tr>
              <td>
                <xsl:value-of select="."/>
              </td>
<!-- tato konstrukce neni mozna -->
              <xsl:for-each select="jidlo/ovoce">
                <td>
                  <xsl:value-of select="./nazev"/>
                </td>
              </xsl:for-each>

            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

- při pokusu o spuštění vypíše

```
Error at xsl: template on line 8 column 27 of
for-each-posloupnost-select-chyba.xsl:
  XPTY0020: Axis step child::element(jidlo, xs:anyType)
```

```
    cannot be used here:
      the context item is an atomic value
  Failed to compile stylesheet. 1 error detected.
```

7.5.4.1. Řešení problému vnořených cyklů

- příklad s předáním kontextu umíme řešit s pomocí lokální proměnné (viz později)
- XSLT styl `for-each-posloupnost-select.xsl`
  - princip řešení je, že použijeme lokální proměnnou `seznamOvoci`, do které uložíme posloupnost uzlů
  - ve vnitřním cyklu `<xsl:for-each>` bude tato proměnná uvedena v `select`, čímž se předá správný kontext

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

  <xsl:output method="html"
              encoding="UTF-8"/>

  <xsl:template match="/">
    <xsl:variable name="seznamOvoci" select="/jidlo/ovoce"/>
    <html>
      <head></head>
      <body>
        <table border="1">
          <xsl:for-each select="'Po','Ut','St'">
            <tr>
              <!-- nazev dne -->
              <td>
                <xsl:value-of select="."/>
              </td>

              <!-- předání kontextu jako posloupnosti uzlů -->
              <xsl:for-each select="$seznamOvoci">
                <td>
                  <xsl:value-of select="./nazev"/>
                </td>
              </xsl:for-each>

            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

- vypíše

```
<html>
   <head>
      <meta http-equiv="Content-Type"
       content="text/html; charset=UTF-8">
   </head>
   <body>
      <table border="1">
         <tr>
            <td>Po</td>
            <td>jabka</td>
            <td>banány</td>
         </tr>
         <tr>
            <td>Ut</td>
            <td>jabka</td>
            <td>banány</td>
         </tr>
         <tr>
            <td>St</td>
            <td>jabka</td>
            <td>banány</td>
         </tr>
      </table>
   </body>
</html>
```

7.5.5. Cyklus pomocí XPath výrazu

- pokud vrací XPath výraz seznam nodů, je možné přes tento seznam provést iteraci
- syntaxe je: `for $proměnná in seznam return výraz`
- pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `xpath-cyklus.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="2.0">

  <xsl:output method="html" encoding="UTF-8"/>

  <xsl:template match="/">
    <html>
      <body>
        <table border="1">
          <xsl:for-each select="//ovoce">
            <tr>
              <td>
                <xsl:value-of select="nazev"/>
              </td>
              <td>
                <xsl:value-of select="vaha"/>
```

```
              </td>
              <td>
                <xsl:value-of select="vaha
                     * nazev/@jednotkovaCena"/>
              </td>
            </tr>
          </xsl:for-each>
          <tr bgcolor="red">
            <td colspan="2">
              <xsl:value-of select="'Celková cena:'"/>
            </td>
            <td>
              <xsl:value-of
                   select="sum(for $x in //ovoce
                        return $x/vaha
                              * $x/nazev/@jednotkovaCena)"/>
            </td>
          </tr>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

- příkaz `for $x in //ovoce` znamená, že proměnná `$x` bude nabývat postupně hodnoty jednotlivých uzlů `ovoce`
  - funkce `sum()` sčítá jednotlivé hodnoty

- vypíše:

```
<html>
   <body>
      <table border="1">
         <tr>
            <td>jablka</td>
            <td>2.5</td>
            <td>25</td>
         </tr>
         <tr>
            <td>banány</td>
            <td>2</td>
            <td>50</td>
         </tr>
         <tr bgcolor="red">
            <td colspan="2">Celková cena:</td>
            <td>75</td>
         </tr>
      </table>
   </body>
</html>
```

7.5.6. Cyklus pomocí XPath výrazu a posloupnosti

- cyklus `for` pomocí XPath nemusí iterovat jen přes seznam nodů
  - je možné iterovat i přes posloupnost
- pro libovolný XML dokument XSLT styl `xpath-cyklus-posloupnost.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="2.0">

  <xsl:output method="text"
              encoding="UTF-8"/>

  <xsl:template match="/">
    <xsl:value-of select="for $i in (1 to 5)
          return concat($i, ': ', $i * $i, '&#xA;')"/>
  </xsl:template>
</xsl:stylesheet>
```

- příkaz `for $i in (1 to 5)` znamená, že proměnná `$i` bude nabývat postupně hodnoty od 1 do 5
  - funkce `concat()` sloučí jednotlivé řetězce
    - `$i` – hodnota proměnné `i`
    - `': '` – oddělovač
    - `$i * $i` – výpočet druhé mocniny
    - `'
'` – odřádkování
- vypíše:

```
1: 1
2: 4
3: 9
4: 16
5: 25
```

7.6. Chybové zprávy

- občas potřebujeme dát uživateli nějakou zprávu
  - většinou se jedná o chybové zprávy, které jsou vypisovány vždy na konzoli nehledě na příkaz pro směr výstupu
- používáme příkaz `<xsl:message>text zprávy</xsl:message>`
- použijeme-li atribut `terminate="yes"`, pak příkaz

```
<xsl:message terminate="yes">text zprávy</xsl:message>
```

vypíše text zprávy a okamžitě ukončí zpracování stylu s příslušným chybovým hlášením

- pro libovolný XML dokument XSLT styl `message.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="2.0">

  <xsl:output method="html" encoding="UTF-8"/>

  <xsl:template match="/">
    <html>
      <body>
        zpracováno správně
      </body>
      <xsl:message>nějaký problém</xsl:message>
<!--  <xsl:message terminate="yes">vážný problém</xsl:message> -->
    </html>
  </xsl:template>
</xsl:stylesheet>
```

spuštěný příkazem vypíše na terminál:

```
saxon -o chyba.html pr07.xml pr49.xsl
nějaký problém
```

a vytvoří soubor `chyba.html` s obsahem:

```
<html>
   <body>
      zpracováno správně

   </body>
</html>
```

- pokud změníme šablonu na:

```
...

  <xsl:template match="/">
    <html>
      <body>
        zpracováno správně
      </body>
<!--  <xsl:message>nějaký problém</xsl:message> -->
      <xsl:message terminate="yes">vážný problém</xsl:message>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

pak se po spuštění vypíše na terminál:

```
d:\xslt\pokusy>saxon -o chyba.html pr07.xml pr49.xsl
vážný problém
Processing terminated by xsl:message at line 14 in pr49.xsl
```

a soubor `chyba.html` bude mít nulovou velikost

Kapitola 8. XSLT 2

8.1. Proměnné a parametry

- obojí slouží pro uchování údajů, které vícenásobně používáme
- jsou netypované, pokud typ explicitně neurčíme

8.1.1. Proměnné

- jmenují se sice proměnné (*variable*) ale ve skutečnosti jsou to konstanty
  - jejich hodnotu nelze po prvním nastavení dále měnit
- proměnné jsou globální – pro celý styl, nebo lokální – platné jen v šabloně, ve které jsou definovány
  - lokální proměnná zastiňuje globální proměnnou stejného jména
    - snažíme se, aby k těmto konfliktům nedocházelo
- proměnná může obsahovat výsledek libovolného XPath výrazu, tj. posloupnost (v ní booleovské či číselné hodnoty, řetězce, seznam uzlů)
  - v proměnné může být dokonce i část vstupního či výstupního dokumentu
- na již definovanou proměnnou se dále odkazujeme jako `$mojePromenna`
  - proměnnou lze samozřejmě využívat opakovaně
  - potřebujeme-li použít hodnotu proměnné jako hodnotu námi připravovaného atributu (uzavřenou do uvozovek jako řetězec), použijeme `"{$mojePromenna}"` – viz též dříve
- hodnotu definované proměnné zapíšeme do výstupního souboru jako

```
<xsl:value-of select="$mojePromenna"/>
```

8.1.1.1. Způsoby definice a naplnění proměnné

- proměnnou lze definovat jedním ze tří způsobů

1. **Přiřazení řetězce do proměnné** – dva způsoby
   - `<xsl:variable name="jméno" select="'řetězcová hodnota'"/>`
     - řetězec `řetězcová hodnota` musí být uveden v apostrofech
       - bez nich by byla `řetězcová hodnota` chápána jako výraz – viz dále
   - `<xsl:variable name="jméno">řetězcová hodnota</xsl:variable>`
     - zde se apostrofy neuvádějí – pokud jsou uvedeny, jsou součástí řetězce
2. **Přiřazení jednoduchých datových typů do proměnné**

- čísla zapisujeme tak, jak jsou
- booleovské hodnoty pomocí XPath funkce `true()` nebo `false()`

```
<xsl:variable name="logickaPromenna" select="true()"/>
<xsl:variable name="celeCislo" select="5"/>
<xsl:variable name="realneCislo" select="3.14"/>

<xsl:template match="/">
  <xsl:value-of select="if ($logickaPromenna = false())
    then 'splněno' else $logickaPromenna"/>
  <xsl:text>&#xA;</xsl:text>
  <xsl:value-of select="if ($celeCislo = 5)
    then 'splněno' else 'nesplněno'"/>
  <xsl:text>&#xA;</xsl:text>
  <xsl:value-of select="if ($realneCislo >= 4.0)
    then 'splněno' else 'nesplněno'"/>
  <xsl:text>&#xA;</xsl:text>
</xsl:template>
```

3. **Přiřazení výsledku libovolného výrazu do proměnné**

- `<xsl:variable name="jméno" select="výraz"/>`
- pro XML dokument `jidlo-2-ovoce.xml`

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

- XSLT styl `promenna-vyraz.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

  <xsl:output method="text"
              encoding="UTF-8"/>

  <xsl:variable name="pocet" select="count(/jidlo/ovoce)"/>
  <xsl:variable name="nazvy" select="/jidlo/ovoce/nazev"/>
  <xsl:variable name="sekvenceCiselna" select="0 to $pocet"/>
  <xsl:variable name="pracovniDny"
```

```
                select="'Po', 'Út', 'St', 'Čt', 'Pá'"/>

  <xsl:template match="/">
    <xsl:text>Na skladě jsou </xsl:text>
    <xsl:value-of select="$pocet"/>
    <xsl:text> druhy ovocí: </xsl:text>
    <xsl:value-of select="$nazvy"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="$sekvenceCiselna"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="for $x in $pracovniDny return $x"/>
  </xsl:template>
</xsl:stylesheet>
```

- při definici `pocet` je použit XPath výraz, který vrací posloupnost uzlů
  - u `nazvy` je jedná o posloupnost s jedním celočíselným prvkem
- proměnná `pocet` je použita dále při definici proměnné `sekvenceCiselna`

■ vypíše:

```
Na skladě jsou 2 druhy ovocí: jablka banány
0 1 2
Po Út St Čt Pá
```

4. **Přiřazení části výstupního dokumentu do proměnné**

■ je to jen varianta nastavení řetězce do proměnné, ve které jsou však použity i XML nebo HTML značky

```
<xsl:variable name="jméno">
  libovolný zápis výstupu včetně <značek/>
  i na více řádek
</xsl:variable>
```

■ při použití tohoto způsobu je vhodné vypsat hodnotu proměnné pomocí

```
<xsl:copy-of select="$vystup"/>
```

nikoliv

```
<xsl:value-of select="$vystup"/>
```

protože `<xsl:copy-of>` zkopíruje na výstup i použité elementy, které `<xsl:value-of>` „požere“

■ pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `promenna-text-znacky.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

    <xsl:output method="html"
```

```
        encoding="UTF-8"/>

  <xsl:variable name="vystup">
    <em>Nabízíme </em><strong>chutné</strong> ovoce!
  </xsl:variable>

  <xsl:template match="/">
    <html>
      <body>
        <xsl:value-of select="$vystup"/>
        <xsl:copy-of select="$vystup"/>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

- vypíše:

```
<html>
   <body>Nabízíme chutné ovoce!
      <em>Nabízíme </em><strong>chutné</strong> ovoce!

   </body>
</html>
```

8.1.1.2. Naplnění proměnné obsahem externího souboru

- XSLT 2.0 přináší velmi užitečnou možnost, jak přečíst naráz obsah libovolného textového souboru
  - tuto možnost pak lze velmi dobře využít ke generování různých hlaviček a patiček (obecně proměnného textu nezávislého na souboru stylu) výsledného souboru
    - Pozor na skutečnost, že pokud jsou v externím souboru znaky `< & >`, je třeba vypisovat obsah proměnné pomocí `<xsl:value-of>` s nastaveným atributem `disable-output-escaping="yes"`

      Poznámka

      Instrukce `<xsl:copy-of>` nepomůže.

- pro načtení použijeme funkci `unparsed-text(jméno_souboru, kódování_souboru)`
  - obsah souboru nejčastěji uložíme do proměnné
- obsah externího souboru `pozdrav.txt`, který je v kódování UTF-8 bez BOM

```
Ahoj <strong>programátoři</strong>!
```

- pro libovolný XML dokument XSLT styl `promenna-obsah-souboru.xsl`

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

```
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

  <xsl:output method="html"
              encoding="UTF-8"/>

  <xsl:variable name="kodovaniVstupu" select="'UTF-8'"/>
  <xsl:variable name="jmenoSouboru" select="'pozdrav-UTF-8.txt'"/>

  <xsl:variable name="obsahSouboru"
    select="unparsed-text($jmenoSouboru, $kodovaniVstupu)"/>

  <xsl:template match="/">
    <html>
      <head></head>
      <body>
        <xsl:value-of disable-output-escaping="yes"
                      select="$obsahSouboru"/>
        <xsl:text>&#xA;</xsl:text>
        <xsl:copy-of select="$obsahSouboru"/>
        <xsl:text>&#xA;</xsl:text>
        <xsl:value-of select="$obsahSouboru"/>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

- spuštěn příkazem (na jménu XML souboru nezáleží):

```
saxon -o pozdrav.html jidlo-2-ovoce.xml promenna-obsah-souboru.xsl
```

vytvoří soubor `pozdrav.html` s obsahem:

```
<html>
   <head>
      <meta http-equiv="Content-Type"
        content="text/html; charset=UTF-8">
   </head>
   <body>Ahoj <strong>programátoři</strong>!
      Ahoj &lt;strong&gt;programátoři&lt;/strong&gt;!
      Ahoj &lt;strong&gt;programátoři&lt;/strong&gt;!
   </body>
</html>
```

- funkce `unparsed-text()` si bez problémů poradí i se souborem v UTF-8 s BOM (BOM „požere“)

Poznámka

Pokud načítaný soubor leží v jiném adresáři, je nutné použít `file:/` před cestou k souboru:

```
  <xsl:variable name="jmenoSouboru" ►
select="'file:/d:/zzz/pozdrav-UTF-8.txt'"/>
```

8.1.1.3. Změna hodnoty globální proměnné

Varování

Toto je rozšíření poskytované implementací Saxon. Nejedná se o obecnou vlastnost XSLT 2.0! Z tohoto důvodu je vhodné omezit používání jen na nezbytně nutné případy.

Varování

Nelze použít Saxon-HE, který toto rozšíření nemá. Použijte Saxon-B 9.1 nebo Saxon-PE.

- proměnné v XSLT 2.0 mají konstantní hodnotu
  - toto je i bezpečnostní prvek – při použití stylu, který se skládá z více souborů (viz později) se nemůže stát, že je hodnota omylem změněna
- občas ale může být výhodné tuto hodnotu umět změnit
  - Saxon pro tuto aktivitu poskytuje rozšiřující instrukci `<saxon:assign>`
- aby jakékoliv rozšíření oproti XSLT 2.0 moho fungovat, musí být začátek XSLT stylu pozměněn takto:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:saxon="http://saxon.sf.net/"
  extension-element-prefixes="saxon"
  version="2.0">
```

  - `xmlns:saxon="http://saxon.sf.net/"` je definice jmenného prostoru pojmenovaného `saxon`
  - `extension-element-prefixes="saxon"` jmenný prostor `saxon` je uveden v seznamu povolených rozšíření
- podmínky pro použití instrukce `<saxon:assign>`
  - měněná proměnná musí být globální
  - a musí být deklarována s atributem `saxon:assignable="yes"`
- XML dokument `nazvy-4-ovoce.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<jidlo>
  <nazev>jablka</nazev>
  <nazev>banány</nazev>
  <nazev>grapefruity</nazev>
  <nazev>švestky sušené</nazev>
</jidlo>
```

- XSLT styl `zmena-promenne.xsl`

- před vypsáním každého názvu ovoce odřádkuje a vypíše zvětšující se počet úvodních hvězdiček
- tento řetězec bude vypisován pomocí postupně měněné proměnné `zacatek`
  - ta je při deklaraci nastavena na hodnotu odřádkování (`'
'`)
  - aby mohla být měněna, má nastaven atribut `saxon:assignable="yes"`
  - při každém vstupu do šablony `<xsl:template match="nazev">` se k ní přidá jedna hvězdička pomocí funkce `concat()`
  - s proměnnou `zacatek` se jinak pracuje jako s běžnou proměnnou
- proměnná `pocet` ukazuje použití běžného čítače

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:saxon="http://saxon.sf.net/"
  extension-element-prefixes="saxon"
  version="2.0">

  <xsl:output method="text"
              encoding="UTF-8"/>

  <xsl:variable name="zacatek" select="'&#xA;'"
    saxon:assignable="yes"/>
  <xsl:variable name="pocet" select="1"
    saxon:assignable="yes"/>

  <xsl:template match="/">
     <xsl:apply-templates select="//nazev"/>
  </xsl:template>

  <xsl:template match="nazev">
    <saxon:assign name="zacatek" select="concat($zacatek, '*')"/>
    <xsl:value-of select="$zacatek"/>
    <xsl:value-of select="."/>
    <xsl:value-of select="$pocet"/>
    <saxon:assign name="pocet" select="$pocet + 1"/>
  </xsl:template>
</xsl:stylesheet>
```

- vypíše:

```
*jablka1
**banány2
***grapefruity3
****švestky sušené4
```

8.1.2. Parametry

- parametry mají stejné použití i definice jako proměnné
  - liší se tím, že jim lze při volání stylu nebo šablony nastavit hodnotu příkazem

    `<xsl:with-param>`
  - fungují tak jako formální parametry v běžném programovacím jazyce
- definice parametru je pomocí `<xsl:param>` a je možné použít všechny tři způsoby uvedené u definic proměnných
  - Pozor na skutečnost, že uvnitř šablony musí být definice parametru na prvním místě!
- na parametry se odkazujeme pomocí `$jméno`
- hodnotu parametru vypíšeme pomocí `<xsl:value-of select="$jméno"/>`
  - potřebujeme-li použít hodnotu parametru jako hodnotu atributu (uzavřenou do uvozovek jako řetězec), použijeme `"{$jméno}"`
- pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `parametry-zaklad.xsl`

Poznámka

Celý zdrojový kód stylu je vypsán postupně ve více částech, které jsou ihned komentovány.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

    <xsl:output method="html"
        encoding="UTF-8"/>

  <xsl:template match="/">
    <html>
      <body>
        <table border="1">
          <xsl:apply-templates select="jidlo/ovoce">
            <xsl:with-param name="pozadi" select="'blue'"/>
          </xsl:apply-templates>
        </table>
      </body>
    </html>
  </xsl:template>
```

- při volání `<xsl:apply-templates select="jidlo/ovoce">`
  - ♦ je nastaven („skutečný“) parametr volání šablony se jménem `pozadi` na hodnotu `'blue'`

```
  <xsl:template match="ovoce">
    <xsl:param name="pozadi"/>
```

```
    <tr bgcolor="{$pozadi}">
      <xsl:apply-templates select="@cislo">
<!--  <xsl:with-param name="slovo">číslo: </xsl:with-param> -->
      </xsl:apply-templates>
      <xsl:apply-templates select="nazev">
        <xsl:with-param name="pozadi">yellow</xsl:with-param>
      </xsl:apply-templates>
    </tr>
  </xsl:template>
```

- jako první příkaz v šabloně je definován prázdný („formální“) parametr `pozadi`
  - ten bude při volání nastaven na hodnotu `'blue'`
  - do hodnoty atributu `bgcolor` se zapíše příkazem `bgcolor="{$pozadi}`
- dále se volá šablona pro atribut `cislo`: `<xsl:apply-templates select="@cislo">`
  - protože je nastavení skutečného parametru `slovo` zakomentováno, použije se v této šabloně přednastavená hodnota
- při volání šablony pro element `nazev` se přenastavuje hodnota parametru `pozadi` na `yellow`

```
  <xsl:template match="@cislo">
    <xsl:param name="slovo" select="'č.'"/>
    <td>
      <xsl:value-of select="$slovo"/>
      <xsl:value-of select="."/>
    </td>
  </xsl:template>
```

- formální parametr `slovo` je definován s nastavením implicitní hodnoty `'č.'`
  - ta bude použita v případě, že skutečný parametr `slovo` není z volající šablony nastaven

```
  <xsl:template match="nazev">
    <xsl:param name="pozadi"/>
    <td bgcolor="{$pozadi}">
      <xsl:value-of select="."/>
    </td>
  </xsl:template>

</xsl:stylesheet>
```

- formální parametr `pozadi` je definován bez nastavení implicitní hodnoty
  - pokud by nebyl skutečný parametr `pozadi` z volající šablony nastaven, měl by prázdnou hodnotu

■ vypíše:

```
<html>
   <body>
      <table border="1">
         <tr bgcolor="blue">
            <td>č.1</td>
```

```
            <td bgcolor="yellow">jablka</td>
         </tr>
         <tr bgcolor="blue">
            <td>č.2</td>
            <td bgcolor="yellow">banány</td>
         </tr>
      </table>
   </body>
</html>
```

8.1.3. Globální parametry

- definujeme-li parametry na globální úrovni (mimo jakoukoliv šablonu), nemají zdánlivě význam (neliší se od proměnných) – není, kdo by je jako skutečné parametry ve stylu nastavil
  - nastavit se ale dají z vnějšku při volání transformace
    - je to **velmi používaná možnost**, jak ovlivnit práci stylu bez změny XSL souboru
- pro libovolný XML dokument XSLT styl `parametr-globalni.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

    <xsl:output method="html"
        encoding="UTF-8"/>

  <xsl:param name="globalniPozdrav" select="'nazdar'"/>

  <xsl:template match="/">
    <html>
      <body>
        <xsl:value-of select="$globalniPozdrav"/>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

- bez dalších nastavení vypíše:

```
<html>
   <body>nazdar</body>
</html>
```

- při spuštění XSLT procesoru příkazem (na jménu XML souboru nezáleží):

```
saxon -o a.html jidlo-2-ovoce.xml parametr-globalni.xsl "globalniPozdrav=Ahoj"
```

- vytvoří soubor `a.html` s obsahem:

```
<html>
   <body>Ahoj</body>
</html>
```

8.1.4. Typová kompatibilita proměnných a parametrů

- jazyk XSLT není silně typovaný jazyk
  - v naprosté většině případů není datový typ podstaný
  - XSLT však umožňuje použít datové typy známé z XSD schémat
    - např. `xs:integer`, `xs:double` apod.
- číslelná proměnná (nebo parametr) je dle okolností vzniku celočíselná či reálná
  - pokud vznikla transformací z řetězce pomocí funkce `number()`, je reálná
    - to pak může způsobit potíže při překladu nebo při běhu stylu např. v instrukci `<xsl:for-each>`, která striktně vyžaduje v sekvenci celočíselnou proměnnou
- při definici proměnné (nebo parametru), lze pomocí atributu `as` definovat typ proměnné
  - typy jsou z XSD a pro jejich použití je v hlavičce stylu nutno definovat příslušný jmenný prostor, např. příkazem

    ```
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    ```

  - prarametr (nebo proměnná) `celeCislo` pak může být definován jako:

    ```
    <xsl:param name="celeCislo" as="xs:integer"/>
    ```

- kromě definování typu proměnné/parametru, lze přetypovat i výsledek XPath výrazu, např.

  ```
  <xsl:variable name="c" select="xs:integer($a - $b)"/>
  ```

- pro libovolný XML dokument XSLT styl `pretypovani.xsl`

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <xsl:stylesheet
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      version="2.0">

    <xsl:output method="text"
                encoding="UTF-8"/>

   <xsl:variable name="a" select="number('10')"/>
    <xsl:variable name="b" select="number('4')"/>
  <!--   <xsl:variable name="a" select="10"/>
    <xsl:variable name="b" select="4"/>
  -->
  ```

```
  <xsl:template match="/">
    <!-- pretypovani vyrazu -->
<!--  <xsl:variable name="c" select="xs:integer($a - $b)"/>
-->
    <!-- druha moznost -->
    <xsl:variable name="c" as="xs:integer" select="xs:integer($a - $b)"/>

    <!-- chybna moznost - projevi se az v cyklu  -->
<!--  <xsl:variable name="c" select="$a - $b"/>-->

    <xsl:text>Výpočet: </xsl:text>
    <xsl:value-of select="$a"/>
    <xsl:text> - </xsl:text>
    <xsl:value-of select="$b"/>
    <xsl:text> = </xsl:text>
    <xsl:value-of select="$c"/>

    <xsl:text>&#xA;Cyklus: </xsl:text>
    <xsl:for-each select="1 to $c">
      <xsl:value-of select="."/>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

- vypíše očekávané:
  - všimněte si, že hodnoty `10` a `4` jsou vypisovány jako celá čísla

```
Výpočet: 10 - 4 = 6
Cyklus: 123456
```

- pokud ale vynecháme přetypování při odčítání, tj.

```
<xsl:variable name="c" select="$a - $b"/>
```

  - hlásí Saxon chybu:

  ```
  Required item type of second operand of 'to' is xs:integer;
  supplied value has item type xs:double
  ```

    - a kuriozně tuto chybu přiřazuje k řádce, kde začíná definice šablony, tj. `<xsl:template match="/">`

8.2. Pojmenované šablony

- umíme-li využít proměných a zejména parametrů, můžeme v XSLT vytvářet podprogramy tak, jak jsme zvyklí z běžných programovacích jazyků
- tyto podprogramy jsou opět šablony (definované pomocí `<xsl:template>`)

- od dosud známých šablon, které byly identifikovány pomocí atributu `match`, se liší tím, že jsou pojmenovány pomocí atributu `name`
  - podle tohoto jména se pak dají kdykoliv vyvolávat (i nezávisle na XPath výrazech) příkazem

    `<xsl:call-template>`
- typické použití je pro:
  - akce, které se vícekrát opakují
  - lepší strukturování kódu – rozdělení kódu do více menších částí
- pro libovolný XML dokument je ve stylu připravena šablona pro součet dvou čísel (`pojmenovana-sablona.xsl`)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

  <xsl:output method="text"
              encoding="UTF-8"/>

  <xsl:template match="/">
    <xsl:call-template name="sectiDveCisla">
      <xsl:with-param name="a" select="3"/>
      <xsl:with-param name="b" select="7"/>
    </xsl:call-template>
  </xsl:template>

  <xsl:template name="sectiDveCisla">
    <xsl:param name="a"/>
    <xsl:param name="b"/>
    <xsl:text>Součet je </xsl:text>
    <xsl:value-of select="$a + $b"/>
  </xsl:template>
</xsl:stylesheet>
```

- vypíše:

```
Součet je 10
```

8.2.1. Pojmenovaná šablona volaná z příkazové řádky

- XSLT transformaci nemusíme nutně spouštět nad určitým (konkrétním) XML dokumentem
  - vstupní data lze získat i jinak
    - z jiného typu vstupního souboru, který načteme – viz dále `unparsed-text()`
    - výpočtem např. ze sekvence – viz dále

♦ výpočtem inicializovaným globálním parametrem – viz dále

- je možné vyvolat pojmenovanou šablonu přímo z příkazové řádky, např. transformaci spustit příkazem:

```
saxon -o výstupní_soubor -it jméno_šablony soubor_stylu.xsl
```

kde:

♦ `-o výstupní_soubor` je označení, kam se zapíše výstup

Varování

Musí to být první parametr!

♦ `-it` je příkaz *initialize template*

♦ `jméno_šablony` je jméno uvedené v `<xsl:template name="jméno_šablony">`

♦ `soubor_stylu.xsl` je jméno souboru s XSLT stylem

■ XSLT styl (`pojmenovana-sablona-cmd.xsl`) spuštěný příkazem:

- všimněte si, že zde nepotřebujeme žádný pomocný XML soubor

```
saxon -o ahoj.html -it pojmenovanaSablona pojmenovana-sablona-cmd.xsl
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

    <xsl:output method="html"
        encoding="UTF-8"/>

  <xsl:template name="pojmenovanaSablona">
    <html>
      <body>
        ahoj
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

■ vytvoří soubor `ahoj.html` s obsahem:

```
<html>
   <body>
      ahoj

   </body>
</html>
```

8.2.2. Pojmenovaná šablona a globální parametry

- tento příklad bude ukázkou, jak lze využít pojmenované šablony v kombinaci s globálními parametry
  - příklad nepřináší žádné nové poznatky, pouze využívá již dříve uvedených možností
    - globálního parametru `pocet`, který udává počet opakování a má implicitní hodnotu 3
    - dvou pojmenovaných šablon
      - `radky` – vypíše jednosloupcovou tabulku se zadaným počtem řádek
      - `sloupce` – vypíše jednořádkovou tabulku se zadaným počtem sloupců
    - cyklu pomocí XPath výrazu a posloupnosti – viz podrobně dříve
- XSLT styl `sablona-a-glob-param.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="2.0">

  <xsl:output method="html"
              encoding="UTF-8"/>

  <xsl:param name="pocet" select="3"/>

  <xsl:template name="radky">
    <html>
      <body>
        <table border="1">
          <xsl:for-each select="1 to $pocet">
            <tr>
              <td>
                <xsl:value-of select="."/>
              </td>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

  <xsl:template name="sloupce">
    <html>
      <body>
        <table border="1">
          <tr>
            <xsl:for-each select="1 to $pocet">
              <td>
                <xsl:value-of select="."/>
              </td>
            </xsl:for-each>
```

```
          </tr>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

- spuštěn příkazem:

```
saxon -o sloupce.html -it sloupce sablona-a-glob-param.xsl
```

vytvoří soubor `sloupce.html` s obsahem:

```
<html>
   <body>
      <table border="1">
         <tr>
            <td>1</td>
            <td>2</td>
            <td>3</td>
         </tr>
      </table>
   </body>
</html>
```

- spuštěn příkazem:

```
saxon -o sloupce.html -it sloupce sablona-a-glob-param.xsl "pocet=2"
```

vytvoří soubor `sloupce.html` s obsahem:

```
<html>
   <body>
      <table border="1">
         <tr>
            <td>1</td>
            <td>2</td>
         </tr>
      </table>
   </body>
</html>
```

- spuštěn příkazem:

```
saxon -o radky.html -it radky sablona-a-glob-param.xsl "pocet=4"
```

vytvoří soubor `radky.html` s obsahem:

```
<html>
   <body>
      <table border="1">
```

```
        <tr><td>1</td></tr>
        <tr><td>2</td></tr>
        <tr><td>3</td></tr>
        <tr><td>4</td></tr>
      </table>
   </body>
</html>
```

8.3. Definice vlastních funkcí

- představují jistou variantu pojmenovaných šablon
- jedná se skutečně o funkce, ve významu funkcí z běžných programovacích jazyků
  - již jsme se s nimi setkali – jsou to funkce známé z XPath
- funkce se deklaruje pomocí instrukce

```
<xsl:function name="prefix:jmenoFunkce" as="typNavratoveHodnoty">
```

  - každá nově deklarovaná funkce musí mít svůj `prefix`, který si můžeme libovolně zvolit z dosud nepoužitých prefixů
    - prefixu (zde `str`) musíme v hlavičce stylu nadefinovat jeho jmenný prostor dle známých pravidel, např.:

    ```
    xmlns:str="http://www.kiv.zcu.cz/~herout/XSLTfunkce"
    ```

  - atribut `as="typNavratoveHodnoty"` určuje typ návratové hodnoty
    - tento typ je v naprosté většině případů typem z XSD schémat
      - aby je bylo možné použít, musíme v hlavičce stylu nadefinovat jeho jmenný prostor, typicky:

      ```
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      ```

Poznámka

Funkce nemusejí mít udán typ návratové hodnoty ani typ parametrů. Tento způsob nedoporučuji, ale pro úplnost je uveden později.

- funkce může mít libovolný počet vstupních (formálních) parametrů
  - deklarují se pomocí již známé instrukce `<xsl:param name="jmeno" as="typParametru"/>`
    - oproti parametrům pojmenovaných šablon se u parametrů funkcí udává jejich typ pomocí atributu `as`
      - u parametrů pojmenovaných šablon se tato možnost využívá jen zřídka
- v těle funkce mohou být pouze některé `<xsl>` instrukce, typicky deklarace případných pomocných proměnných

- nesmějí zde být instrukce způsobující zápis do výstupního proudu, jako `<xsl:text>`, `<xsl:value-of>` apod.
  - pokud se vyskytnou, je pak při běhu hlášena chyba

```
A sequence of more than one item is not allowed
as the result of function str:prvniVelkePismeno() ("jablka", "Jablka")
```

- návratová hodnota se vrací pomocí instrukce `<xsl:sequence select="XPath_navratovaHodnota" />`

- pro XML dokument `jidlo-2-ovoce.xml`

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

- XSLT styl `funkce.xsl`
  - funkce `str:prvniVelkePismeno`
    - má návratový typ `xs:string`
    - má jeden vstupní (formální) parametr pojmenovaný `s` typu `xs:string`
    - využívá postupně tří pomocných proměnných `bezMezer`, `prvniVelke`, `ostatniMala`, které jsou deklarovány a používány dle běžných zvyklostí
      - použití pomocných proměnných není nezbytné – celý výraz je možné umístit jen do instrukce `<xsl:sequence>`
  - volání funkce v těle hlavní šablony je podle pravidel známých z volání XPath funkcí

```
<xsl:value-of select="str:prvniVelkePismeno(nazev)"/>
```

  - v příkladu je použit trik, kdy se pomocí `<xsl:if test="not(position() = last())">` nevypisuje oddělovací `,` za posledním uzlem
    - to nemá samozřejmě s funkcemi nic společného

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:str="http://www.kiv.zcu.cz/~herout/XSLTfunkce"
```

```
  version="2.0">

  <xsl:output method="html" encoding="UTF-8"/>

  <xsl:function name="str:prvniVelkePismeno" as="xs:string">
    <xsl:param name="s" as="xs:string"/>
    <xsl:variable name="bezMezer"
                  select="normalize-space($s)"/>
    <xsl:variable name="prvniVelke"
                  select="upper-case(substring($bezMezer, 1, 1))"/>
    <xsl:variable name="ostatniMala"
                  select="lower-case(substring($bezMezer, 2))"/>
    <xsl:sequence select="concat($prvniVelke, $ostatniMala)"/>
  </xsl:function>

  <xsl:template match="/jidlo">
    <html>
      <head></head>
      <body>
        <xsl:for-each select="ovoce">
          <xsl:value-of select="str:prvniVelkePismeno(nazev)"/>

          <xsl:if test="not(position() = last())">
            <xsl:text>, </xsl:text>
          </xsl:if>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

- vypíše

```
<html xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:str="http://www.kiv.zcu.cz/~herout/XSLTfunkce">
   <head>
      <meta http-equiv="Content-Type"
         content="text/html; charset=UTF-8">
   </head>
   <body>Jablka, Banány</body>
</html>
```

8.3.1. Funkce generující HTML bez jmenných prostorů

- v předchozím případě nám ve vygenerovaném HTML mohlo vadit zbytečné uvádění jmenných prostorů
  - ty neměly s dalším obsahem HTML souboru nic společného
- výpis jmenných prostorů, které nemají být v HTML uvedeny, se dá potlačit pomocí atributu `exclude-result-prefixes` elementu `<xsl:stylesheet>`
  - seznam jmenných prostorů oddělených mezerami se uvede jako hodnota tohoto atributu, např.:

```
  exclude-result-prefixes="xs str"
```

- pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `funkce-html-bez-NS.xsl`
  - celý styl je naprosto stejný, jako předchozí styl `funkce.xsl`
  - je doplněna se pouze řádka `exclude-result-prefixes="xs str"`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:str="http://www.kiv.zcu.cz/~herout/XSLTfunkce"
    exclude-result-prefixes="xs str"
    version="2.0">
...
```

- vypíše

```
<html>
   <body>Jablka, Banány</body>
</html>
```

8.3.2. Funkce s více parametry

- má-li funkce více formálních parametrů, pak jejich pořadí v deklaraci určuje pořadí skutečných parametrů při volání funkce

- pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `funkce-dva-parametry.xsl`
  - funkce `str:nejakeVelkePismeno` změní písmeno na určitém indexu na velké, ostatní ponechá malá
  - u volání této funkce si všimněte přetypování součtu u druhého parametru `xs:integer(@cislo + 2)`, kdy tento parametr musí být typu `xs:integer`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:str="http://www.kiv.zcu.cz/~herout/XSLTfunkce"
    exclude-result-prefixes="xs str"
    version="2.0">

  <xsl:output method="html"
              encoding="UTF-8"/>

  <xsl:function name="str:nejakeVelkePismeno" as="xs:string">
    <xsl:param name="s" as="xs:string"/>
    <xsl:param name="index" as="xs:integer"/>
```

```
    <xsl:variable name="zacatek"
                  select="lower-case(substring($s, 1, $index - 1))"/>
    <xsl:variable name="velke"
                  select="upper-case(substring($s, $index, 1))"/>
    <xsl:variable name="konec"
                  select="lower-case(substring($s, $index + 1))"/>
    <xsl:sequence select="concat($zacatek, $velke, $konec)"/>
  </xsl:function>

  <xsl:template match="/jidlo">
    <html>
      <body>
        <xsl:for-each select="ovoce">
<!--          <xsl:value-of select="str:nejakeVelkePismeno(nazev, ►
xs:integer(@cislo + 2))"/>-->
          <xsl:value-of select="str:nejakeVelkePismeno(nazev, ►
xs:integer(@cislo) + 2)"/>
         <!--        <xsl:value-of select="str:nejakeVelkePismeno(nazev, ►
@cislo + 2)"/>-->

          <xsl:if test="not(position() = last())">
            <xsl:text>, </xsl:text>
          </xsl:if>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

- vypíše

```
<html>
   <body>jaBlka, banÁny</body>
</html>
```

8.3.3. Složitější funkce

- v těle funkce mohou být běžné instrukce pro podmínky, cykly apod., např. `<xsl:if>`, `<xsl:choose>`, apod.

- pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `funkce-slozitejsi.xsl`
  - funkce `str:nejakeVelkePismenoTest` pracuje jako dříve uvedená funkce `str:nejakeVelke-Pismeno`, pouze navíc otestuje vhodné nastavení `indexu`
    - je-li `index` větší než délka řetězce, ponechá původní řetězec nezměněn

```
...
  <xsl:function name="str:nejakeVelkePismenoTest" as="xs:string">
    <xsl:param name="s" as="xs:string"/>
    <xsl:param name="index" as="xs:integer"/>
```

```
      <xsl:choose>
        <xsl:when test="string-length($s) &gt;= $index">
          <xsl:variable name="zacatek"
                        select="lower-case(substring($s, 1, $index - 1))"/>
          <xsl:variable name="velke"
                        select="upper-case(substring($s, $index, 1))"/>
          <xsl:variable name="konec"
                        select="lower-case(substring($s, $index + 1))"/>
          <xsl:sequence select="concat($zacatek, $velke, $konec)"/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:sequence select="$s"/>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:function>
...
      <xsl:for-each select="ovoce">
         <xsl:value-of select="str:nejakeVelkePismenoTest(nazev,
                                  xs:integer(@cislo + 5))"/>
...
```

■ vypíše

```
<html>
   <body>jablkA, banány</body>
</html>
```

8.3.4. Funkce bez udaného typu návratové hodnoty nebo parametrů

■ dosud uváděné příklady funkcí měly striktně definovaný jak typ návratové hodnoty, tak i typy formálních parametrů

- ani jeden typ není nutné udávat
  - u těchto funkcí pak neplatí některá pravidla uvedená dříve
  - takovéto funkce lze využít pro jednoduché naprogramování speciální funkčnosti

Poznámka

Z hlediska čistoty programování by bylo vhodné se zamyslet, zda by nám pro tuto funkčnost stejně dobře neposloužily pojmenované šablony.

■ pro libovolný XML dokument XSLT styl `funkce-bez-typu.xsl`

- v příkladu jsou užity tři různé nestandardní funkce

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:str="http://www.kiv.zcu.cz/~herout/XSLTfunkce"
    exclude-result-prefixes="xs str"
    version="2.0">

    <xsl:output method="html" encoding="UTF-8"/>

  <!-- Funkce bez definovaneho navratoveho typu -->
  <xsl:function name="str:ahoj1">
    <xsl:param name="s" as="xs:string" />
    <xsl:sequence select="concat('Ahoj ', $s)" />
  </xsl:function>

  <!-- Funkce, která vrací řetězec pomocí value-of -->
  <xsl:function name="str:ahoj2">
    <xsl:param name="s"/>
    <xsl:value-of select="'Ahoj'" />
    <xsl:text> </xsl:text>
    <xsl:value-of select="$s" />
  </xsl:function>

  <!-- Funkce, která vrací nějaké XML -->
  <xsl:function name="str:ahoj3">
    <xsl:param name="s" as="xs:string" />
    <ahoj>
      <pozdravOd><xsl:value-of select="$s" /></pozdravOd>
    </ahoj>
  </xsl:function>

    <xsl:template match="/">
        <html>
            <body>
              <xsl:value-of select="str:ahoj1('lidi')"/>
              <xsl:text>&#xA;</xsl:text>
              <xsl:value-of select="str:ahoj2(3.14)"/>
              <xsl:text>&#xA;</xsl:text>
              <xsl:copy-of select="str:ahoj3('studenti')"/>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

- vypíše

```
<html>
   <body>Ahoj lidi
      Ahoj 3.14

      <ahoj>
         <pozdravOd>studenti</pozdravOd>
      </ahoj>
```

```
  </body>
</html>
```

Kapitola 9. XSLT 3

9.1. Řazení uzlů

- pokud zpracováváme více uzlů (pomocí `<xsl:for-each>` nebo pomocí `<xsl:apply-template>`) jsou uzly v tom pořadí, v jakém byly načteny z XML dokumentu
  - pořadí uzlů můžeme před zpracováním změnit, např. pomocí sekvence -- viz dříve
  - mnohem častěji ale uzly řadíme podle různých kriterií
- pro řazení se používá instrukce `<xsl:sort>`, která má následující atributy
  - `select` – výraz, který vybere hodnoty (klíče), podle kterých se řadí
  - `data-type` – typ řazení
    - defaultně je lexikografické – porovnávají se řetězce – hodnota `text`

      pořadí je např. 1, 11, 17, 3, 3.14, 31, 40, 8
    - pro porovnávání čísel se použije hodnota `number` – pak lze společně řadit (není nutno rozlišovat) celá i reálná čísla (např. 3, 3.14, 4)

      pořadí je např. 1, 3, 3.14, 8, 11, 17, 31, 40
  - `order` – vzestupné `ascending` nebo sestupné `descending`
  - `lang` – kód jazyka, podle jehož pravidel se řadí
    - české řazení je plně implementováno – hodnota `cs`
  - `case-order` – malá písmena budou řazena před velká = česká norma (`lower-first`) nebo naopak (`upper-first`)
  - `stable` – zda má být řazení stabilní (`yes`), čili položky se stejným klíčem zůstanou v původním pořadí, nebo nestabilní (`no`), položky se stejným klíčem budou v náhodném pořadí
- doporučuje se nastavit všechny relevantní atributy a nespoléhat se na defaultní hodnoty

9.1.1. Řazení podle jednoho klíče

- příkaz `<xsl:sort>` se uvede pouze jednou
- pro XML dokument `vyrazy-razeni.xml`, ve kterém atribut `poradi` určuje správné pořadí podle české normy řazení

```
<?xml version="1.0" encoding="UTF-8"?>
<vyrazy>
  <vyraz cislo="1" poradi="14">platnost</vyraz>
  <vyraz cislo="2" poradi="10">plát</vyraz>
  <vyraz cislo="3" poradi="12">plátno</vyraz>
  <vyraz cislo="4" poradi="7">plankton</vyraz>
```

```
  <vyraz cislo="5" poradi="3">plachta</vyraz>
  <vyraz cislo="6" poradi="8">plášť</vyraz>
  <vyraz cislo="7" poradi="2">plahočení</vyraz>
  <vyraz cislo="8" poradi="5">Plánička</vyraz>
  <vyraz cislo="9" poradi="11">platno</vyraz>
  <vyraz cislo="10" poradi="1">placebo</vyraz>
  <vyraz cislo="11" poradi="6">plaňka</vyraz>
  <vyraz cislo="12" poradi="9">plat</vyraz>
  <vyraz cislo="13" poradi="4">plánička</vyraz>
  <vyraz cislo="14" poradi="13">platnost</vyraz>
</vyrazy>
```

- XSLT styl `sort-primarni.xsl`, který řetězce seřadí podle pravidel českého řazení
  - je použita instrukce `<xsl:for-each>`
    - instrukce `<xsl:sort>` pak musí být první instrukce v `<xsl:for-each>`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

  <xsl:output method="text"
              encoding="UTF-8"/>

  <xsl:template match="/vyrazy">
    <xsl:for-each select="vyraz">
      <xsl:sort select="."
                data-type="text"
                order="ascending"
                lang="cs"
                case-order="lower-first"
                stable="yes"/>
      <xsl:value-of select="@cislo"/>
      <xsl:text> - </xsl:text>
      <xsl:value-of select="@poradi"/>
      <xsl:text>. </xsl:text>
      <xsl:value-of select="."/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

- vypíše:

Varování

Je nutné použít Saxon-B. Saxon-HE (i Saxon-PE) řadí špatně písmeno ch.

písmeno h se řadí před ch („plahočení“ před „plachta“) nikoliv ch hned za c („placebo“, „plachta“)

```
10 - 1. placebo
7 - 2. plahočení
5 - 3. plachta
13 - 4. plánička
8 - 5. Plánička
11 - 6. plaňka
4 - 7. plankton
6 - 8. plášť
12 - 9. plat
2 - 10. plát
9 - 11. platno
3 - 12. plátno
1 - 14. platnost
14 - 13. platnost
```

- všimněte si posledních dvou položek `platnost`, kdy jsou řetězce totožné – stabilní metoda řazení dodrží původní pořadí těchto položek, tzn. číslo `1` před číslem `14`

9.1.2. Řazení podle více klíčů

- též víceúrovňové řazení
- pokud jsou primární klíče stejné, lze další (následující) instrukcí `<xsl:sort>` určit sekundární klíč, případně další `<xsl:sort>` klíč ternární

Varování

Atribut `stable` může být nastaven pouze pro první instrukci `<xsl:sort>`.

- XSLT styl (`sort-sekundarni.xsl`), který řetězce seřadí podle pravidel českého řazení a stejné položky pak podle sekundárního klíče `poradi`
  - je použita instrukce `<xsl:apply-templates>`
    - tento způsob je přehlednější – odděluje řazení od formátování výstupu

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

  <xsl:output method="text"
              encoding="UTF-8"/>

  <xsl:template match="/vyrazy">
    <xsl:apply-templates select="vyraz">
      <xsl:sort select="." data-type="text"
                order="ascending" lang="cs"
                case-order="lower-first"
                stable="yes"/>
      <xsl:sort select="@poradi" data-type="number"
                order="ascending"/>
    </xsl:apply-templates>
```

```
  </xsl:template>

  <xsl:template match="vyraz">
    <xsl:value-of select="@cislo"/>
    <xsl:text> - </xsl:text>
    <xsl:value-of select="@poradi"/>
    <xsl:text>. </xsl:text>
    <xsl:value-of select="."/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

- vypíše:

```
10 - 1. placebo
7 - 2. plahočení
5 - 3. plachta
13 - 4. plánička
8 - 5. Plánička
11 - 6. plaňka
4 - 7. plankton
6 - 8. plášť
12 - 9. plat
2 - 10. plát
9 - 11. platno
3 - 12. plátno
14 - 13. platnost
1 - 14. platnost
```

poslední dvě položky `platnost` už jsou ve správném pořadí určeném sekundárním klíčem

9.2. Seskupování uzlů

- XSLT 2.0 umožňuje použít mocnou instrukci `<xsl:for-each-group>`, která dovoluje vytvářet skupiny uzlů a dále je zpracovávat podle mnoha kritérií

9.2.1. Základní použití

- v instrukci `<xsl:for-each-group>` se množina zpracovávaných uzlů vybírá standardně pomocí atributu `select`
- seskupování (seskupovací kriterium) se pak nastaví v atributu `group-by`
- hodnotu seskupovacího kriteria lze získat pomocí funkce `current-grouping-key()`
- seznam uzlů v dané skupině se získá funkcí `current-group()`
- XML dokument `vyrazy-razeni-akcenty.xml`

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

```
  <vyraz cislo="1" poradi="14" skupina="ascii">platnost</vyraz>
  <vyraz cislo="2" poradi="10" skupina="akcenty">plát</vyraz>
  <vyraz cislo="3" poradi="12" skupina="akcenty">plátno</vyraz>
  <vyraz cislo="4" poradi="7" skupina="ascii">plankton</vyraz>
  <vyraz cislo="5" poradi="2" skupina="ascii">plachta</vyraz>
  <vyraz cislo="6" poradi="8" skupina="akcenty">plášť</vyraz>
  <vyraz cislo="7" poradi="2" skupina="akcenty">plahočení</vyraz>
  <vyraz cislo="8" poradi="5" skupina="akcenty">Plánička</vyraz>
  <vyraz cislo="9" poradi="11" skupina="ascii">platno</vyraz>
  <vyraz cislo="10" poradi="1" skupina="ascii">placebo</vyraz>
  <vyraz cislo="11" poradi="6" skupina="akcenty">plaňka</vyraz>
  <vyraz cislo="12" poradi="9" skupina="ascii">plat</vyraz>
  <vyraz cislo="13" poradi="4" skupina="akcenty">plánička</vyraz>
  <vyraz cislo="14" poradi="13" skupina="ascii">platnost</vyraz>
</vyrazy>
```

- XSLT styl `seskupovani-zaklad.xsl` má za úkol vypsat odděleně skupiny slov, které jsou pouze v ASCII nebo které obsahují akcenty
  - pořadí slov ve skupinách zůstane zachováno

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="2.0">

  <xsl:output method="html"
    encoding="UTF-8"/>

  <xsl:template match="/vyrazy">
    <html>
      <head></head>
      <body>
        <xsl:for-each-group select="vyraz"
                            group-by="@skupina">
          <p>Slova
           <strong><xsl:value-of select="current-grouping-key()"/></strong>
            <br/>
            <xsl:apply-templates select="current-group()"/>
          </p>
        </xsl:for-each-group>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="vyraz">
    <xsl:value-of select="@cislo"/>
    <xsl:text>.</xsl:text>
    <xsl:value-of select="."/>
    <xsl:text>, </xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

■ vypíše:

```
<html>
   <body>
      <p>Slova
         <strong>ascii</strong><br>1.platnost, 4.plankton, 5.plachta, ►
9.platno, 10.placebo, 12.plat, 14.platnost,
      </p>
      <p>Slova
        <strong>akcenty</strong><br>2.plát, 3.plátno, 6.plášť, 7.plahočení, ►
8.Plánička, 11.plaňka, 13.plánička,
      </p>
   </body>
</html>
```

9.2.2. Řazení ve skupině

■ jak skupiny samotné, tak i uzly uvnitř skupiny lze řadit pomocí `<xsl:sort>`

■ pro stejný XML dokument `vyrazy-razeni-akcenty.xml` XSLT styl `seskupovani-sort.xsl`

- skupiny seřazené podle abecedy – nejdříve `akcenty` a pak `ascii`
  - skupiny jsou číslované pomocí funkce `position()`
- slova uvnitř skupiny seřazená podle abecedy

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0">

  <xsl:output method="html"
              encoding="UTF-8"/>

  <xsl:template match="/vyrazy">
    <html>
      <head></head>
      <body>
        <xsl:for-each-group select="vyraz"
                            group-by="@skupina">
          <xsl:sort select="current-grouping-key()" lang="cs"/>
          <p>Slova
            <strong>
              <xsl:value-of select="position()"/>
              <xsl:text>. </xsl:text>
              <xsl:value-of select="current-grouping-key()"/>
            </strong>
            <br/>
            <xsl:apply-templates select="current-group()">
              <xsl:sort select="." lang="cs"/>
            </xsl:apply-templates>
```

```
        </p>
      </xsl:for-each-group>
    </body>
  </html>
</xsl:template>

<xsl:template match="vyraz">
  <xsl:value-of select="@cislo"/>
  <xsl:text>.</xsl:text>
  <xsl:value-of select="."/>
  <xsl:text>, </xsl:text>
</xsl:template>

</xsl:stylesheet>
```

- vypíše:

```
<html>
   <body>
      <p>Slova
        <strong>1. akcenty</strong><br>7.plahočení, 13.plánička, 8.Plánička, ►
11.plaňka, 6.plášť, 2.plát, 3.plátno,
      </p>
      <p>Slova
         <strong>2. ascii</strong><br>10.placebo, 5.plachta, 4.plankton, ►
12.plat, 9.platno, 1.platnost, 14.platnost,
      </p>
   </body>
</html>
```

9.3. Styl uložený ve více souborech

- máme-li složitější styly s mnoha šablonami, je z hlediska přehlednosti výhodné, mít je uloženy v několika souborech
- další dobrý důvod pro více XSLT souborů je případ, kdy využíváme již hotové části stylu
  - nejčastěji by to byly zřejmě knihovny funkcí nebo vyhovující šablony
- v těchto případech máme dvě možnosti spojení stylů – `<xsl:include>` a `<xsl:import>`

9.3.1. Vložení stylu `<xsl:include>`

- tento způsob je jednodušší a není vhodný pro dodatečné změny šablon
- pomocí této instrukce můžeme do stylu vložit jiný styl
  - prakticky to znamená, že po spuštění transformace se „vkopíruje“ 1 : 1 vkládaný soubor do souboru, ve kterém se nachází instrukce `<xsl:include>`

- ♦ všechny šablony, funkce, proměnné a parametry se chápou tak, jako by byly přímo vloženy do stylu, který instrukci `<xsl:include>` obsahuje
- ♦ to má tu nevýhodu, že případné konflikty mezi těmito částmi stylu (stejná jména apod.) nejsou nijak ošetřovány a projeví se chybou při spuštění transformace

- instrukce `<xsl:include>` může být na libovolném místě

■ formát instrukce je: `<xsl:include href="URI">`

- kde `URI` je nejčastěji jméno XSL souboru

■ praktické použití je pro natažení již odladěného souboru knihovních funkcí, které se nebudou měnit, pouze se budou používat

■ XSLT styl `include-volana.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:str="http://www.kiv.zcu.cz/~herout/XSLTfunkce"
    version="2.0">

   <xsl:variable name="nadpis" select="'seznam ovocí:'"/>

    <xsl:function name="str:velkaPismena" as="xs:string">
        <xsl:param name="s" as="xs:string"/>
        <xsl:sequence select="upper-case($s)"/>
    </xsl:function>

    <xsl:template name="vypis">
        <xsl:param name="jmenoOvoce"/>
      <xsl:text>&#xA;</xsl:text>
      <xsl:value-of select="str:velkaPismena($jmenoOvoce)"/>
    </xsl:template>

</xsl:stylesheet>
```

■ pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `include-hlavni.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:str="http://www.kiv.zcu.cz/~herout/XSLTfunkce"
  exclude-result-prefixes="str"
  version="2.0">

  <xsl:output method="html"
       encoding="UTF-8"/>

  <xsl:include href="include-volana.xsl"/>

  <xsl:template match="/jidlo">
```

```
    <html>
      <body>
        <xsl:value-of select="str:velkaPismena($nadpis)"/>

        <xsl:for-each select="ovoce">
          <xsl:call-template name="vypis">
            <xsl:with-param name="jmenoOvoce" select="nazev"/>
          </xsl:call-template>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

- vypíše

```
<html>
   <body>SEZNAM OVOCÍ:
      JABLKA
      BANÁNY
   </body>
</html>
```

- pokud dojde ke konfliktu jmen, například ve stylu `include-hlavni.xsl` deklarujeme proměnnou

```
  <!-- konflikt stejneho jmena -->
  <xsl:variable name="nadpis" select="'hlavní seznam ovocí:'"/>
```

bude po spuštění transformace nahlášena chyba

```
Duplicate global variable declaration
```

9.3.2. Import stylu `<xsl:import>`

- tento způsob vkládání souborů je složitější než předchozí pomocí `<xsl:include>`
- instrukce `<xsl:import>` musí být ve stylu uvedena jako první (i před `<xsl:output>`)
- opět je vkládán soubor stylu, ale případné konflikty jmen jsou speciálním způsobem ošetřeny
  - prakticky „nové“ (z volajícího souboru) zastiňuje „staré“ (z volaného souboru)
    - to znamená, že zastíněná stará definice nebude přístupná buď vůbec nebo pouze pomocí speciálního příkazu
- formát instrukce je: `<xsl:import href="URI">`
  - kde `URI` je nejčastěji jméno XSL souboru
- tento případ je výhodný, pokud nám ve starém stylu nevyhovují pouze drobnosti, které se v novém stylu snadno upraví

- nejvíce se výhody projeví pro hodně parametrizované styly

■ styly lze importovat i víceúrovňově, tzn. importovaný styl v sobě může importovat jiný styl

■ XSLT styl `import-volana.xsl` (má stejný obsah jako předchozí `include-volana.xsl`)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:str="http://www.kiv.zcu.cz/~herout/XSLTfunkce"
  version="2.0">

  <xsl:variable name="nadpis" select="'seznam ovocí:'"/>

  <xsl:function name="str:velkaPismena" as="xs:string">
    <xsl:param name="s" as="xs:string"/>
    <xsl:sequence select="upper-case($s)"/>
  </xsl:function>

  <xsl:template name="vypis">
    <xsl:param name="jmenoOvoce"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="str:velkaPismena($jmenoOvoce)"/>
  </xsl:template>

</xsl:stylesheet>
```

■ pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `import-hlavni.xsl`

- je definována proměnná stejného jména `nadpis` – ve výpisu bude použita hodnota této proměnné
- stejně tak je definována šablona `<xsl:template name="vypis">`, která zastíní šablonu z importovaného souboru

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:str="http://www.kiv.zcu.cz/~herout/XSLTfunkce"
  exclude-result-prefixes="str"
  version="2.0">

  <!-- musi byt prvni instrukce -->
  <xsl:import href="import-volana.xsl"/>

  <xsl:output method="html"
       encoding="UTF-8"/>

  <!-- konflikt stejneho jmena -->
  <xsl:variable name="nadpis" select="'hlavní seznam ovocí:'"/>

  <xsl:template match="/jidlo">
    <html>
      <body>
```

```
        <xsl:value-of select="str:velkaPismena($nadpis)"/>

        <xsl:for-each select="ovoce">
          <xsl:call-template name="vypis">
            <xsl:with-param name="jmenoOvoce" select="nazev"/>
          </xsl:call-template>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>

  <!-- zastinuje puvodni vypis - tento je podrobnejsi -->
  <xsl:template name="vypis">
    <xsl:param name="jmenoOvoce"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="str:velkaPismena($jmenoOvoce)"/>
    <xsl:text> za </xsl:text>
    <xsl:value-of select="$jmenoOvoce/@jednotkovaCena"/>
    <xsl:text> Kč/kg</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

- vypíše

```
<html>
   <body>HLAVNÍ SEZNAM OVOCÍ:
      JABLKA za 10 Kč/kg
      BANÁNY za 25 Kč/kg
   </body>
</html>
```

9.3.2.1. Využití zastíněných šablon

- v objektově orientovaných programovacích jazycích lze zastíněnou metodu volat (např. v Javě pomocí `super`)
- podobnou možnost nám dává i XSLT, ale pouze pro šablony definované s `match` nikoliv s `name`
  - zastíněná šablona se volá pomocí instrukce `<xsl:apply-imports/>`
- XSLT styl `import-volana-zastinena.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:str="http://www.kiv.zcu.cz/~herout/XSLTfunkce"
  version="2.0">

  <xsl:variable name="nadpis" select="'seznam ovocí:'"/>

  <xsl:function name="str:velkaPismena" as="xs:string">
```

```
    <xsl:param name="s" as="xs:string"/>
    <xsl:sequence select="upper-case($s)"/>
  </xsl:function>

  <xsl:template match="nazev">
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="str:velkaPismena(.)"/>
  </xsl:template>

</xsl:stylesheet>
```

- pro známý XML dokument `jidlo-2-ovoce.xml` XSLT styl `import-hlavni-vola.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:str="http://www.kiv.zcu.cz/~herout/XSLTfunkce"
  exclude-result-prefixes="str"
  version="2.0">

  <!-- musi byt prvni instrukce -->
  <xsl:import href="import-volana-zastinena.xsl"/>

  <xsl:output method="html" encoding="UTF-8"/>

  <!-- stejne jmeno -->
  <xsl:variable name="nadpis" select="'hlavní seznam ovocí:'"/>

  <xsl:template match="/jidlo">
    <html>
      <body>
        <xsl:value-of select="str:velkaPismena($nadpis)"/>
         <xsl:apply-templates select="ovoce/nazev"/>
      </body>
    </html>
  </xsl:template>

  <!-- rozsiruje puvodni sablonu, kterou zaroven vola -->
  <xsl:template match="nazev">
    <xsl:apply-imports/>
    <xsl:text> za </xsl:text>
    <xsl:value-of select="@jednotkovaCena"/>
    <xsl:text> Kč/kg</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

- vypíše

```
<html>
   <body>HLAVNÍ SEZNAM OVOCÍ:
      JABLKA za 10 Kč/kg
```

```
        BANÁNY za 25 Kč/kg
    </body>
</html>
```

9.3.3. Tipy pro návrh stylu

- neexistuje obecná metodologie
  - obecně lze říci, že je výhodný iterativní přístup k vytváření stylu (zejména za podpory specializovaných nástrojů typu <oXygen/>)
    - připravit několik základních šablon pro elementy na vyšší úrovni hierarchie
    - po jejich odladění postupně řešit detaily
  - dále je třeba si uvědomit, že při výstupu do HTML či XHTML můžeme pro jemné dolaďování finálního vzhledu často jednoduše použít kaskádní styly místo složitého nastavování všech parametrů v XSLT stylu
- je důležité si uvědomit jakou strukturu má vstupní XML soubor a jakou strukturu požadujeme na výstupu
  1. XML soubor je velmi hierarchický (složité dokumenty) a výstupní soubor má víceméně stejnou strukturu
     - vyplatí se použít velké množství „`match`“ šablon – v podstatě „co element, to šablona“
       - převod je totiž „jen“ změna jedněch tagů za jiné tagy
  2. XML soubor má plochou strukturu a výstupní soubor se strukturou značně liší
     - „přeskládané“ tabulky, množství sumačních (agregačních) informací, spojené informace z různých částí XML dokumentu
     - vyplatí se použít `<xsl:for-each>` a spíše šablony „`name`“
- na samém začátku práce si promyslet možnosti rozdělení stylu do více souborů s využitím `<xsl:include>` a/nebo `<xsl:import>`
  - hned zpočátku určit hlavní soubor a promyslet si případné možnosti parametrizace pomocí parametrů či proměnných či `<xsl:attribute-set>` či konfiguračních souborů načítaných při spuštění stylu
    - to je značně výhodné zejména v případě, kdy uvažujeme o možnosti mít několik různých výstupních formátů

9.4. Práce s více soubory

- principiálně se dá za práci s více soubory považovat již výše uvedené rozdělení stylu do více souborů
- zde se ale budeme zabývat rozdělením „datových“ souborů
  1. vstupní data jsou ve více XML souborech, které je nutné najednou zpracovat
     - zpracování vstupních dat z „doplňujících“ CSV souborů viz dříve

2. výstupní data mají být uložena do více souborů
   - typicky sada provázaných HTML stránek

9.4.1. Načítání více XML souborů

- pro tuto činnost slouží funkce XPath `doc()`

  Varování

  Dříve se jmenovala `document()`. Toto jméno lze stále použít, ale v oficiální dokumentaci (`www.w3.org/TR/xpath-functions/`) se nevyskytuje.

  - zpřístupňuje uzly z jiných XML dokumentů (dále budou nazývány „pomocné“) než je „hlavní“ zpracovávaný vstupní dokument
- můžeme ji použít v libovolném XPath výrazu
  - většinou se načtený obsah ukládá do proměnné, která se dále zpracovává
  - načtené uzly „pomocného“ dokumentu nejsou součástí stromu „hlavního“ vstupního XML dokumentu
    - chceme-li je zpracovat, můžeme použít již známé postupy pomocí `<xsl:for-each>`, `<xsl:call-template>` (zejména s využitím parametrů) nebo dokonce i `<xsl:apply-template>`
- je nutné zdůraznit, že hlavní a pomocný dokument spolu nemusejí nijak hierarchicky souviset
  - typický případ je, že pomocný dokument představuje nějaký typ „číselníku“
- základní použití je `doc('jmenoSouboru.xml')`

  Varování

  Tato funkce načte XML dokument do paměti a vrací odkaz na kořenový uzel `/`. Pokud chceme dál načtené hodnoty zpracovávat, musíme použít již známé XPath výrazy pro adresování vnořených elementů a atributů.

9.4.1.1. Základní použití `doc()` – číselník

- typický případ použití je, když jsou v pomocném XML dokumentu doplňující údaje pro hlavní dokument
  - pomocný XML dokument je „číselník“
- v tomto případě je velmi vhodné načíst číselník do globální proměnné, protože
  - se dá očekávat jeho vícenásobné používání
    - opakované načítání ze souboru by zdržovalo
- s touto proměnnou dále zacházíme jako s kořenovým uzlem dokumentu

- pro hlavní XML dokument `pokladna.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<pokladna>
  <polozka idZbozi="2">
    <mnozstvi jednotka="kus">10</mnozstvi>
  </polozka>
  <polozka idZbozi="1">
    <mnozstvi jednotka="kg">2.5</mnozstvi>
  </polozka>
</pokladna>
```

- pro pomocný XML dokument `ciselnik-zbozi.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<ciselnik>
  <zbozi id="1">
    <nazev>brambory</nazev>
    <jednotkovaCena mena="CZK">20</jednotkovaCena>
  </zbozi>
  <zbozi id="2">
    <nazev>DVD disk</nazev>
    <jednotkovaCena mena="USD">0.5</jednotkovaCena>
  </zbozi>
</ciselnik>
```

- XSLT styl `doc-zaklad.xsl`
  - skutečnost, že v šabloně `<xsl:template match="polozka">` zpracováváme najednou dva XML dokumenty, vyžaduje trochu jiný postup, než doposud
    - cyklus `<xsl:for-each select="$cisZbozi/ciselnik/zbozi">` prochází XML dokument číselníku, takže XPath výrazy jsou platné pro tento dokument
      - přístup k elementu `polozka` musí být uvnitř cyklu zajištěn pomocí lokální proměnné `varPolozka`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="2.0">

  <xsl:output method="html" encoding="UTF-8"/>

  <xsl:variable name="cisZbozi"
      select="doc('ciselnik-zbozi.xml')"/>

  <xsl:template match="/pokladna">
    <html>
      <body>
        <xsl:text>Nákup:</xsl:text>
        <xsl:apply-templates select="polozka"/>
      </body>
    </html>
  </xsl:template>
```

```
  <xsl:template match="polozka">
    <xsl:variable name="varPolozka" select="."/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:for-each select="$cisZbozi/ciselnik/zbozi">
      <xsl:if test="@id = $varPolozka/@idZbozi">
        <xsl:value-of select="nazev"/>
        <xsl:value-of select="$varPolozka/mnozstvi"/>
        <xsl:value-of select="$varPolozka/mnozstvi/@jednotka"/>
      </xsl:if>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

- vypíše (pro jednoduchost nejsou ve stylu ve výpisu oddělovací mezery):

```
<html>
   <body>Nákup:
      DVD disk10kus
      brambory2.5kg
   </body>
</html>
```

9.4.1.2. Test existence XML souboru

- pokud načítáme pomocný soubor, může se stát, že se spleteme ve jménu
  - před načítáním je vhodné otestovat, zda tento soubor existuje
- test lze provést funkcí `doc-available('jmenoSouboru')`
  - vrací `true()` v případě existujícího souboru

- XSLT styl `doc-test.xsl` má zcela stejnou funkčnost, jako předchozí styl
  - jméno pomocného XML dokumentu je uloženo v proměnné `jmenoCisZbozi`
  - při deklaraci proměnné `cisZbozi` se nejdříve otestuje, zda požadovaný soubor existuje
    - pokud ano, je načten známou funkcí `doc()`, jejíž hodnota je vrácena pomocí instrukce `<xsl:copy-of>`

      Varování

      Nepoužít `<xsl:value-of>`, která vrací jen hodnoty elementů!

    - pokud ne, je pomocí instrukce `<xsl:message>` vypsáno chybové hlášení a zpracování je ukončeno

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    version="2.0">

    <xsl:output method="html" encoding="UTF-8"/>

    <xsl:variable name="jmenoCisZbozi" select="'ciselnik-zbozi.xml'"/>

    <xsl:variable name="cisZbozi">
      <xsl:choose>
        <xsl:when test="doc-available($jmenoCisZbozi) = true()">
          <xsl:copy-of select="doc($jmenoCisZbozi)"/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:message terminate="yes">
            <xsl:value-of select="$jmenoCisZbozi"/> - soubor nenalezen
          </xsl:message>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:variable>
  ...
  <!-- dále stejné jako u předchozího stylu -->
```

- při neexistenci souboru (zde `ciselnik-jineho-zbozi.xml`) vypíše:

```
ciselnik-jineho-zbozi.xml - soubor nenalezen
```

9.4.1.3. Větší příklad na načítání více XML souborů

- tento příklad bude rozšířením předchozích příkladů
- pro XML dokument `pokladna.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<pokladna>
  <polozka idZbozi="2">
    <mnozstvi jednotka="kus">10</mnozstvi>
  </polozka>
  <polozka idZbozi="1">
    <mnozstvi jednotka="kg">2.5</mnozstvi>
  </polozka>
</pokladna>
```

- pro XML dokument `ciselnik-zbozi.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<ciselnik>
  <zbozi id="1">
    <nazev>brambory</nazev>
    <jednotkovaCena mena="CZK">20</jednotkovaCena>
  </zbozi>
  <zbozi id="2">
```

```
    <nazev>DVD disk</nazev>
    <jednotkovaCena mena="USD">0.5</jednotkovaCena>
  </zbozi>
</ciselnik>
```

■ pro XML dokument `ciselnik-kurzu-men.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<kurzyMen datum="2009-04-21">
  <mena zkratka="CZK">1</mena>
  <mena zkratka="USD">20.0</mena>
  <mena zkratka="EUR">27.1</mena>
</kurzyMen>
```

■ XSLT styl `doc-dva-soubory.xsl` bude uváděn postupně a komentován

- začátek je zcela standardní

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:funkce="http://www.kiv.zcu.cz/~herout/XSLTfunkce"
  xmlns:saxon="http://saxon.sf.net/"
  extension-element-prefixes="saxon"
  version="2.0">

  <xsl:output method="html" encoding="UTF-8"/>

  <xsl:variable name="jmenoCisZbozi" select="'ciselnik-zbozi.xml'"/>
  <xsl:variable name="jmenoCisKurzu" select="'ciselnik-kurzu-men.xml'"/>
```

- protože se budou načítat dva soubory, je vhodné si pro tuto činnost připravit funkci `nactiDokument()`
  - ♦ ta musí mít definovaný jmenný prostor – zde `funkce`
  - ♦ vrací typ `document-node()`
- pomocí této funkce se jednoduše načtou oba potřebné soubory

```
  <xsl:function name="funkce:nactiDokument" as="document-node()">
    <xsl:param name="jmeno" as="xs:string"/>
    <xsl:choose>
      <xsl:when test="doc-available($jmeno) = true()">
        <xsl:sequence select="doc($jmeno)"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:message terminate="yes">
          <xsl:value-of select="$jmeno"/> - soubor nenalezen </xsl:message>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:function>

  <xsl:variable name="cisZbozi"
```

```
       select="funkce:nactiDokument($jmenoCisZbozi)"/>
<xsl:variable name="cisKurzu"
       select="funkce:nactiDokument($jmenoCisKurzu)"/>
```

- proměnná `celkovaCenaCZK` bude sloužit pro postupnou sumaci ceny a její hodnota bude vypsána na závěr celé transformace

```
<xsl:variable name="celkovaCenaCZK" select="0" saxon:assignable="yes"/>
```

- hlavní šablona, ve které přibyl pouze
  - výpis datumu získaný pomocí funkce `current-date()` a oříznutý
    - bez oříznutí vrací `2009-04-21+02:00`, tj. včetně časové zóny
  - výpis celkové sumy uložené v proměnné `celkovaCenaCZK`

```
<xsl:template match="/pokladna">
  <html>
    <body>
      <xsl:text>Nákup ze dne </xsl:text>
      <xsl:value-of select="substring(string(current-date()),1,10)"/>

      <xsl:apply-templates select="polozka"/>

      <xsl:text>&#xA;Celková cena: </xsl:text>
      <xsl:value-of select="$celkovaCenaCZK"/>
      <xsl:text> CZK</xsl:text>
    </body>
  </html>
</xsl:template>
```

- tato šablona pouze přiřazuje položky z pokladny ke konkrétnímu zboží
  - vlastní výpis je vhodné provést pomocí `<xsl:call-template name="vypis">`, které se předají jednoznačné parametry
    - odpadne tak nutnost zkoumání, nad jakým XML dokumentem se právě pracuje

```
<xsl:template match="polozka">
  <xsl:variable name="varPolozka" select="."/>
  <xsl:text>&#xA;</xsl:text>
  <xsl:for-each select="$cisZbozi/ciselnik/zbozi">
    <xsl:if test="@id = $varPolozka/@idZbozi">
      <xsl:call-template name="vypis">
        <xsl:with-param name="polozka" select="$varPolozka"/>
        <xsl:with-param name="zbozi" select="."/>
      </xsl:call-template>
    </xsl:if>
  </xsl:for-each>
</xsl:template>
```

- výpisová šablona
  - pro zjištění hodnoty v CZK volá šablonu `<xsl:call-template name="cenaCZK">`

```
  <xsl:template name="vypis">
    <xsl:param name="polozka"/>
    <xsl:param name="zbozi"/>
    <xsl:value-of select="$zbozi/nazev"/>
    <xsl:text>: </xsl:text>
    <xsl:value-of select="$polozka/mnozstvi"/>
    <xsl:text> [</xsl:text>
    <xsl:value-of select="$polozka/mnozstvi/@jednotka"/>
    <xsl:text>] po </xsl:text>
    <xsl:value-of select="$zbozi/jednotkovaCena"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="$zbozi/jednotkovaCena/@mena"/>
    <xsl:text> za </xsl:text>
    <xsl:call-template name="cenaCZK">
      <xsl:with-param name="zkrMeny" select="$zbozi/jednotkovaCena/@mena"/>
      <xsl:with-param name="cena"
                      select="$zbozi/jednotkovaCena * $polozka/mnozstvi"/>
    </xsl:call-template>
  </xsl:template>
```

- šablona přiřazuje hodnotu kurzu podle zkratky měny a vypočtenou cenu v CZK vypisuje
- dále tuto cenu sumuje do proměnné `celkovaCenaCZK`

```
  <xsl:template name="cenaCZK">
    <xsl:param name="zkrMeny"/>
    <xsl:param name="cena"/>
    <xsl:for-each select="$cisKurzu/kurzyMen/mena">
      <xsl:if test="@zkratka = $zkrMeny">
        <xsl:variable name="pom" select="number($cena * text())"/>
        <xsl:value-of select="$pom"/>
        <xsl:text> CZK</xsl:text>
        <saxon:assign name="celkovaCenaCZK"
                      select="$celkovaCenaCZK + $pom"/>
      </xsl:if>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

- vypíše:

```
<html
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:funkce="http://www.kiv.zcu.cz/~herout/XSLTfunkce">
   <body>Nákup ze dne 2009-04-21
      DVD disk: 10 [kus] po 0.5 USD za 100 CZK
      brambory: 2.5 [kg] po 20 CZK za 50 CZK
      Celková cena: 150 CZK
   </body>
</html>
```

Poznámka

Pokud načítáme více rozsáhlejších vstupních XML dokumentů, může se stát, že XSLT procesoru dojde paměť. Tomu se dá čelit použitím funkce `saxon:discard-document()`, která dokument načte a při dalším načítání uvolní z paměti předchozí načtený dokument.

9.4.2. Generování více výstupních souborů

- často se setkáváme s potřebou, že výsledkem XSLT transformace má být několik výstupních souborů
- to lze poměrně snadno zajistit pomocí instrukce `<xsl:result-document>`
  - ta výstup posílá do souboru, který je určen jejím atributem `href`
  - instrukce má množství dalších atributů, které se víceméně shodují s atributy známé instrukce `<xsl:output>`
    - využijeme zejména atributy `method`, `encoding` a `indent`

9.4.2.1. Jednoduchý příklad

- pro XML dokument `jidlo-2-ovoce.xml`

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

- XSLT styl `vystup-do-souboru-jednoduchy.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="2.0">

  <xsl:template match="/">
    <xsl:for-each select="jidlo/ovoce">
      <xsl:result-document href="{concat(nazev, '.xml')}"
                           method="html" encoding="UTF-8">
        <html>
          <body>
            <xsl:value-of select="nazev"/>
            <xsl:text> za </xsl:text>
            <xsl:value-of select="nazev/@jednotkovaCena"/>
            <xsl:text> Kč</xsl:text>
          </body>
```

```
        </html>
      </xsl:result-document>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

- dále vytvoří soubor `jablka.html`

```
<html>
   <body>jablka za 10 Kč</body>
</html>
```

- dále vytvoří soubor `banány.html`

```
<html>
   <body>banány za 25 Kč</body>
</html>
```

9.4.2.2. Více výstupních souborů vzájemně provázaných

- často mají být generované HTML soubory vzájemně provázány
  - to lze zajistit HTML příkazem `<a href="URL">`

- pro XML dokument `pokladna.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<pokladna>
  <polozka idZbozi="2">
    <mnozstvi jednotka="kus">10</mnozstvi>
  </polozka>
  <polozka idZbozi="1">
    <mnozstvi jednotka="kg">2.5</mnozstvi>
  </polozka>
</pokladna>
```

- pro XML dokument `ciselnik-zbozi.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<ciselnik>
  <zbozi id="1">
    <nazev>brambory</nazev>
    <jednotkovaCena mena="CZK">20</jednotkovaCena>
  </zbozi>
  <zbozi id="2">
    <nazev>DVD disk</nazev>
    <jednotkovaCena mena="USD">0.5</jednotkovaCena>
  </zbozi>
</ciselnik>
```

■ XSLT styl `vystup-do-souboru.xsl` bude uváděn postupně a komentován

- začátek je zcela stejný, jako dříve uvedený příklad ve stylu `doc-dva-soubory.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:funkce="http://www.kiv.zcu.cz/~herout/XSLTfunkce"
  exclude-result-prefixes="xs funkce"
  version="2.0">

  <xsl:output method="html" encoding="UTF-8"/>

  <xsl:variable name="jmenoCisZbozi" select="'ciselnik-zbozi.xml'"/>

  <xsl:function name="funkce:nactiDokument" as="document-node()">
    <xsl:param name="jmeno" as="xs:string"/>
    <xsl:choose>
      <xsl:when test="doc-available($jmeno) = true()">
        <xsl:sequence select="doc($jmeno)"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:message terminate="yes">
          <xsl:value-of select="$jmeno"/> - soubor nenalezen </xsl:message>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:function>

  <xsl:variable name="cisZbozi"
                    select="funkce:nactiDokument($jmenoCisZbozi)"/>
```

- funkce pro přípravu jména generovaného HTML souboru ve formátu `zbozi1.html`

```
  <xsl:function name="funkce:jmenoSouboru" as="xs:string">
    <xsl:param name="cislo" as="xs:integer"/>
    <xsl:sequence select="concat('zbozi', $cislo, '.html')"/>
  </xsl:function>
```

- hlavní šablona, která svým prvním příkazem `<xsl:call-template name="souboryZbozi"/>` generuje všechny HTML soubory

```
  <xsl:template match="/pokladna">
    <xsl:call-template name="souboryZbozi"/>
    <html>
      <body>
        <table border="1">
          <xsl:apply-templates select="polozka"/>
        </table>
      </body>
    </html>
  </xsl:template>
```

- šablona, která vytváří jednu řádku tabulky
  - ♦ v prvním sloupci je ID zboží, které odkazuje na odpovídající HTML soubor
  - ♦ ve druhém sloupci je množství zboží

```
<xsl:template match="polozka">
  <xsl:variable name="varPolozka" select="."/>
  <xsl:for-each select="$cisZbozi/ciselnik/zbozi">
    <xsl:if test="@id = $varPolozka/@idZbozi">
      <tr>
        <td>
          <a>
            <xsl:attribute name="href">
              <xsl:value-of
                  select="funkce:jmenoSouboru($varPolozka/@idZbozi)"/>
            </xsl:attribute>
            <xsl:value-of select="$varPolozka/@idZbozi"/>
          </a>
        </td>
        <td>
          <xsl:value-of select="$varPolozka/mnozstvi"/>
        </td>
      </tr>
    </xsl:if>
  </xsl:for-each>
</xsl:template>
```

- šablona vytvářející HTML soubor zboží pomocí instrukce `<xsl:result-document>`

```
<xsl:template name="souboryZbozi">
  <xsl:for-each select="$cisZbozi/ciselnik/zbozi">
    <xsl:result-document href="{funkce:jmenoSouboru(@id)}">
      <html>
        <body>
          <xsl:value-of select="nazev"/>
          <xsl:text> za </xsl:text>
          <xsl:value-of select="jednotkovaCena"/>
          <xsl:text> </xsl:text>
          <xsl:value-of select="jednotkovaCena/@mena"/>
        </body>
      </html>
    </xsl:result-document>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

■ vypíše:

```
<html>
   <body>
      <table border="1">
```

```
          <tr>
             <td><a href="zbozi2.html">2</a></td>
             <td>10</td>
          </tr>
          <tr>
             <td><a href="zbozi1.html">1</a></td>
             <td>2.5</td>
          </tr>
       </table>
    </body>
</html>
```

- dále vytvoří soubor `zbozi1.html`

```
<html>
   <body>brambory za 20 CZK</body>
</html>
```

- dále vytvoří soubor `zbozi2.html`

```
<html>
   <body>DVD disk za 0.5 USD</body>
</html>
```

9.5. Regulární výrazy

- dávají nám značné možnosti při zpracování (a případné kontrole) řetězců
- používají se např. v instrukci `<xsl:analyze-string>`, podrobně

  `http://www.w3.org/TR/xslt20/#analyze-string`

- delší příklad, který využije mnoho znalostí z dřívějška, ukáže, jak pomocí pojmenované šablony převést do XML dokumentu CSV soubor
  - tato akce může být mnohdy užitečná, protože výsledný XSLT styl je poměrně krátký – rozhodně kratší, nežli by byl program v běžném programovacím jazyce
- vstupní soubor `jidlo.csv`
  - je v kódování UTF-8 bez BOM (ale BOM nevadí)
  - řádky jsou ukončeny <CR><LF>
  - na konci řádky může být libovolný počet bílých znaků
  - ve jménech ovocí jsou vícenásobné mezery na začátku, na konci i uprostřed

```
1;10;jablka  červená;2.5
2;25;  banány  ;2
```

■ XSLT styl `regularni-vyraz.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="2.0">

  <xsl:output method="xml" encoding="UTF-8"
    indent="yes"/>
```

- atribut `indent="yes"` zajistí automatické odřádkování a odsazování výstupního XML dokumentu

```
  <xsl:param name="kodovani" select="'UTF-8'"/>
  <xsl:param name="jmenoSouboru" select="'jidlo.csv'"/>
  <xsl:variable name="obsahSouboru"
         select="unparsed-text($jmenoSouboru, $kodovani)"/>
```

- parametry `kodovani` a `jmenoSouboru` jsou globální, což umožňuje jejich případné nastavení z příkazové řádky – podrobně viz dříve
- proměnná `obsahSouboru` obsahuje načtený obsah souboru

```
  <xsl:variable name="radky">
    <xsl:analyze-string select="$obsahSouboru" regex="^.*$" flags="m">
      <xsl:matching-substring>
        <ovoce>
          <xsl:value-of select="normalize-space(.)"/>
        </ovoce>
      </xsl:matching-substring>
    </xsl:analyze-string>
  </xsl:variable>
```

- obsah proměnné `obsahSouboru` bude pomocí regulárního výrazu parsován a výsledek bude zapsán do proměnné `radky`
  - ♦ pro parsování se používá instrukce `<xsl:analyze-string>` s parametry:
    - – `select="$obsahSouboru"` – co se parsuje
    - – `regex="^.*$"` – regulární výraz, podle kterého se parsuje s významem:
      - `^` – od začátku řádky
      - `.*` – libovolné znaky na řádce, přičemž tam nemusí být ani jeden (`*`), tzn. jednalo by se o prázdnou řádku
      - `$` – konec řádky – je to vždy jen <LF>, případné <CR> se odstraní dále
    - – `flags="m"` – `m` znamená *multi-line mode* – podrobnosti na

      `http://www.w3.org/TR/xpath-functions/#flags`
  - ♦ pokud řetězec vyhovuje regulárnímu výrazu, provede se část `<xsl:matching-substring>`
    - – zde se každá řádka „obalí“ do elementu `<ovoce>`

– hodnotou elementu je obsah řádky zbavený případných úvodních a počátečních bílých znaků pomocí funkce `normalize-space()`

• zde mezery na konci řádky a zejména <CR> na konci řádky

● proměnná `obsahSouboru` obsahuje řetězec, kdežto proměnná `radky` seznam uzlů !

♦ nelze ji bezproblémově vypsat pro ladicí účely

```
  <xsl:template name="prevodDoXML">
    <jidlo>
      <xsl:for-each select="$radky/ovoce">
        <xsl:analyze-string select="." regex="^(\d+);(\d+);(.+);(.+)\s*$">
          <xsl:matching-substring>
            <ovoce cislo="{regex-group(1)}">
              <nazev jednotkovaCena="{regex-group(2)}">
                <xsl:value-of select="normalize-space(regex-group(3))"/>
              </nazev>
              <vaha>
                <xsl:value-of select="regex-group(4)"/>
              </vaha>
            </ovoce>
          </xsl:matching-substring>
          <xsl:non-matching-substring>
            <xsl:message>
              <xsl:text>Chyba ve vstupních datech: </xsl:text>
              <xsl:value-of select="."/>
            </xsl:message>
          </xsl:non-matching-substring>
        </xsl:analyze-string>
      </xsl:for-each>
    </jidlo>
  </xsl:template>

</xsl:stylesheet>
```

● pojmenovaná šablona `prevodDoXML`, která bude volána z příkazové řádky

● provádí finální výstup do XML souboru, kde kořenovým elementem je `jidlo`

● instrukce `<xsl:for-each>` prochází seznam uzlů v proměnné `radky` a zpracovává každý element `ovoce` – `select="$radky/ovoce"`

● pro parsování řádky s údaji o jednotlivých ovocích se opět používá `<xsl:analyze-string>`

♦ `regex="^(\d+);(\d+);(.+);(.+)\s*$"` má význam

– `^` – od začátku řádky

– `(\d+)` – alespoň jedna dekadická číslice – zde ve významu pořadového čísla

– `;` – oddělovací znak regulárního výrazu v CSV souboru

– `(\d+)` – alespoň jedna dekadická číslice – zde ve významu jednotkové ceny

- `(.+)` – alespoň jeden znak – zde ve významu názvu ovoce
  - na jeho začátku a konci se může vyskytnout vždy jedna nevýznamová mezera
  - pokud byl v CSV souboru větší počet mezer i uvnitř řádky (nikoliv jen na začátku a na konci), byly odstraněny už při konstrukci proměnné `radky` funkcí `normalize-space()`
- `(.+)` – alespoň jeden znak – zde ve významu váhy
  - není možné použít `(\d+)`, protože údaj o váze je reálné číslo – obsahuje desetinný oddělovač znak `.`
- `\s*` – libovolný počet bílých znaků na konci řádky – zde by se již neměly vyskytovat díky předchozí `normalize-space()`
- `$` – konec řádky

● pokud řetězec vyhovuje regulárnímu výrazu, provede se část `<xsl:matching-substring>`

♦ zde se již zapisují jednotlivé elementy a atributy výstupního XML dokumentu

♦ jednotlivé naparsované části získáme pomocí funkce `regex-group()`

– pro hodnotu atributů musí být uvedena v `{ }`

♦ u názvu ovoce odstraníme případnou úvodní a počáteční mezeru pomocí `normalize-space()`

● pokud řetězec nevyhovuje regulárnímu výrazu, provede se část `<xsl:non-matching-substring>`

♦ instrukcí `<xsl:message>` vypíše chybové hlášení a hodnotu řádky, na které nastala chyba – tzn. nevyhovoval pro ni regulární výraz

■ příkaz:

```
saxon -o jidlo-csv.xml -it prevodDoXML regularni-vyraz.xsl
```

vytvoří soubor `jidlo-csv.xml` s obsahem:

```
<?xml version="1.0" encoding="UTF-8"?>
<jidlo>
   <ovoce cislo="1">
      <nazev jednotkovaCena="10">jablka červená</nazev>
      <vaha>2.5</vaha>
   </ovoce>
   <ovoce cislo="2">
      <nazev jednotkovaCena="25">banány</nazev>
      <vaha>2</vaha>
   </ovoce>
</jidlo>
```

9.6. Volání externích funkcí

■ ačkoliv mají XPath 2.0 a XSLT 2.0 k dispozici širokou škálu možností a funkcí, postupem času jistě budeme potřebovat něco, co v XSLT (a v XPath) lze provést obtížně nebo vůbec

- pak můžeme využít možnosti externích funkcí

- XSLT procesor Saxon je napsaný v Javě
  - můžeme bez větších problémů využít třídy a metody z Java Core API
  - můžeme dokonce využít i vlastní programový kód
- pro volání externích funkcí není v XSLT žádná speciální instrukce
  - využívá se jen prostředků, které již známe
- všechny dále uvedené příklady budou využívat pojmenované šablony spuštěné z příkazové řádky (viz dříve), např.:

```
saxon -it math externi-funkce-math-konst.xsl
```

  - toto samozřejmě není podmínka jejich využívání

9.6.1. Volání statických metod z JavaCore API

- statické metody z Java Core API lze použít zcela bezproblémově
  - dávají nám např. mnohem větší možnosti formátování řetězců, matematických výpočtů atp.
    - je možné dokonce využít i statických konstant, jako je např. `Math.PI`
- XSLT styl `externi-funkce-math.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:math="java.lang.Math"
  extension-element-prefixes="math"
  version="2.0">

  <xsl:output method="text" encoding="UTF-8" indent="yes"/>

  <xsl:template name="math">
    <xsl:text>Pi = </xsl:text>
    <xsl:value-of select="math:PI()"/>
    <xsl:text>&#xA;E = </xsl:text>
    <xsl:value-of select="math:E()"/>
    <xsl:text>&#xA;2^10 = </xsl:text>
    <xsl:value-of select="math:pow(2, 10)"/>
    <xsl:text>&#xA;ln(E) = </xsl:text>
    <xsl:value-of select="math:log(math:E())"/>
  </xsl:template>

</xsl:stylesheet>
```

  - důležité je zavést nový jmenný prostor, zde:

```
xmlns:math="java.lang.Math"
```

    - jméno prefixu si můžeme zvolit libovolně, zde `math`

– toto jméno pak bude při použití nahrazovat jméno třídy

♦ hodnota jmenného prostoru musí být přesné pojmenování Java třídy včetně balíku, do kterého patří, zde `java.lang.Math`

● metody třídy se nevolají pomocí tečkové notace jako v Javě (`Math.pow(2, 10)`), ale pomocí jmenného prostoru a dvojtečky, např. `math:pow(2, 10)`

♦ skutečné parametry volání se píší dle běžných zvyklostí

● použijeme-li statickou konstantu, musíme ji volat s přídavnými závorkami jako funkci, zde `math:PI()`

■ po spuštění příkazem

```
saxon -it math externi-funkce-math.xsl
```

vypíše:

```
Pi = 3.141592653589793
E = 2.718281828459045
2^10 = 1024
ln(E) = 1
```

9.6.2. Volání instančních metod z JavaCore API

■ je možné využívat i instanční metody, ovšem jejich použití je poněkud netypické

● prvním parametrem instanční metody je totiž volání konstruktoru této třídy

■ příklad ukáže i použití funkcí z XSLT, kterými můžeme „obalit“ volání externích funkcí

● tím lze pro další použití zcela odstínit problematiku volání externích funkcí

■ XSLT styl `externi-funkce-string.xsl`

Varování

Tento příklad funguje jen pro Saxon 9, nikoliv pro Saxon 8 a nižší.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:str="java.lang.String"
  xmlns:kal="java.util.GregorianCalendar"
  xmlns:funkce="http://www.kiv.zcu.cz/~herout/XSLTfunkce"
  extension-element-prefixes="str kal"
  version="2.0">

  <xsl:output method="text" encoding="IBM852" indent="yes"/>

  <xsl:function name="funkce:hexaVypis" as="xs:string">
    <xsl:param name="cislo" as="xs:integer"/>
    <xsl:sequence select="str:format('%04X', $cislo)"/>
  </xsl:function>
```

```
  <xsl:template name="string">
    <xsl:text>Hex = </xsl:text>
    <xsl:value-of select="funkce:hexaVypis(26)"/>
    <xsl:text>&#xA;Oct = </xsl:text>
    <xsl:value-of select="str:format('%o', 26)"/>
    <xsl:text>&#xA;den = </xsl:text>
    <xsl:value-of select="str:format('%tA', kal:new())"/>
    <xsl:text>&#xA;měsíc = </xsl:text>
   <xsl:value-of select="str:format('%tB', kal:new(2009, kal:APRIL(), 21))"/>
    <xsl:text>&#xA;přestupný rok 2008 = </xsl:text>
    <xsl:value-of select="kal:isLeapYear(kal:new(), 2008)"/>
  </xsl:template>

</xsl:stylesheet>
```

- jako příklad třídy s instančními metodami bude použita třída `java.util.GregorianCalendar` se jmenným prostorem `kal`
- výstupní kódování je `IBM852`, aby se v příkazové řádce správně zobrazovaly akcentované znaky
- funkce `funkce:hexaVypis()` je běžná XSLT funkce, ovšem využívající značných možností formátování poskytovaných metodou `String.format()`
  - ♦ toto považuji za velmi užitečnou možnost, protože pomocí `String.format()` lze řešit prakticky všechny požadavky na libovolně sofistikované formátování řetězců
- pro ukázku je uvedeno použití metody `String.format()` přímo v šabloně – oktalový výpis čísla
- příkaz: `<xsl:value-of select="str:format('%tA', kal:new())"/>`
  - ♦ ukazuje použití bezparametrického konstruktoru třídy `java.util.GregorianCalendar`
  - ♦ Java formát výpisu `'%tA'` znamená „vypiš celý název dne“
- další příklad je použití parametrického konstruktoru
  - ♦ Java formát výpisu `'%tB'` znamená „vypiš celý název měsíce“
- poslední příkaz je nejméně pochopitelný: `kal:isLeapYear(kal:new(), 2008)`
  - ♦ ukazuje použití instanční metody `isLeapYear()` ze třídy `GregorianCalendar` pro zjištění, zda je rok přestupný
    - – tato metoda má v Javě pouze jeden parametr – zkoumaný rok, což je v XSLT druhý parametr
    - – v XSLT má dva parametry, přičemž první z nich je volání konstruktoru třídy `GregorianCalendar`

Poznámka

V Javě by toto volání bylo zapsáno např. jako:

```
new GregorianCalendar().isLeapYear(2008)
```

- po spuštění příkazem

  ```
  saxon -it string externi-funkce-string.xsl
  ```

  vypíše:

  ```
  Hex = 001A
  Oct = 32
  den = Úterý
  měsíc = duben
  přestupný rok 2008 = true
  ```

9.6.3. Volání metod vlastních Java tříd

- při volání externích funkcí se není třeba omezovat jen na knihovní metody Java Core API
  - je možné připravit libovolnou vlastní třídu s požadovanými metodami
- při přípravě metod musíme mít na paměti, že pokud mají být v XSLT použity jako funkce, je vhodné v Javě vracet nejlépe primitivní datové typy, nebo typ `String`
  - pokud chceme vracet jiné typy musí vyhovovat typům známým z XSD
  - je samozřejmě možné připravit třídy a metody, které budou výsledek své práce ukládat ve formě souborů na disk a tudíž nemusejí do XSLT nic vracet
    - jednalo by se např. o generování grafických souborů provázaných s HTML výstupem XSLT transformace
- program v Javě se píše zcela podle pravidel programování v Javě, tzn. nemá žádné zvláštnosti
  - je vhodné, aby obsahoval metodu `main()`, která se při spuštění z JVM použije pro otestování správné činnosti Java programu
  - metoda `main()` ale není v XSLT nijak využívána
- pro XSLT jsou důležité `.class` soubory vzniklé překladem
  - zde je nezbytné, aby byly uloženy v adresáři, který je nastaven v CLASSPATH při spuštění Saxonu
  - typicky (při našich pokusech) se nacházejí v adresáři, ve kterém je též soubor XSL (aktuální adresář)
    - bohužel aktuální adresář (`.`) často není uveden v dávkovém souboru, kterým se XSLT transformace spouští
  - spuštění transformace, kdy `.class` soubory jsou ve stejném (tj. aktuálním) adresáři, jako `.xsl` soubor, tedy může vypadat takto:

    ```
    java -cp "C:\Program Files\Java\saxon\saxon9.jar";. net.sf.saxon.Transform ►
    -it mujFaktorial faktorial-java.xsl
    ```

    - kde důležité je závěrečné `;.` v parametru `-cp`
  - spuštění transformace, kdy `.class` soubory jsou v adresáři `D:\pokusy`, může vypadat takto:

```
java -cp "C:\Program Files\Java\saxon\saxon9.jar";D:\pokusy ►
net.sf.saxon.Transform -it mujFaktorial faktorial-java.xsl
```

- `.class` soubory mohou být zabaleny do JAR souboru, např. do `mujjar.jar`:

```
java -cp "C:\Program Files\Java\saxon\saxon9.jar";mujjar.jar ►
net.sf.saxon.Transform -it mujFaktorial faktorial-java.xsl
```

- pro intenzivnější pokusy je ovšem nejlepší doplnit adresář, kde jsou `.class` soubory, přímo do dávkového souboru `saxon.bat`, jímž se spouští Saxon
  - jeho obsah je např.:

```
@java -cp C:\Progra~1\Java\saxon\saxon9.jar; ►
C:\Progra~1\Java\saxon\saxon9-s9api.jar; ►
C:\Progra~1\Java\saxon\saxon9-dom.jar; ►
 net.sf.saxon.Transform %1 %2 %3 %4 %5 %6 %7 %8 %9
```

  - pak je spuštění:

```
saxon -it mujFaktorial faktorial-java.xsl
```

- Java soubor `Faktorial.java`
  - metoda `public static void main(String[] args)` je použita jen pro ověření funkčnosti

```
import java.math.*;

public class Faktorial {

  public String faktorial(int n) {
    BigDecimal bd = new BigDecimal("1");
    for (int i = 2;  i <= n;  i++) {
      bd = bd.multiply(new BigDecimal(i));
    }
    return bd.toString() + "\n";
  }

  public static void main(String[] args) {
    System.out.println(new Faktorial().faktorial(20));
  }
}
```

- XSLT styl `faktorial-java.xsl`
  - má definovaný jmenný prostor `moje` jehož hodnotou je jméno Java třídy `Faktorial`
    - třída `Faktorial` neleží v žádném balíku, takže se uvádí jen její jméno
  - volání metody `faktorial()` je podobné volání metody `isLeapYear()` – viz výše

- ♦ první parametr představuje volání konstruktoru
- ♦ druhý parametr je hodnota, ze které se bude faktoriál počítat

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:moje="Faktorial"
    extension-element-prefixes="moje"
    version="2.0">

<xsl:output method="text"/>

<xsl:template name="mujFaktorial">
  <xsl:value-of select="moje:faktorial(moje:new(), 20)"/>
</xsl:template>

</xsl:stylesheet>
```

- po spuštění příkazem

```
saxon -it mujFaktorial faktorial-java.xsl
```

vypíše:

```
2432902008176640000
```

9.6.4. Praktický příklad

- tento příklad je ukázkou akce, kterou nelze provést jen prostředky jazyka XSLT 2.0
- vypíše přehlednou tabulku jmen souborů a jejich velikostí
  - soubory jsou hledány v zadaném adresáři – přednastaven je aktuální adresář .
  - soubory musí mít zvolenou příponu – přednastavena je `xsl`
- Java soubor `Soubory.java`
  - využívá třídu `BigDecimal`, která umí počítat v libovolné přesnosti
  - metoda `public static void main(String[] args)` je použita jen pro ověření funkčnosti

```
/**
 * Ukazkovy soubor pro spojeni Javy a XSLT
 * Zjistuje jmena a velikosti vybranych souboru ve zvolenem adresari
 * @author P.Herout
 * @version 2009-04-21
 */

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

```
/**
 * Hlavni trida
 */
public class Soubory {

  /**
   * vnitrni trida
   * slouzi pro filtrovani nalezenych souboru podle jejich pripony
   * pripona se zadava v konstruktoru
   */
  class FiltrPripony implements FilenameFilter{
    private String pripona;
    FiltrPripony(String pripona) {
      this.pripona = pripona;
    }

    /**
     * Filtr podle pripony
    * @param nevyuzito nevyuzity parametr - je vyzadovan definici metody v ►
rozhrani
     * @param jmeno cele jmeno souboru
     * @return true, pokud pripona souboru vyhovuje zvolene pripone
     */
    public boolean accept(File nevyuzito, String jmeno) {
      if (jmeno.endsWith(this.pripona) == true) {
        return true;
      }
      else {
        return false;
      }
    }
  }

  /**
   * zjisti jmena souboru dane pripony v danem adresasi
   * a vraci nazvy souboru a jejich velikosti oddelene strednikem
   * @param adresar kde se soubory hledaji
   * @param pripona hledana pripona souboru
   * @return retezec, udaje o kazdem souboru jsou na samostatne radce, ►
jmeno_souboru;velikost
   */
  public String jmenaSouboru(String adresar, String pripona) {
    File dir = new File(adresar);
    File[] soubory = dir.listFiles(new FiltrPripony(pripona));
    StringBuffer sb = new StringBuffer(1000);
    for (int i = 0;  i < soubory.length;  i++) {
      sb.append(soubory[i].getName() + ";" + soubory[i].length()+ "\n");
    }
    return sb.toString();
  }

  /* jen pro vyzkouseni */
  public static void main(String[] args) {
```

```
    System.out.println(new Soubory().jmenaSouboru(".", "class"));
  }
}
```

- XSLT styl `externi-funkce-soubory.xsl`
  - využívá znalostí uvedených dříve, nepřidává nic principiálně nového
  - v proměnné `infoSoubory` je uložen výsledek volání Javovské metody `JmenaSouboru()` jako jeden řetězec
  - v proměnné `radky` je tento řetězec převeden na uzel `soubory`
  - v šabloně `<xsl:template name="vypis">` jsou jednotlivé řádky rozparsovány a vypsány do výstupního HTML

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:moje="Soubory"
  extension-element-prefixes="moje"
  version="2.0">

  <xsl:output method="xml" encoding="UTF-8" indent="yes"/>

  <xsl:param name="pripona" select="'xsl'"/>
  <xsl:param name="adresar" select="'.'"/>

  <xsl:variable name="infoSoubory"
    select="moje:jmenaSouboru(moje:new(), $adresar, $pripona)"/>

  <xsl:variable name="radky">
    <xsl:analyze-string select="$infoSoubory" regex="^.*$" flags="m">
      <xsl:matching-substring>
        <soubory>
          <xsl:value-of select="normalize-space(.)"/>
        </soubory>
      </xsl:matching-substring>
    </xsl:analyze-string>
  </xsl:variable>

  <xsl:template name="souboryVelikosti">
    <html>
      <body>
        <table border="1">
          <xsl:call-template name="vypis"/>
        </table>
      </body>
    </html>
  </xsl:template>

  <xsl:template name="vypis">
    <xsl:for-each select="$radky/soubory">
```

```
      <xsl:analyze-string select="." regex="^(.+);(\d+)\s*$">
        <xsl:matching-substring>
          <tr>
            <td>
              <xsl:value-of select="regex-group(1)"/>
            </td>
            <td align="right">
              <xsl:value-of select="regex-group(2)"/>
            </td>
          </tr>
        </xsl:matching-substring>
      </xsl:analyze-string>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

- po spuštění příkazem

```
saxon -it souboryVelikosti externi-funkce-soubory.xsl "pripona=xml"
```

vypíše např.:

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
   <body>
      <table border="1">
         <tr>
            <td>ciselnik-kurzu-men.xml</td>
            <td align="right">186</td>
         </tr>
         <tr>
            <td>ciselnik-zbozi.xml</td>
            <td align="right">288</td>
         </tr>
         <tr>
            <td>jidlo-2-ovoce.xml</td>
            <td align="right">265</td>
         </tr>
         <tr>
            <td>jidlo-4-ovoce.xml</td>
            <td align="right">488</td>
         </tr>
         <tr>
            <td>pokladna.xml</td>
            <td align="right">231</td>
         </tr>
      </table>
   </body>
</html>
```

- výsledek práce lze samozřejmě zapsat i do souboru (zde `javax-xpath.html`) a lze změnit i prohledávaný adresář, např. (všechny údaje jsou ve skutečnosti na jedné řádce):

```
saxon -o javax-xpath.html -it souboryVelikosti externi-funkce-soubory.xsl ►
"pripona=html" ►
"adresar=C:\Program Files\Java\jdk1.6.0_03\docs\api\javax\xml\xpath"
```

9.7. Spouštění XSLT transformace z programu v Javě

- XSLT transformaci není nutné volat jen z příkazové řádky
- protože množství dostupných XSLT procesorů je napsáno v Javě (např. Saxon ano), je možné transformaci bez problémů spouštět z Java programu
- transformace bude ověřována na jednoduchém stylu, který obsahuje:
  - globální parametr `vstupniParametr` pro nastavení z vnějšku transformace
  - šablonu `<xsl:template match="/">` pro transformaci nad vstupním XML dokumentem (na jehož obsahu zde nezáleží)
  - šablonu `<xsl:template name="pojmenovanaSablona">` pro transformaci nezávislou na vstupním XML dokumentu
  - nastavení výstupu instrukcí `<xsl:output>`, ve které se budou lišit dva styly
    - `transformace-z-Javy-html.xsl` s obsahem:

      ```
      <xsl:output method="html" indent="yes" encoding="UTF-8"/>
      ```

    - `transformace-z-Javy-xhtml.xsl` s obsahem:

      ```
      <xsl:output method="xhtml" indent="yes" encoding="UTF-8"/>
      ```

      tato transformace půjde spustit jen s XSLT procesorem, který umí XSLT 2.0

- XSLT styl `transformace-z-Javy-html.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">

  <xsl:output method="html" indent="yes" encoding="UTF-8"/>

  <xsl:param name="vstupniParametr" select="'defaulní nastavení'"/>

  <xsl:template match="/">
    <html>
      <body>
        <xsl:text>transformace nad konkrétním XML souborem</xsl:text>
```

```
      </body>
    </html>
  </xsl:template>

  <xsl:template name="pojmenovanaSablona">
    <html>
      <body>
        <xsl:text>transformace pomocí pojmenované šablony se vstupním ►
parametrem: </xsl:text>
        <xsl:value-of select="$vstupniParametr"/>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

- začátek XSLT stylu `transformace-z-Javy-xhtml.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="2.0">

  <xsl:output method="xhtml" indent="yes" encoding="UTF-8"/>

  ... <!-- zbytek stejný -->
```

Poznámka

Pokusy ukázaly, že kódování akcentů je pro všechny následující pokusy vždy správné. Nebude tedy dále zmiňováno.

Poznámka

Výjimky vzniklé při transformaci budou ve všech příkladech pro jednoduchost ošetřeny deklarací.

9.7.1. Transformace s využitím JAXP a Java Core API Xalan

- v Java Core API je pro transformace připraveno rozhraní JAXP – viz dříve
  - využívá se známé třídy `javax.xml.transform.Transformer`
    - tu implementuje XSLT procesor Xalan
      - má problémy s verzí XSLT 2.0
  - protože všechna potřebná nastavení pro výstup transformace jsou uvedena přímo v XSLT stylu, nemá význam je znovu nastavovat pomocí vlastností, např.:

```
zapisovac.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "no");
zapisovac.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
```

```
zapisovac.setOutputProperty(OutputKeys.STANDALONE, "yes");
zapisovac.setOutputProperty(OutputKeys.INDENT, "yes");
zapisovac.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", ►
"2");
```

- Java soubor `XsltJaxp.java`
  - připojení souboru s XSLT stylem se děje při vytváření instance třídy `Transformer`

```
Transformer zapisovac = tf.newTransformer(new StreamSource(XSLT_STYL));
```

  - celý zbytek transformace je pak stejný, jako pro DOM či StAX

```
import java.io.File;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

/**
 * Ukazka XSLT transformace s procesorem z JAXP
 */
public class XsltJaxp {

  static final String ADRESAR = "d:/xslt/12/";
  static final String XSLT_STYL = ADRESAR + "transformace-z-Javy-html.xsl";
  static final String XML_VSTUP = ADRESAR + "jidlo-2-ovoce.xml";
  static final String HTML_VYSTUP = ADRESAR + "jaxp-html.html";

  public static void main(String[] args) throws Exception {
    TransformerFactory tf = TransformerFactory.newInstance();
    Transformer zapisovac = tf.newTransformer(new StreamSource(XSLT_STYL));
    zapisovac.transform(new StreamSource(new File(XML_VSTUP)),
                        new StreamResult(new File(HTML_VYSTUP)));
  }
}
```

- vytvoří soubor `jaxp-html.html` s obsahem:

```
<html>
   <body>transformace nad konkrétním XML souborem</body>
</html>
```

9.7.2. Transformace s využitím JAXP a Saxon

- skutečnost, že máme k dispozici Saxon (např. v souboru `saxon9.jar`), lze v Java programu využít
  - Saxon samozřejmě splňuje rozhraní JAXP, takže implicitní Xalan lze nahradit Saxonem
  - Saxon je v balíku `net.sf.saxon`

- musíme zajistit, aby soubor `saxon9.jar` byl při spuštění v CLASSPATH
  - např. přidáním `saxon9.jar` do projektu v Eclipse

- Java soubor `XsltJaxp.java`
  - nahrazení Xalanu pomocí Saxonu se provede v nastavení systémových proměnných

```
Properties props = System.getProperties();
props.put("javax.xml.transform.TransformerFactory",
          "net.sf.saxon.TransformerFactoryImpl");
System.setProperties(props);
```

  - celý zbytek transformace je pak stejný, jako u předchozího příkladu, ovšem můžeme ve stylu použít XSLT 2.0 – zde reprezentované výstupem do XHTML

```
import java.io.File;
import java.util.Properties;

import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;


/**
 * Ukazka XSLT transformace z JAXP s procesorem Saxon
 */
public class XsltJaxpSaxon {

  static final String ADRESAR = "d:/xslt/12/";
  static final String XSLT_STYL = ADRESAR + "transformace-z-Javy-xhtml.xsl";
  static final String XML_VSTUP = ADRESAR + "jidlo-2-ovoce.xml";
  static final String HTML_VYSTUP = ADRESAR + "jaxp-saxon.html";

  public static void main(String[] args) throws Exception {
    Properties props = System.getProperties();
    props.put("javax.xml.transform.TransformerFactory",
              "net.sf.saxon.TransformerFactoryImpl");
    System.setProperties(props);

    TransformerFactory tf = TransformerFactory.newInstance();
    Transformer zapisovac = tf.newTransformer(new StreamSource(XSLT_STYL));
    zapisovac.transform(new StreamSource(new File(XML_VSTUP)),
                        new StreamResult(new File(HTML_VYSTUP)));
  }
}
```

- vytvoří soubor `jaxp-saxon.html` s obsahem:

```
<?xml version="1.0" encoding="UTF-8"?><html>
   <body>transformace nad konkrétním XML souborem</body>
</html>
```

9.7.3. Přímé použití Saxon

- využití JAXP pomocí třídy `Transformer` je sice standardní, ale přicházíme např. o možnost spouštět transformaci pomocí pojmenované šablony
- pokud rezignujeme na využití JAXP, je možné přímo použít Saxon tak, jak jsme byli zvyklí z příkazové řádky
  - pomocí přepínače `-it` lze vyvolat pojmenovanou šablonu
  - lze nastavit i globální parametry
- Java soubor `XsltSaxon.java`
  - celé zapojení Saxonu je umožněno pomocí:

    ```
    import net.sf.saxon.Transform;
    ```

    - tato třída má metodu `main(String[] args)`, jejíž nastavení parametrů již známe ze spouštění z příkazové řádky
    - stačí pouze umístit jednotlivé parametry do požadovaného pole `String` pomocí triku:

      ```
      Transform.main(new String[] { ... }
      ```

```
import net.sf.saxon.Transform;

/**
 * Ukazka XSLT transformace primo procesorem Saxon
 */
public class XsltSaxon {

  static final String ADRESAR = "d:/xslt/12/";
  static final String XSLT_STYL = ADRESAR + "transformace-z-Javy-xhtml.xsl";
  static final String XML_VSTUP = ADRESAR + "jidlo-2-ovoce.xml";
  static final String HTML_VYSTUP = ADRESAR + "java-saxon.html";
  static final String HTML_VYSTUP_SABLONA_DEF =
                                 ADRESAR + "java-saxon-sablona-def.html";
  static final String HTML_VYSTUP_SABLONA =
                                  ADRESAR + "java-saxon-sablona.html";

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

    // volani pro XML soubor
    Transform.main(new String[] {
        "-o", HTML_VYSTUP,
        XML_VSTUP,
```

```
        XSLT_STYL
      });
```

- odpovídá volání z příkazové řádky:

```
saxon -o java-saxon.html jidlo-2-ovoce.xml transformace-z-Javy-xhtml.xsl
```

- vytvoří soubor `jaxp-saxon.html` s obsahem:

```
<?xml version="1.0" encoding="UTF-8"?><html>
   <body>transformace nad konkrétním XML souborem</body>
</html>
```

- další volání v Java souboru `XsltSaxon.java`

```
    // volani pro pojmenovanou sablonu - defaultni nastaveni
    Transform.main(new String[] {
        "-o", HTML_VYSTUP_SABLONA_DEF,
        "-it", "pojmenovanaSablona",
        XSLT_STYL
      });
```

- odpovídá volání z příkazové řádky:

```
saxon -o java-saxon-sablona-def.html -it pojmenovanaSablona ►
transformace-z-Javy-xhtml.xsl
```

- vytvoří soubor `java-saxon-sablona-def.html` s obsahem:

```
<?xml version="1.0" encoding="UTF-8"?><html>
   <body>transformace pomocí pojmenované šablony se vstupním parametrem: ►
defaulní nastavení</body>
</html>
```

- další volání v Java souboru `XsltSaxon.java`

```
    // volani pro pojmenovanou sablonu
    Transform.main(new String[] {
        "-o", HTML_VYSTUP_SABLONA,
        "-it", "pojmenovanaSablona",
        XSLT_STYL,
        "vstupniParametr=můj"
      });
```

- odpovídá volání z příkazové řádky:

```
saxon -o java-saxon-sablona.html -it pojmenovanaSablona ►
transformace-z-Javy-xhtml.xsl "vstupniParametr=můj"
```

- vytvoří soubor `java-saxon-sablona.html` s obsahem:

```
<?xml version="1.0" encoding="UTF-8"?><html>
   <body>transformace pomocí pojmenované šablony se vstupním parametrem: ►
můj</body>
</html>
```

Kapitola 10. SAX, StAX

10.1. SAX – *Simple API for XML*

10.1.1. Úvodní informace

- proudové zpracování XML dokumentu typu *push parser*
- SAX je tvořeno několika rozhraními v Java Core API popsanými 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

10.1.2. 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

10.1.2.1. 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`

10.1.2.2. 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 (ParserCongiruration Exception e) {
      výjimky vzniklé při nastavování vlastností parseru – viz dále
  }
  catch (IOException e) {
     výjimky vzniklé při čtení XML dokumentu ze souboru nebo z proudu
  }
```

10.1.2.3. 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 {
  // zformatovani textu hlaseni
  private String textHlaseni(SAXParseException e) {
    return e.getSystemId() + "\n"
           + "radka: " + e.getLineNumber()
           + " sloupec: " + e.getColumnNumber()
           + "\n" + e.getMessage();
  }

  // obsluha varovnych hlaseni
  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 fatalnich chyb
  public void fatalError(SAXParseException e) throws SAXException {
    throw new SAXException("Fatalni chyba: " + textHlaseni(e));
  }
}
```

10.1.3. 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

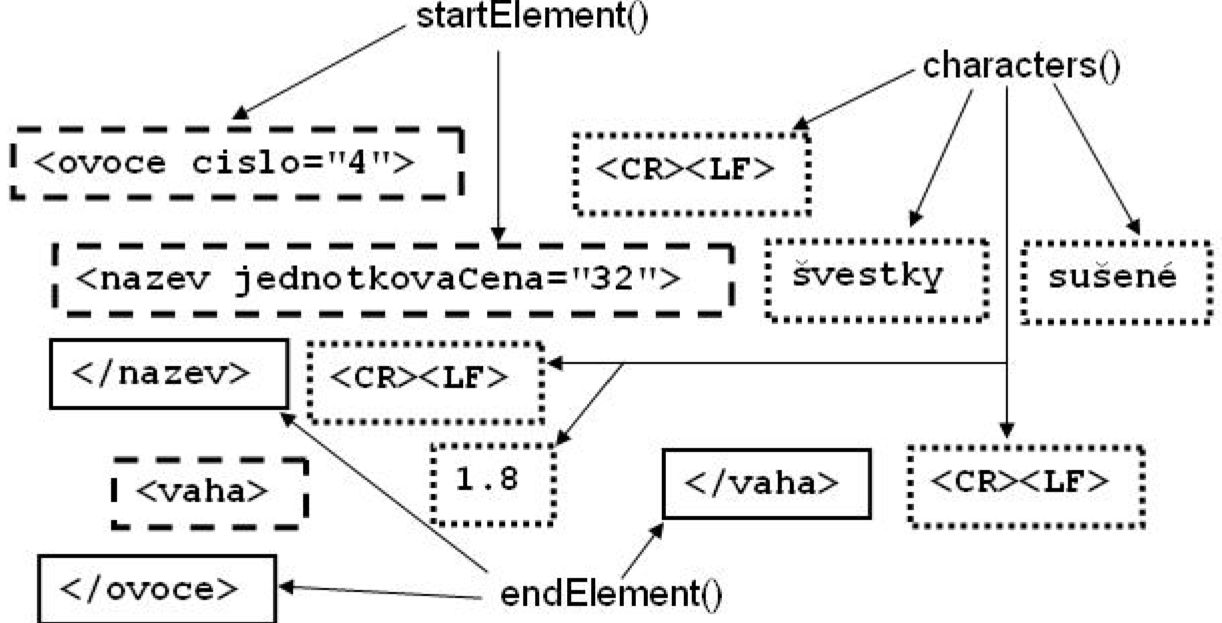
  (oba viz v předchozím příkladě)

- `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 zpracováních XML.

10.1.3.1. 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

10.1.3.2. 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);
    }
```

```
  }
}
```

10.1.3.3. 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));
```

10.1.4. Zpracování složitějšího XML dokumentu

- pokud se v jednotlivých elementech neopakují na různých místech stejně pojmenované vnořené podelementy, 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

10.1.5. 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

10.1.6. 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/namespaces", 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 schematu 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"));
    ```

10.1.7. 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/namespaces", true);
//      spf.setNamespaceAware(true);      // stejna 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("Fatalni 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\jaxp\samples\sax\SAXLocalNameCount.java`

10.1.8. 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

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* – nízkoúrovňové, 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 vracen 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 nevšímá (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 FileInputStream(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 FileInputStream(SOUBOR));
```

  - soubor skutečně čteme pomocí `FileInputStream()`
    - umožní, aby parser použil kódování uvedené v hlavičce XML dokumentu
  - pozor na skutečnost, že při nevhodném použití `FileReader(SOUBOR)` by byl XML soubor čten v nativním kódování platformy!
    - to znamená, že na informaci o použitém kódování uvedenou na začátku XML souboru by se *nebral zřetel*!
- 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 FileInputStream(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. Práce se jmennými prostory

■ obsahuje-li XML soubor jmenné prostory je třeba věnovat práci s elementy větší pozornost

■ nejjednodušší situace je, když celý XML dokument používá pouze jeden jmenný prostor

- pak není třeba nijak měnit již dříve uvedené postupy, protože metoda `getLocalName()` vrací stále jméno elementu

■ stejně jednoduchá je i situace, kdy je použito více jmenných prostorů, ale jména značek se neopakují

■ praktické použití jmenných prostorů je ale až v případě, že se jména značek opakují

- pak je třeba rozlišovat stejně pojmenované značky podle prefixu nebo podle URI
- kromě nich lze využít i třídu `QName`, která uchovává URI a lokální jméno

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

  <potrava:ovoce cislo="1">
    <potrava:nazev potrava:jednotkovaCena="10">jablka</potrava:nazev>
    <miry:vaha>
      <miry:hodnota>2.5</miry:hodnota>
      <miry:nazev>kg</miry:nazev>
    </miry:vaha>
  </potrava:ovoce>
  <potrava:ovoce cislo="2">
    <potrava:nazev jednotkovaCena="25">banány</potrava:nazev>
```

```
    <miry:vaha>
      <miry:hodnota>2</miry:hodnota>
      <miry:nazev>kg</miry:nazev>
    </miry:vaha>
  </potrava:ovoce>
</potrava:jidlo>
```

Pozor na skutečnost, že prefix a URI se od sebe významně liší, např. pro element `potrava:nazev` je:

```
URI:    http://www.kiv.zcu.cz/~herout/xml/jidlo-sada
QName: {http://www.kiv.zcu.cz/~herout/xml/jidlo-sada}nazev
Prefix: potrava
```

Program, který vypíše informace o jednotlivých ovocích.

- element `nazev` se v XML souboru opakuje, proto je nutné jej rozlišovat
- atribut `jednotkovaCena` nemusí ale může mít uveden jmenný prostor
  - program proto používá zdlouhavý, ale bezpečný způsob čtení tohoto atributu pomocí indexu
  - není-li totiž jmenný prostor uveden, má URI (získaný metodou `getAttributeNamespace()`) hodnotu `null`, ale prefix (získaný metodou `getAttributePrefix()`) je prázdný řetězec `""`
  - nelze tedy jednoduše použít metodu `getAttributeValue(String namespaceURI, String localName)`

```
public class StAXJmenneProstory {
  public static final String SOUBOR = "jidlo-ns.xml";

  public static void main(String[] args) {
    try {
      XMLInputFactory f = XMLInputFactory.newInstance();
     XMLStreamReader r = f.createXMLStreamReader(new FileInputStream(SOUBOR));
      vypisPrehled(r);
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

  static void vypisPrehled(XMLStreamReader r) throws Exception {
    while (r.hasNext() == true) {
      r.next();
      if (r.isStartElement() == true) {
        QName qn = r.getName();
        String prefix = r.getPrefix();
        String localName = qn.getLocalPart();
        if (localName.equals("nazev") == true
            &&  prefix.equals("potrava")) {
          int pocetAtr = r.getAttributeCount();
          int indexAtr;
          for (indexAtr = 0; indexAtr < pocetAtr; indexAtr++) {
            if (r.getAttributeLocalName(indexAtr).equals("jednotkovaCena")) {
```

```
              break;
            }
          }
          String cena = r.getAttributeValue(indexAtr);
          System.out.print(r.getElementText() + " za " + cena + ",-Kc");
          continue;
        }
        else if (r.getLocalName().equals("hodnota") == true) {
          String vaha = r.getElementText();
          System.out.print(", mnozstvi: " + vaha);
          continue;
        }
        else if (localName.equals("nazev") == true
                 &&  prefix.equals("miry")) {
          System.out.println(" [" + r.getElementText() + "]");
        }
      }
    }
  }
}
```

vypíše:

```
jablka za 10,-Kc, mnozstvi: 2.5 [kg]
banány za 25,-Kc, mnozstvi: 2 [kg]
```

10.2.6. Zápis do XML dokumentu

- generování dokumentu je „přímočaré“
  - automaticky převádí `&` a `<` na `&` a `<`
  - lze nastavit použité výstupní kódování
  - odřádkování a odsazování elementů možné provést „ručně“
    - není to však nutné – pro tyto účely lze použít známou třídu `Transformer` (viz též dále)
  - další použitelné 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 final String KODOVANI = "UTF-8";

  public static void main(String[] args) {
    try {
      Random r = new Random();
      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 + " & <";
        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="32">ovoce 1 &amp; &lt;</nazev>
    <vaha>3.4</vaha>
  </ovoce>
  <ovoce cislo="2">
    <nazev jednotkovaCena="33">ovoce 2 &amp; &lt;</nazev>
    <vaha>5.5</vaha>
  </ovoce>
  <ovoce cislo="3">
    <nazev jednotkovaCena="39">ovoce 3 &amp; &lt;</nazev>
    <vaha>2.6</vaha>
  </ovoce>
</jidlo>
```

10.2.7. Validace a transformace dokumentu

■ na předchozím příkladu je nevhodné „ruční“ odřádkování a odsazování

■ při požadavku na validaci vzniklého souboru by jej bylo nutné číst znovu z disku

■ řešením je vytvářet XML soubor jako proud bajtů v paměti

- nevýhodou je, že ztrácíme paměťovou nezávislost na velikosti zapisovaného souboru (už nezapisujeme do streamovaného souboru)

```
public class JidloStAXWriteValidator {
  public static final int POCET = 2;
  public static final String KODOVANI = "UTF-8";

  public static byte[] zapisDoXMLStreamuVPameti() {
    byte[] byteXMLStream = null;
    try {
      Random r = new Random();

      XMLOutputFactory fo = XMLOutputFactory.newInstance();
      CharArrayWriter chaw = new CharArrayWriter();
      XMLStreamWriter w = fo.createXMLStreamWriter(chaw);

      w.writeStartDocument(KODOVANI, "1.0");
      w.writeComment(" přehled šmakovních dobrůtek ");
      w.writeStartElement("jidlo");
      for (int i = 1;  i <= POCET;  i++) {
        w.writeStartElement("ovoce");
        w.writeAttribute("cislo", "" + i);
        w.writeStartElement("nazev");
```

```
        int cena = r.nextInt(40) + 10;
        w.writeAttribute("jednotkovaCena", "" + cena);
        String nazev = "nějaké ovoce " + i + " & <";
        w.writeCharacters(nazev);
        w.writeEndElement();
        w.writeStartElement("vaha");
        double vaha = r.nextDouble() * 10.0;
        String oriznute = String.valueOf(vaha).substring(0, 3);
        w.writeCharacters(oriznute);
        w.writeEndElement();
        w.writeEndElement();
      }
      w.writeEndElement();
      w.writeEndDocument();
      w.close();
      byteXMLStream = chaw.toString().getBytes(KODOVANI);
    } catch (Exception e) {
      e.printStackTrace();
    }

    return byteXMLStream;
  }
```

■ tento proud bajtů (pole bajtů) se použije jak pro validaci, tak i pro zápis do souboru

```
  public static final String SCHEMA = "jidlo.xsd";

  public static void validace(byte[] byteXMLStream) {
    try {
      SchemaFactory sf = SchemaFactory.newInstance(
                           XMLConstants.W3C_XML_SCHEMA_NS_URI);
      Schema sch = sf.newSchema(new File(SCHEMA));
      Validator val = sch.newValidator();
      val.setErrorHandler(new ValidaceChybyZjisteneParserem());
     val.validate(new StreamSource(new ByteArrayInputStream(byteXMLStream)));
      System.out.println("Validace probehla uspesne");
    }
    catch (SAXParseException e) {
      System.out.println("Chyba validace:");
    } catch (SAXException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
```

■ pro zápis se využije třída `Transformer`, která mj. umí zajistit i požadované odřádkování a odsazení

■ vedlejším efektem bude i úplná hlavička výsledného XML souboru (včetně `standalone="yes"`)

```
  public static final String SOUBOR = "jidlo-gener-valid.xml";

  public static void zapisDoSouboru(byte[] byteXMLStream) {
    try {
```

```
      TransformerFactory tf = TransformerFactory.newInstance();
      Transformer zapisovac = tf.newTransformer();
      zapisovac.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "no");
      zapisovac.setOutputProperty(OutputKeys.ENCODING, KODOVANI);
      zapisovac.setOutputProperty(OutputKeys.STANDALONE, "yes");
      zapisovac.setOutputProperty(OutputKeys.INDENT, "yes");
     zapisovac.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", ►
"2");

      zapisovac.transform(new StreamSource(
                            new ByteArrayInputStream(byteXMLStream)),
                          new StreamResult(
                            new File(SOUBOR)));
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
```

vytvoří soubor:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- přehled šmakovních dobrůtek -->
<jidlo>
  <ovoce cislo="1">
    <nazev jednotkovaCena="22">nějaké ovoce 1 &amp; &lt;</nazev>
    <vaha>2.7</vaha>
  </ovoce>
  <ovoce cislo="2">
    <nazev jednotkovaCena="30">nějaké ovoce 2 &amp; &lt;</nazev>
    <vaha>6.8</vaha>
  </ovoce>
</jidlo>
```

Kapitola 11. DOM – *Document Object Model*

11.1. 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 zajišťují 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

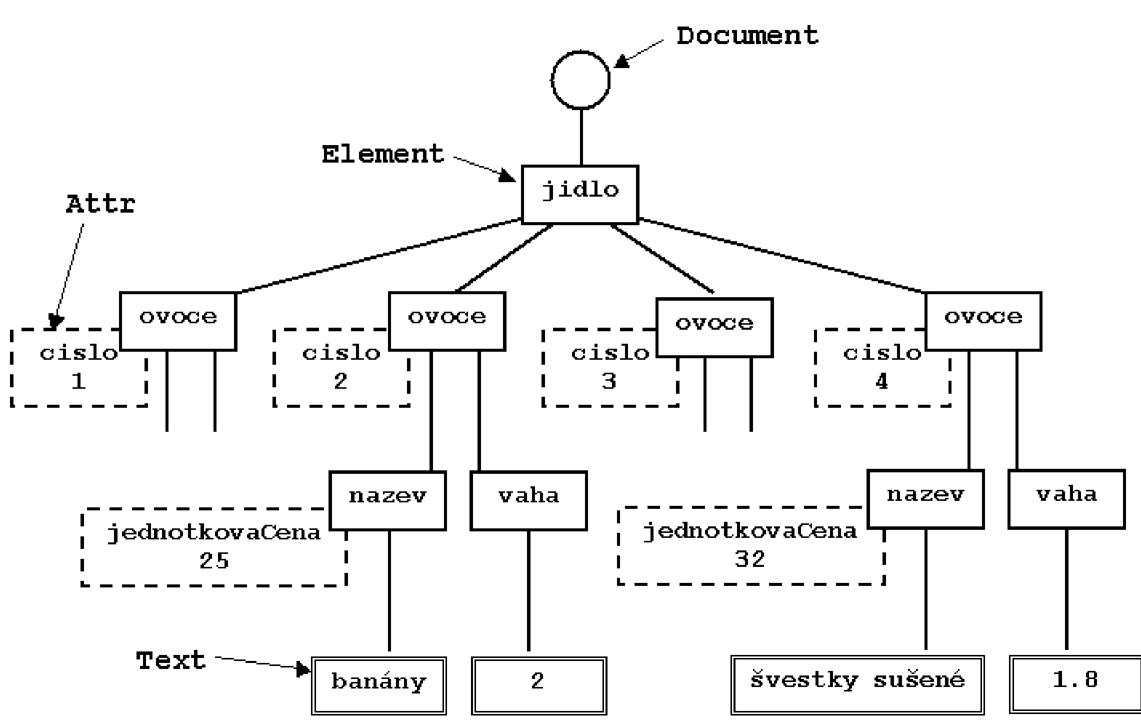
Příklad 11.1. pro jidlo.xml

```
<?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>

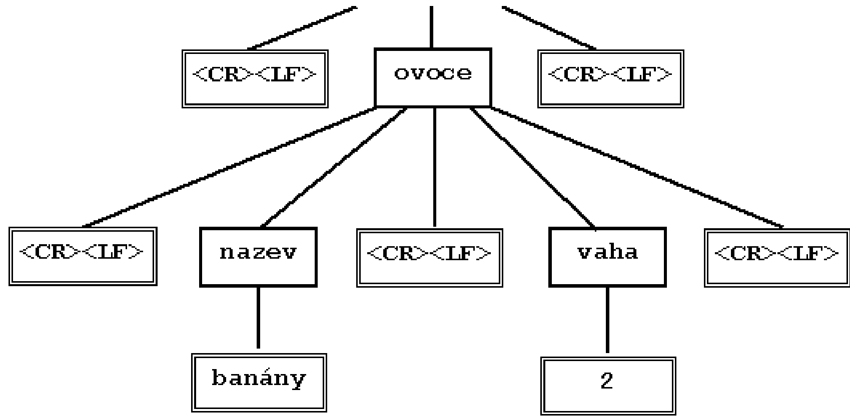
  <ovoce cislo="3">
    <nazev jednotkovaCena="19">grapefruity</nazev>
    <vaha>0.75</vaha>
  </ovoce>

  <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

11.2. 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.*;           // kvuli 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

11.3. 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.

11.3.1. 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;
}
```

11.4. 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`

11.4.1. 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 komenáře je jméno nodu „`#comment`“ a pro text je „`#text`“

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

  Výstraha

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

11.4.2. Metoda pro pohyb nahoru (na rodiče)

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

11.4.3. Metody pro horizontální pohyb (na sourozence)

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

11.4.4. 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

11.4.5. Metody pro práci s atributy

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

`NamedNodeMap` má metody:

- `int getLength()` – počet atributů
- `Node getNamedItem(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.getNamedItem("jednotkovaCena").getNodeValue();
```

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

11.5. 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.getNamedItem("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;
  }
```

11.6. Problém vkládaných elementů (odřádková-ní)

- 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:
nazev
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 11.2. Funkční program s automatickým odstraňováním „odřádkovávacích“ 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 „odřádkovávací“ nody – viz dále

11.7. Práce se jmennými prostory

11.7.1. Jmenné prostory fakticky neuvažujeme

- pokud chceme XML dokument pouze jednoduše zpracovat, je práce se jmennými prostory zcela stejná, jako práce s elementy či atributy bez jmenných prostorů
  - pouze se uvádí jméno atributu či elementu včetně jmenného prostoru a oddělovací dvojtečky
    - pragmaticky můžeme říci, že název elementu či atributu se prostě rozšíří o identifikátor jmenného prostoru
- pokud by byl použit jmenný prostor `potrava`, pak by byl
  - element `potrava:nazev` zpracováván např. jako:

    ```
    NodeList nl = doc.getElementsByTagName("potrava:nazev");
    ```

  - atribut `potrava:jednotkovaCena` zpracováván např. jako:

    ```
    String hodnota = nm.getNamedItem("potrava:jednotkovaCena").getNodeValue();
    ```

- například XML dokument

  ```
  <?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>
    <potrava:ovoce cislo="2">
      <potrava:nazev jednotkovaCena="25">banány</potrava:nazev>
      <potrava:vaha>2</potrava:vaha>
    </potrava:ovoce>
  ...
  ```

  zpracujeme programem:

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

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

        Document doc = builder.parse(SOUBOR);
        vypisNazvyACeny(doc);
      }
      catch (Exception e) {
  ```

```
      e.printStackTrace();
    }
  }

  private static void vypisNazvyACeny(Document doc) {
    NodeList nl = doc.getElementsByTagName("potrava:nazev");
    for (int i = 0;  i < nl.getLength();  i++) {
      Node eNazev = nl.item(i);               // Element
      Node tNazev = eNazev.getFirstChild();   // Text
      String nazev = tNazev.getNodeValue().trim();
      NamedNodeMap nnm = eNazev.getAttributes();
      Node eJednotkovaCena;
      eJednotkovaCena = nnm.getNamedItem("potrava:jednotkovaCena");
      if (eJednotkovaCena == null) {
        eJednotkovaCena = nnm.getNamedItem("jednotkovaCena");
      }
      String cena = eJednotkovaCena.getNodeValue();
      System.out.println(nazev + ": " + cena);
    }
  }
}
```

a dostaneme výsledek:

```
jablka: 10
banány: 25
grapefruity: 19
švestky sušené: 32
```

v programu je vidět, jak je třeba ošetřit případ atributů, které mohou nebo nemusejí mít uveden jmenný prostor (srovnej `potrava:jednotkovaCena="10"` u prvního ovoce a `jednotkovaCena="25"` u druhého ovoce)

11.7.2. Se jmennými prostory pracujeme

pokud chceme se jmennými prostory pracovat, je nutné dávat pozor na několik skutečností

- po vytvoření `DocumentBuilderFactory` je třeba zapnout práci se jmennými prostory (implicitně je vypnutá)

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

  - bez tohoto nastavení nejsou jmenné prostory uvažovány!
  - například zpracování předchozího XML dokumentu programem

```
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    dbf.setNamespaceAware(false);
    DocumentBuilder builder = dbf.newDocumentBuilder();
    Document doc = builder.parse(SOUBOR);
    vypisJmena(doc);
  ...
    private static void vypisJmena(Document doc) {
```

```
    NodeList nl = doc.getChildNodes();
    for (int i = 0;  i < nl.getLength();  i++) {
      Node eNazev = nl.item(i);
      System.out.println("LocalName:    " + eNazev.getLocalName());
      System.out.println("NodeName:     " + eNazev.getNodeName());
      System.out.println("NamespaceURI: " + eNazev.getNamespaceURI());
    }
  }
```

poskytne výsledek:

```
LocalName:    null
NodeName:     potrava:jidlo
NamespaceURI: null
```

zatímco po nastavení:

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

dosáváme již použitelný (nikoliv však očekávaný) výsledek:

```
LocalName:    jidlo
NodeName:     potrava:jidlo
NamespaceURI: http://www.kiv.zcu.cz/~herout/xml/jidlo-sada
```

všimněte si, že:

- ♦ hodnota `NodeName` zůstává v obou případech nezměněna
  - – to je také důvod, proč funguje způsob jednoduchého čtení - viz předchozí část
- ♦ hodnota `NamespaceURI` není `potrava`, jak by se očekávalo, ale hodnota ze samého začátku XML dokumentu
  - – záměna těchto dvou hodnot je velmi častá chyba!

- ■ chceme-li vypsat hodnoty elementů a atributů jako v dřívějším případě, zjistíme NSURI, které pak použijeme:

```
  private static void vypisNazvyACeny(Document doc) {
    // zcela chybne - pouziva "potrava" misto URI
    // NodeList nl = doc.getElementsByTagNameNS("potrava", "nazev");

    String nsURI = doc.getFirstChild().getNamespaceURI();
    NodeList nl = doc.getElementsByTagNameNS(nsURI, "nazev");

    //    odtud stejné jako v předchozím případě
    for (int i = 0;  i < nl.getLength();  i++) {
      Node eNazev = nl.item(i);               // Element
      Node tNazev = eNazev.getFirstChild();   // Text
      String nazev = tNazev.getNodeValue().trim();
      NamedNodeMap nnm = eNazev.getAttributes();
      Node eJednotkovaCena;
```

```
      eJednotkovaCena = nnm.getNamedItem("potrava:jednotkovaCena");
      if (eJednotkovaCena == null) {
        eJednotkovaCena = nnm.getNamedItem("jednotkovaCena");
      }
      String cena = eJednotkovaCena.getNodeValue();
      System.out.println(nazev + ": " + cena);
    }
  }
```

program vypíše:

```
jablka: 10
banány: 25
grapefruity: 19
švestky sušené: 32
```

- trik je, že místo NSURI lze použít univerzální řetězec `"*"`:

```
  private static void vypisNazvyACeny(Document doc) {
    // String nsURI = doc.getFirstChild().getNamespaceURI();
    // NodeList nl = doc.getElementsByTagNameNS(nsURI, "nazev");
    NodeList nl = doc.getElementsByTagNameNS("*", "nazev");
...
```

11.8. Automatické odstranění komentářů

Výstraha

Funguje jen pro JDK 1.6, respektive pro JAXP 1.4.

- velice podobné výše uvedenému způsobu odstranění „odřádkovávacích“ 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">
    <nazev jednotkovaCena="10">jablka</nazev>
    <!-- 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.

11.9. 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;
  }
}
```

11.10. 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 vracet 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 11.3.

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 vsech 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 „odřádkovávací“ 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 „profiltrované“ příslušným `NodeFilter`, takže obsahují jen to, co nás zajímá, např. elementy

11.11. Snadné odstranění „odřádkovávacích“ 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 „odřádkovávací“ nod.

11.12. 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 (*eXtensible 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

11.12.1. 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
  ```

11.12.2. Ukázka zápisu do souboru

Program přečte soubor `jidlo.xml` a uloží jej do souboru `jidlo-utf-8.xml`, tj. změní 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 změní 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.OutputKeys.STANDALONE, "yes"`

a musí se proto použít trik, ve kterém se nastaví `standalone` pomocí metody `setXmlStandalone(true);`

bez tohoto komplikovaného postupu dvojí transformace dostaneme výstupní XML soubor v požadovaném kódování, ale v hlavičce bude uvedeno kódování původního vstupního XML dokumentu, což je samozřejmě hrubá chyba

11.12.3. 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 nepříjemná 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();
    }
  }
}
```

11.13. 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 `<`

11.13.1. 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)`

11.13.2. 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)`

11.13.3. 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 `
`

11.13.4. 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>
```

11.14. 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

11.14.1. 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>
```

11.15. 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 12. Úvod do SVG

Poznámka

Účelem je podat přehled základních možností SVG formátu tak, aby bylo možné bez předchozích znalostí počítačové grafiky vytvořit obrázek pomocí jednoduché vektorové grafiky.

12.1. Úvodní informace

- SVG = *Scalable Vector Graphics*
- vektorový formát pro popis 2D grafiky vhodný pro použití na WWW
  - vyzrálý, obecně přijímaný grafický formát, který se již nebude významně měnit
    - podporován četnými grafickými editory minimálně na úrovni exportu do SVG
  - lze srovnávat s technologií Flash, oproti které má četné výhody
    - podrobnosti v `interval.cz/clanky/pruvodce-svg-svg-versus-flash/`
  - spolupracuje s kaskádními styly (CSS)
- zápis kompletně v XML
  - lze validovat
  - lze jednoduše programově měnit jednotlivé prvky přes DOM model
  - umožňuje zavést interaktivitu – reakce na události nad prvkem obrázku (kliknutí apod.)
  - dává schopnosti vytvářet vektorovou grafiku
    - minimálními prostředky (textový editor) s maximální možnou jednoduchostí a čitelností
    - libovolně sofistikovanými metodami s využitím všech XML technologií – zejména XSLT
- `http://www.w3.org/TR/SVG/` – technická specifikace
  - `http://www.w3.org/Graphics/SVG/` – popis
- dobré zdroje informací a tutoriály
  - články Martina Hejrala na `interval.cz`
  - články Pavla Tišnovského na `www.root.cz`
- vznik
  - od 1998 vývoj
  - od 2001 verze 1.0
  - od 2003 verze 1.1 – vznik subverzí, vylepšená práce s textem

- od 2005 verze 1.2 – další vylepšení práce s textem

- existující subverze (plná verze se označuje jako SVG nebo někdy *SVG Full*)
  - celá specifikace SVG Full je komplikovaná pro implementaci, proto od verze 1.1 existují dvě podmnožiny SVG (někdy též „profily“)
    - SVG Basic (SVGB) – pro PDA
      - omezeny filtry a použití ořezových cest
    - SVG Tiny (SVGT) – pro mobilní telefony
      - vypouští skripty, filtry, přechody barev, vzorky, průhlednost a CSS formátování
      - od 2008 verze Tiny 1.2
    - ani v jednom profilu nejsou omezeny animace!
- MIME typ je `image/svg+xml`
- přípona souboru `.svg`, při komprimování pomocí GZIP `.svgz`
- podporující HTML prohlížeče:
  - Opera – tradičně nejlepší podpora, od verze 9 i SVG Full
  - Firefox – částečná podpora
  - MSIE – do verze 8.0 vůbec nepodporuje
- běžně využívaný prohlížeč Adobe SVG Viewer (ASV 3)

  `http://www.adobe.com/svg/viewer/install/`

12.1.1. Struktura SVG dokumentu

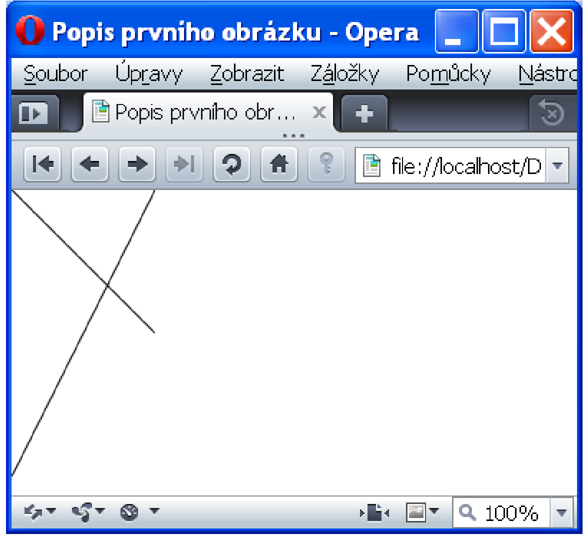
- pokud je SVG dokument samostatný (není vkládán), pak může být následující

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg
  xmlns="http://www.w3.org/2000/svg"
  width="100" height="200px"
  viewBox="0 0 100% 100%">

  <title>Popis prvního obrázku</title>
  <line stroke="black" x1="0" y1="0" x2="100%" y2="100"/>
  <line stroke="black" x1="0" y1="100%" x2="100%" y2="0"/>
</svg>
```

- `standalone="no"` – důležité – množství atributů SVG formátu je přednastavených v DTD, pokud je připojen

- `<!DOCTYPE svg PUBLIC` ... – vhodné pro doplnění přednastavených atributů a pro validaci
  - není nezbytné pro zobrazování
- kořenový element je `<svg>`
  - je nazývaný „fragment SVG“
  - může být samostatný nebo zanořen do jiného XML dokumentu nebo zanořen i sám do sebe (`<svg>` v `<svg>`)
  - má množství atributů
    - `xmlns="http://www.w3.org/2000/svg"` – implicitní jmenný prostor
      - nutné uvést, aby byl obrázek brán jako obrázek, nikoliv jako běžný XML dokument
      - bez tohoto atributu se obrázek nezobrazí
    - `width="100" height="200px"` – šířka a výška kreslicího plátna
      - lze používat běžně známé jednotky `px`, `mm`, `cm`, ...
      - bezrozměrné údaje jsou v pixelech
    - `viewBox="0 0 100% 100%"` – význam "počátekX počátekY šířka výška"
      - pohled na kreslicí plochu
      - počátek souřadnic je vlevo nahoře
      - množství způsobů použití, zde nastaven na režim „je vidět celé kreslicí plátno nedeformovaně“
      - nepoužívat, když je prohlížeč Firefox
- element `<title>` – popis obrázku
- `<line stroke="black" x1="0" y1="0" x2="100%" y2="100"/>`
  - vykreslení čáry implicitní šířky 1 px černou barvou z počátečních souřadnic [0,0] do
    - koncové souřadnice X (`100%` = max. X rozměr kreslícího plátna)
    - koncové souřadnice Y (`100px` = absolutní souřadnice)
- zobrazí se



- běžný začátek SVG dokumentu je:

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

<svg
  xmlns="http://www.w3.org/2000/svg"
  width="100" height="200">
```

12.2. Základní elementy SVG

- budou popisovány jen nejzákladnější elementy a atributy s ohledem na možnosti jednoduché vektorové kresby
- SVG podporuje tři typy objektů:
  - základní geometrické tvary (*basic shapes*)
  - cesty (*path*)
  - texty

12.2.1. Základní geometrické tvary (*basic shapes*)

12.2.1.1. Společné vlastnosti zadávané v atributech

- `stroke` – barva, která se zadává jako
  - pojmenovaná barva: `stroke="deeppink"` – k dispozici asi 100 barev, vzorník na:

    `http://www.w3.org/TR/SVG11/types.html#ColorKeywords`
  - barevné složky RGB: `stroke="rgb(255, 20, 147)"`
  - hexa RGB: `stroke="#FF1493"`
- `fill` – barva výplně – zadává se jako u `stroke`

- `fill="none"` – bez výplně

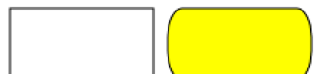
- `stroke-width` – šířka čáry či obrysu, zadává se
  - bezrozměrně – `stroke-width="1"` – v pixelech
  - s jednotkou – `stroke-width="1mm"`

12.2.1.2. Úsečka

- `<line x1="0" y1="0" x2="100%" y2="50%"/>`
  - [`x1`, `y1`] – souřadnice počátečního bodu
  - [`x2`, `y2`] – souřadnice koncového bodu

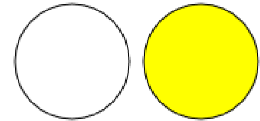
12.2.1.3. Obdélník

- vykresluje se osově orientovaný obdélník
  - otočení možné pomocí transformace – viz dále
- povinné atributy jsou souřadnice levého horního rohu [`x`, `y`], šířka `width` a výška `height`
- `<rect fill="none" x="10" y="10" width="100" height="50"/>`
  - obdélník bez výplně
- `<rect fill="yellow" x="120" y="10" width="100" height="50" rx="10" ry="20"/>`
  - obdélník se žlutou výplní s nestejně zakulacenými rohy



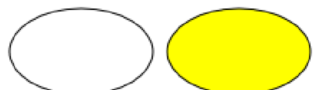
12.2.1.4. Kružnice

- povinné atributy jsou souřadnice středu [`cx`, `cy`] a poloměr `r`
```
<circle fill="none" cx="50" cy="50" r="40"/>
<circle fill="yellow" cx="140" cy="50" r="40"/>
```



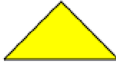
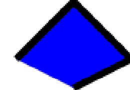
12.2.1.5. Elipsa

- povinné atributy jsou souřadnice středu [`cx`, `cy`] a poloměry os `rx` a `ry`
```
<ellipse fill="none" cx="60" cy="50" rx="50" ry="30"/>
<ellipse fill="yellow" cx="170" cy="50" rx="50" ry="30"/>
```



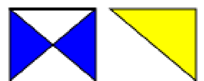
12.2.1.6. Lomená čára (*polyline*)

- sestává z libovolného počtu na sebe navazujících úseček, které jsou zapsány v povinném atributu `points`
  - v něm jsou dvojice souřadnic x,y
  - typicky se od sebe oddělují čárkou a dvojice navzájem mezerou, ale není to nezbytné
- pokud požadujeme vyplnění neuzavřeného objektu, vyplní se do koncového bodu
  - toto ale není dobrý způsob použití výplní – lepší viz dále *Polygon*
-
```
<polyline fill="none" points="10,50 50,10 90,50 10,50"/>
<polyline fill="yellow" points="110,50 150,10 190,50 110,50"/>
<polyline fill="blue" stroke-width="5"
          points="210,50 250,10 290,50 250,70"/>
```

12.2.1.7. Uzavřený polygon

- zapisuje se jako `<polyline>` pomocí `points`
  - není nutno uvádět návrat z předposledního bodu do bodu počátečního
  - pokud se čáry polygonu kříží, používá se pro určení vnějších a vnitřních prostorů speciální algoritmus
-
```
<polygon fill="blue" points="10,10 70,60 70,10 10,60 10,10 70,10"/>
<polygon fill="yellow" points="80,10 140,10 140,60"/>
```



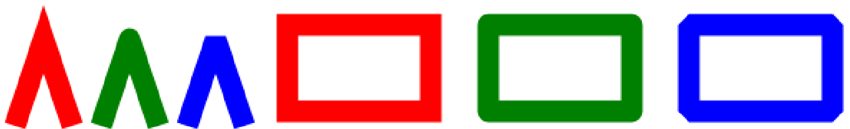
12.2.2. Napojování a ukončování čar

- čáry vykreslené některými předchozími způsoby mohou mít různá napojení a ukončení
  - tytéž možnosti platí pro dále uvedené cesty (*path*)

12.2.2.1. Styl napojení čar

- pokud jsou v kresleném tvaru ostře lomené čáry (např. i rohy obdélníka), lze pomocí `stroke-linejoin` specifikovat způsob zakončení rohu
  - `miter` – spojení protažené do špičky (ostrý roh)
  - `round` – spojení ukončené zakulacením úměrným šířce čáry
  - `bevel` – spojení ukončené seseknutím
- v příkladu je šířka všech čar `stroke-width="15"`

```
<polyline stroke="red" stroke-linejoin="miter"
                              points="10,90 30,30 50,90" />
<polyline stroke="green" stroke-linejoin="round"
                              points="70,90 90,30 110,90" />
<polyline stroke="blue" stroke-linejoin="bevel"
                              points="130,90 150,30 170,90" />
<rect stroke="red" stroke-linejoin="miter"
                          x="200" y="20" width="100" height="60"/>
<rect stroke="green" stroke-linejoin="round"
                          x="340" y="20" width="100" height="60"/>
<rect stroke="blue" stroke-linejoin="bevel"
                          x="480" y="20" width="100" height="60"/>
```



12.2.2.2. Styl ukončení čar

- koncové body čar mohou mít různá zakončení specifikovaná pomocí `stroke-linecap`
  - `butt` – kolmý řez v místě koncového bodu
  - `round` – zakulacení, které prodlouží délku čáry
  - `square` – kolmý řez prodloužený o 1/2 šířky cesty (čára je opět delší)
- v příkladu je šířka všech čar `stroke-width="15"`, černá čára šířky 1px uvnitř je pro ukázku skutečné délky čáry

```
<polyline stroke="red" stroke-linecap="butt" points="10,90 30,10 50,90" />
<polyline stroke="green" stroke-linecap="round" points="70,90 90,10 110,90" />
<polyline stroke="blue" stroke-linecap="square" points="130,90 150,10 170,90" />
```

12.2.3. Cesty (*path*)

- univerzální element `<path>` pro tvorbu obrázků, pokud nestačí základní geometrické tvary
  - pro výplně, tloušťky čar, jejich barvy, napojení a ukončení platí stejné způsoby nastavení jako pro základní geometrické tvary
- při kreslení se používá analogie s perem plotteru (*concept of a current point*)
  - pero se pohybuje z předchozího bodu podle charakteru zadaného příkazu (např. úsečka, kružnice, apod.) a pokud je dole, tak se při pohybu kreslí
  - pokud je pero nahoře (jen u příkazu *moveto*), pouze se přesouvá, tj. stanoví se další bod
- příkazy pro pohyb pera se zadávají do atributu `d` (`<path d=""/>`)
  - v jednom atributu `d` se udávají najednou všechny příkazy popisující danou cestu
- příkaz využívá ABSOLUTNÍCH souřadnic, pokud je jeho zkratka psána VELKÝM písmenem a relativních souřadnic, je-li malým písmenem
  - souřadnice (a případné další parametry) jsou uvedeny hned za zkratkou příkazu
    - z důvodů lepší čitelnosti se opět dvojice X a Y souřadnic oddělují čárkou

12.2.3.1. Základní příkazy pro vytváření cesty

- *moveto*: `M` nebo `m` – přesouvá pero (jako jediný příkaz nekreslí – pero je nahoře)
  - `M100,50` – přesune pero na absolutní souřadnice [100, 50]
  - `m100,-50` – přesune pero o 100 doprava a o 50 nahoru
- *closepath*: `Z` nebo `z` – uzavře segment cesty nakreslením úsečky z aktuálního do počátečního bodu
  - parametry nemá a mezi `Z` a `z` není rozdíl
  - segment cesty je kreslená čára nepřerušovaná příkazem *moveto*
- *lineto*: `L` nebo `l` – kreslí úsečku
  - `L100,50` – vykreslí úsečku s koncovým bodem na absolutních souřadnicích [100, 50]
  - `l100,-50` – vykreslí úsečku s koncovým bodem o 100 doprava a o 50 nahoru
- *horizontallineto*: `H` nebo `h` – kreslí vodorovnou úsečku
  - `H100` – vykreslí úsečku s koncovým bodem na absolutní souřadnici X=100
  - `h100` – vykreslí úsečku s koncovým bodem o 100 doprava
- *verticallineto*: `V` nebo `v` – kreslí svislou úsečku
  - `V50` – vykreslí úsečku s koncovým bodem na absolutní souřadnici Y=50

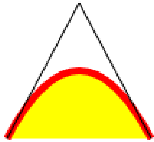
- `v-50` – vykreslí úsečku s koncovým bodem o 50 nahoru

```
<path fill="yellow" stroke="red"
      stroke-width="5" stroke-linejoin="round"
      d="M100,50 L10,10 V50 z m-20,-40 h100 l30,30 z" />
```

12.2.3.2. Složitější příkazy pro vytváření cesty

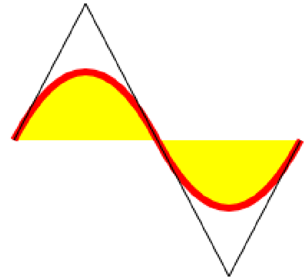
- `Q` nebo `q` – kreslí kvadratickou Beziérovu křivku
  - parametry `x1,y1 x,y`
    - `x1,y1` – souřadnice řídicího bodu (na obrázku průsečík černých čar)
    - `x,y` – souřadnice koncového bodu křivky

```
<path fill="yellow" stroke="red"
      stroke-width="5" stroke-linejoin="round"
      d="M10,100 Q60,5 110,100"
/>
<line x1="10" y1="100" x2="60" y2="5" stroke="black"/>
<line x1="60" y1="5" x2="110" y2="100" stroke="black"/>
```



- `T` nebo `t` – kreslí kvadratickou Beziérovu křivku, kde řídící bod je zrcadlením řídícího bodu z předchozího křivkového úseku – výsledkem je hladké napojení těchto dvou křivkových úseků
  - parametry `x,y`
    - `x,y` – souřadnice koncového bodu křivky

```
<path fill="yellow" stroke="red"
      stroke-width="5" stroke-linejoin="round"
      d="M10,100 Q60,5 110,100 T210,100"
/>
<line x1="10" y1="100" x2="60" y2="5" stroke="black"/>
<line x1="60" y1="5" x2="110" y2="100" stroke="black"/>
<line x1="110" y1="100" x2="160" y2="195" stroke="black"/>
<line x1="160" y1="195" x2="210" y2="100" stroke="black"/>
```



- *curveto*: `C` nebo `c` – kreslí kubickou Beziérovu křivku
  - parametry `x1,y1 x2,y2 x,y`
    - ♦ `x1,y1` – souřadnice prvního řídicího bodu
    - ♦ `x2,y2` – souřadnice druhého řídicího bodu
    - ♦ `x,y` – souřadnice koncového bodu křivky

```
<path fill="yellow" stroke="red"
      stroke-width="5" stroke-linejoin="round"
      d="M10,50 C110,-50 210,150 310,50" />
```

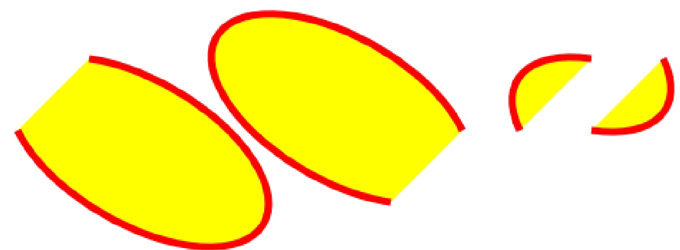
- `S` nebo `s` – kreslí kubickou Beziérovu křivku, kde první řídící bod je zrcadlením druhého řídícího bodu z předchozího křivkového úseku – výsledkem je hladké napojení těchto dvou křivkových úseků
  - parametry `x2,y2 x,y`
    - ♦ `x2,y2` – souřadnice druhého řídicího bodu
    - ♦ `x,y` – souřadnice koncového bodu křivky

```
<path fill="yellow" stroke="red"
      stroke-width="5" stroke-linejoin="round"
      d="M10,50  C110,-50 210,150 310,50 s10,0 310,0" />
```

- *elliptical arc*: `A` nebo `a` – eliptická výseč
  - parametry `rx,ry xRotace vetsiVysec poSmeru x,y`
    - ♦ `rx,ry` – poloměry os elipsy
    - ♦ `xRotace` – otočení osy X ve stupních – má význam pouze pokud se poloměry `rx` a `ry` liší
    - ♦ `vetsiVysec` – vykresluje se větší (1) nebo menší (0) než 180° výseč

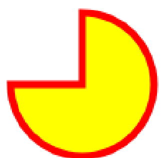
- poSmeru – kreslení od počátku po směru hodinových ručiček (1) nebo proti směru (0)
- x,y – souřadnice koncového bodu křivky

```
<path d="M100,50  a100,50 30 1 1 -50,50
         M360,100 a100,50 30 1 0 -50,50
         M450,50  a100,50 30 0 0 -50,50
         M500,50  a100,50 30 0 1 -50,50"
      fill="yellow" stroke="red" stroke-width="5" />
```



- ukázka části koláčového grafu – všimněte si v-50 a z, které způsobí uzavření a orámování výseče

```
<path d="M100,100 v-50 a50,50 0 1,1 -50,50 z"
      fill="yellow" stroke="red" stroke-width="5" />
```



12.2.4. Texty

- texty jsou v SVG plnohodnotné uzavřené objekty
  - pro výplně, tloušťky obrysů písma, jejich barvy atd. platí stejné způsoby nastavení jako pro základní geometrické tvary a cesty
- za vypisovaný text je považováno vše, co je zapsáno mezi párovými závorkami <text> a </text>
- v SVG dokumentu se doporučuje používat kódování UTF-8
- nejdůležitější atributy elementu <text>
  - x y – počáteční souřadnice levého „dolního“ rohu výpisu
    - pojem „dolní“ představuje účaří (*baseline*)
    - levý roh je implicitní nastavení – viz dále text-anchor
  - font-family – název fontu (rodiny písem)
    - zde je problém s názvem fontu – nepotřebujete-li speciální font, použijte generická jména:
      - Serif – patkové písmo

– SansSerif – bezpatkové písmo

♦ při nenalezení fontu používá většinou bezpatkové písmo

- font-size – stupeň (velikost) písma
- fill – barva (fill="none" – „obrysové“ písmo, kde stroke="black" je barva obrysové čáry)
- font-weight – duktus („tloušťka písma“), možnosti: lighter, normal, bold, bolder
- font-style – sklon písma, možnosti: italic, normal, oblique

```
<path d="M10,60 h600" stroke="gray"></path>

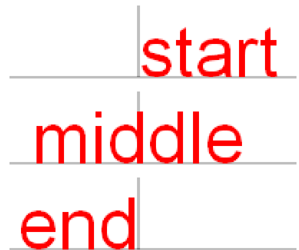
<text x="10" y="60" font-family="Serif" font-size="50"
  font-weight="bold" font-style="italic" >
  Příšerně žluťoučký kůň
</text>
<text x="10" y="110" font-family="SansSerif" font-size="50"
  fill="red">
  úpěl ďábelské ódy
</text>
<text x="10" y="190" font-family="Courier" font-size="100"
  font-weight="bold" fill="none" stroke="black" stroke-width="3">
  TEXT
</text>
```

Příšerně žluťoučký kůň
úpěl ďábelské ódy
TEXT

■ atribut text-anchor určuje, jaký význam na účaří má souřadnice x

- start – levý roh textu
- middle – střed textu
- end – pravý roh textu

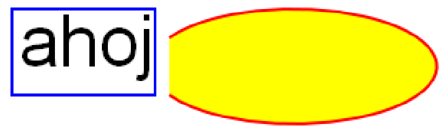
```
<path d="M10,60 h200 M100,60 v-50" stroke="gray"/>
<text x="100" y="60" text-anchor="start">
  start </text>
<path d="M10,120 h200 M100,120 v-50" stroke="gray"/>
<text x="100" y="120" text-anchor="middle">
  middle </text>
<path d="M10,180 h200 M100,180 v-50" stroke="gray"/>
<text x="100" y="180" text-anchor="end">
  end </text>
```



12.2.5. Seskupování a společné nastavení vlastností

- základní geometrické tvary, cesty a texty se mohou seskupovat
  - výhodné pro společné nastavování vlastností, transformace apod.
  - další výhoda je při případném zpracování, kdy se dá najednou pracovat se skupinami objektů
- používá se párová značka `<g>` a `</g>`
  - v ní lze nastavit tloušťky čar, barvy, způsoby zakončení apod.
- značky `<g>` se mohou vnořovat, pak platí, že vnitřní nastavení překryje vnější
  - v příkladu je seskupen obdélník s textem (jen značka `<g>` bez dalších atributů), což může později umožnit jejich společný výběr

```
<g fill="none" stroke-width="2">
  <g>
    <g stroke="blue">
      <rect x="10" y="10" width="100" height="60"/>
    </g>
    <text fill="black" x="15" y="50" font-family="SanSerif" font-size="50">
      ahoj
    </text>
  </g>
  <g stroke="red" fill="yellow">
    <path d="M120,30 a100,40 0 1,1 0,40"/>
  </g>
</g>
```



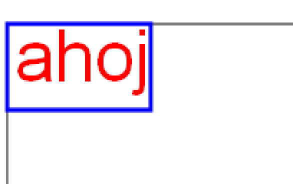
12.2.6. Transformace

- významná možnost SVG
- všechny základní elementy SVG (cesty, texty, geometrické tvary) mohou transformaci použít využitím atributu `transform`
  - prakticky je ale většinou vhodnější (čitelnější) provádět transformaci na nadřazené skupině (`<g>`)

- existuje celkem 5 základních transformací
  - zapisují se do atributu `transform`
  - dají se navzájem kombinovat
- všechny transformace budou ukázány na stejném obrázku (tj. stejném kódu SVG), kde bude použit i ohraničující obdélník
  - to znamená, že v dalších ukázkách bude měněna pouze jedna řádka elementu `<g>`

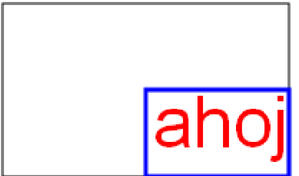
```
<rect x="10" y="10" width="200" height="120"
      stroke-width="1" stroke="black" fill="none"/>

 <g> <!-- zde bude transformace -->
    <g fill="none" stroke-width="3">
    <g stroke="blue">
      <rect x="10" y="10" width="100" height="60"/>
    </g>
    <text fill="red" x="15" y="50" font-family="SanSerif" font-size="50">
      ahoj </text>
  </g>
</g>
```



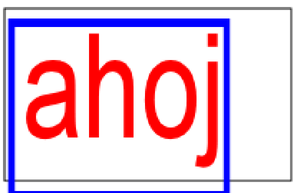
- `translate(tx,ty)` – posun
  - `tx` a `ty` udávají posun na příslušné ose

```
<g transform="translate(100,60)">
```



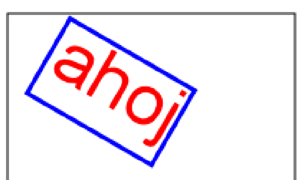
- `scale(sx,sy)` – zvětšení
  - `sx` a `sy` udávají zvětšení na příslušné ose
  - na obrázku v porovnání s ohraničujícím obdélníkem je dobře vidět, jak transformace fungují
    - v příslušných měřítcích jsou zvětšeny i souřadnice počátku [10,10]
      - proto se modrý obdélník, který je na výšku zvětšen dvakrát, nevejde do ohraničujícího obdélníka
    - transformace se aplikují na všechna relevantní nastavení vykreslovaného objektu, tj. i na tloušťky čar apod.

```
<g transform="scale(1.5,2)">
```



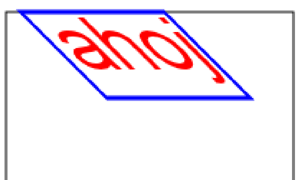
- `rotate(uhel)` – otočení
  - `uhel` udává otočení ve stupních
  - na příkladu je vidět i seskupení transformací, kdy se nejprve provede posun a pak otočení
    - posun je nutný, protože otočení je prováděno vůči bodu [0,0]
    - SVG norma udává možnost `rotate(uhel [stredX,stredY])`, kde se stanoví střed otáčení – tuto funkčnost se mi ale nepodařilo ověřit

```
<g transform="translate(50,0) rotate(30)">
```



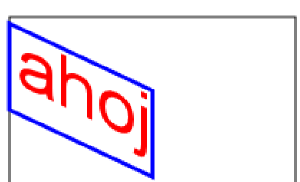
- `skewX(uhel)` – zkosení podél osy x
  - `uhel` udává zkosení ve stupních

```
<g transform="skewX(45)">
```



- `skewY(uhel)` – zkosení podél osy y
  - `uhel` udává zkosení ve stupních

```
<g transform="skewY(25)">
```



12.2.7. Výplň – složitější možnosti

- pro výplň se používá atribut `fill` – viz výše
  - vyplňuje jednou neprůhlednou barvou
    - je možné výplňovou barvu dále upřesňovat

Varování

Tyto možnosti nemusejí být podporovány v profilech SVGT a SVGB!

12.2.7.1. Průhlednost

- lze nastavit pomocí atributu `fill-opacity`
  - hodnotou je číslo v rozsahu <0,0..1,0>.
    - nulová hodnota značí plně průhlednou výplň (0 %)
    - jedničková pak úplnou neprůhlednost (100 %)
- příklad ukáže možnosti nastavení průhlednosti i pro skupinu

```
<text fill="black" x="15" y="140" font-family="SanSerif" font-size="80">
    Cena 99,- Kč </text>

<g fill-opacity="1">
  <g transform="translate(130,0) rotate(30)">
    <rect x="0" y="0" width="350" height="100" fill="orange" rx="30"/>
    <text fill="red" x="10" y="90" font-family="SanSerif"
          font-size="100" font-weight="bold">
      SLEVA </text>
  </g>
</g>
```

pro `<g fill-opacity="1">`

pro <g fill-opacity="0.6">

12.2.7.2. Barevné přechody

- výplň je pomocí několika barev
- element `<stop>` – definuje tzv. „přechodový bod“
  - používá se pro nastavení jednotlivé barvy a průhlednosti v přechodu
  - musí být použit minimálně dvakrát – pro počáteční a koncovou barvu
  - je možné mít barevný přechod z více barev
  - atributy
    - `offset` – místo, ve kterém má barva nejvyšší intenzitu
      - udává se poměrově, typicky 0% pro výchozí barvu a 100% pro cílovou barvu
    - `stop-color` – barva přechodu – zadává se běžným způsobem
    - `stop-opacity` – průhlednost této barvy v přechodu
- aby bylo možné (pracně) připravený barevný přechod vícekrát využít, připravuje se jeho definice pomocí elementu `<defs>`
- lineární barevný přechod – element `<linearGradient>`
  - atributy
    - `id` – pojmenování pro pozdější odkaz
    - `x1 y1 x2 y2` – definice směru přechodu – vždy hodnoty čtveřice `x1 y1 x2 y2` (počáteční a koncový bod úsečky)
      - `0 0 0 1` – svislý přechod
      - `0 0 1 0` – vodorovný přechod
      - `0 0 1 1` – přechod z levého horního rohu do pravého dolního
      - `0 1 1 0` – přechod z levého dolního rohu do pravého horního
  - vnořené elementy – minimálně dva elementy `<stop>`

- příklad ukazuje definici lineárního přechodu
  - název `CervenoModroZluta`
  - směr přechodu je vodorovný
  - barvy přechodu
    - červená – velmi průhledná (`0.2`)
    - žlutá – neprůhledná, s maximální intenzitou v polovině (`50%`)
    - modrá – téměř neprůhledná (`0.8`)
  - obdélník s přechodem využívá pojmenovanou definici pomocí

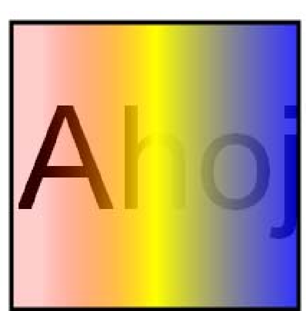
    `fill="url(#CervenoModroZluta)"`

    text **Ahoj** je zde vypsán pod obdélníkem, aby bylo možné vidět funkci průhlednosti jednotlivých barev

```
<text fill="black" x="15" y="140" font-family="SanSerif" font-size="100">
  Ahoj </text>

<defs>
  <linearGradient id="CervenoModroZluta"
    x1="0" y1="0" x2="1" y2="0">
    <stop offset="0%" stop-color="red" stop-opacity="0.2"/>
    <stop offset="50%" stop-color="yellow" />
    <stop offset="100%" stop-color="blue" stop-opacity="0.8"/>
  </linearGradient>
</defs>

<rect fill="url(#CervenoModroZluta)" stroke="black" stroke-width="3"
     x="10" y="10" width="200" height="200"/>
```



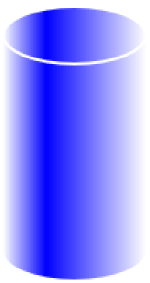
- podobný barevný přechod (pouze svislý) použitý pro výplň textu

```
<text x="10" y="110" font-family="SansSerif" font-size="100"
      font-weight="bold" fill="url(#CervenoModroZluta)"
      stroke="black" stroke-width="0">
  AHOJ
</text>
```

- ukázka barevných přechodů pro zobrazení efektních grafů
  - posun sloupečku na správné místo je řešen pomocí `<g transform="translate(10,30)">`

```
<defs>
  <linearGradient id="Modra"
      x1="0" y1="0" x2="1" y2="0">
    <stop offset="0%" stop-color="white"/>
    <stop offset="30%" stop-color="blue"/>
    <stop offset="100%" stop-color="white"/>
  </linearGradient>
</defs>

<g transform="translate(10,30)">
  <g fill="url(#Modra)" stroke-width="0" stroke="blue">
    <rect x="0" y="0" width="100" height="150"/>
    <ellipse cx="50" cy="00" rx="50" ry="20"/>
    <ellipse cx="50" cy="150" rx="50" ry="20"/>
  </g>
  <ellipse cx="50" cy="0" rx="50" ry="20"
    fill="none" stroke-width="2" stroke="white"/>
</g>
```



Poznámka

Kromě lineárního přechodu existuje ještě kruhový – element `<radialGradient>`.

12.2.8. Animace

- SVG má velmi široké možnosti animací
  - animace je deklarativní – je přesně deklarováno jaký konkrétní atribut konkrétního objektu bude animován
    - to znamená změnu hodnoty tohoto atributu v definovaném
      - rozsahu hodnot (od – do)

– čase (počáteční a koncový čas animace)

♦ existuje obrovská škála možností, co se týče hodnot i času

■ základní možnosti jsou však jednoduché a jsou založeny na použití elementu `<animate>`

12.2.8.1. Element `<animate>`

■ vkládá se do kresleného objektu

■ základní atributy `<animate>`

- `attributeName` – jméno atributu z animovaného objektu, který se bude měnit
  - ♦ v daném čase lze měnit jen jeden atribut
    - – toto se dá snadno obejít použitím dalšího elementu `<animate>`, protože jejich počet ve vykreslovaném objektu není omezen
    - – tak lze průběžně měnit více atributů vykreslovaného objektu najednou
- `from` – počáteční hodnota animace

  koncová hodnota se udává dvěma různými atributy (používá se vždy jen jeden z nich)
  - ♦ `to` – skutečná koncová hodnota (`from="10" to="100"`)
  - ♦ `by` – velikost změny (`from="10" by="90"`)
- `fill` – zachování výsledku animace – možnosti:
  - ♦ `freeze` – výsledek animace zůstane zachován i po jejím skončení
  - ♦ `remove` – po provedení animace se stav vrátí k původnímu stavu
- `begin` – co spustí animaci
  - ♦ zadává se v sec (např. `begin="2s"`) nebo v milisec (např. `begin="2000ms"`)
    - – za jak dlouho po načtení SVG souboru animace začne
  - ♦ další možností je `begin="click"`
    - – animace začne při kliknutí na objekt
    - – dalším kliknutím ji lze opakovat
- `dur` – doba trvání animace – opět v sec či milisec

■ příklad ukáže čtyři základní animace

- žlutý obdélník za 2 sec začne zvětšovat svoji šířku až do koncové hodnoty 100, celé to trvá 3000 msec a po skončení se výsledný obrázek zachová

```
<rect x="10" y="10" width="10" height="30"
      fill="yellow" stroke="blue" stroke-width="2">
```

```
  <animate attributeName="width" from="10" to="100"
           begin="2s" dur="3000ms" fill="freeze"/>
</rect>
```

- po kliknutí na světlezelený obdélník se začne zvětšovat jako předchozí a po skončení se vrátí do původního stavu

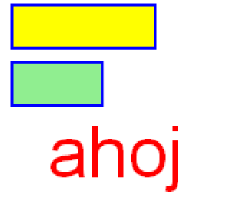
```
<rect x="10" y="50" width="30" height="30"
      fill="lightgreen" stroke="blue" stroke-width="2">
  <animate attributeName="width" from="30" by="70"
           begin="click" dur="3s" fill="remove"/>
</rect>
```

- text se zobrazí vpravo a za 3 sec začne po dobu 5 sec „najíždět“ doleva

```
<text x="100" y="130" fill="red"
      font-family="SanSerif" font-size="50">
  <animate attributeName="x" from="100" to="10"
           begin="3s" dur="5s" fill="freeze"/>
  ahoj
</text>
```

- červený svislý sloupec vpravo začíná okamžitě dvě vteřiny růst směrem vzhůru – to je umožněno dvojitou animací, kdy se mění `height` a `y`, a současně mění barvu z červené na světle modrou, ale po delší dobu tří sec

```
<rect x="250" y="180" width="30" height="0"
      fill="red" stroke="blue" stroke-width="2">
  <animate attributeName="height" from="0" to="150"
        begin="0s" dur="2s" fill="freeze"/>
  <animate attributeName="y" from="180" to="30"
        begin="0s" dur="2s" fill="freeze"/>
  <animate attributeName="fill" from="red" to="lightblue"
        begin="0s" dur="3s" fill="freeze"/>
</rect>
```



12.2.8.2. Složitější příklad

- ukázka animace dříve uvedeného sloupcového grafu s gradientními přechody

```
<title>Animace - lineární gradientní přechody</title>
```

```
<defs>
  <linearGradient id="Modra"
    x1="0" y1="0" x2="1" y2="0">
    <stop offset="0%" stop-color="white"/>
    <stop offset="30%" stop-color="blue"/>
    <stop offset="100%" stop-color="white"/>
  </linearGradient>
</defs>

<g transform="translate(10,30)">
  <g fill="url(#Modra)" stroke-width="0" stroke="blue">
    <ellipse cx="50" cy="150" rx="50" ry="0">
      <animate attributeName="ry" from="0" to="20"
        begin="0s" dur="1s" fill="freeze"/>
    </ellipse>
    <rect x="0" y="0" width="100" height="0">
      <animate attributeName="height" from="0" to="150"
        begin="1s" dur="3s" fill="freeze"/>
      <animate attributeName="y" from="150" to="0"
        begin="1s" dur="3s" fill="freeze"/>
    </rect>
    <ellipse cx="50" cy="00" rx="50" ry="0">
      <animate attributeName="ry" from="0" to="20"
        begin="4s" dur="1s" fill="freeze"/>
    </ellipse>
  </g>
  <ellipse cx="50" cy="0" rx="50" ry="0"
    fill="none" stroke-width="2" stroke="white">
    <animate attributeName="ry" from="0" to="20"
      begin="5s" dur="1s" fill="freeze"/>
  </ellipse>
  <text x="30" y="200" fill="white"
    font-family="SanSerif" font-size="20" font-weight="bold">
    <animate attributeName="y" from="150" to="10"
      begin="6s" dur="2s" fill="freeze"/>
    150
  </text>
</g>
```

150

12.3. SVG a Java

- při pokusu využít formát SVG v Java programu narazíme téměř okamžitě na knihovnu **Batik**

- `xmlgraphics.apache.org/batik`
- v prosinci 2009 ve verzi 1.7

- jedná se o vyzrálou knihovnu poskytující množství užitečných funkcí
  - přímo použitelné nástroje – dají se spustit z příkazové řádky nebo a Antu
  - knihovny – využitelné v Java programu

12.3.1. Přímo použitelné nástroje z Batik

- *Squiggle SVG Browser* – komfortní prohlížeč s mnoha možnostmi, např. ovládání animací, doplňování SVG dokumentu apod.
  - téměř úplně splňuje normu SVG Full 1.2
  - ideální pro „vývoj“ SVG dokumentu, pokud jej vytváříme jako XML dokument (ne použitím grafického editoru)
  - spouštění

```
java -jar batik-squiggle.jar
```

- *SVG Rasterizer* – převod vektorového formátu SVG na bitmapovou verzi (JPG, PNG a TIFF)
  - dokáže i převod do PDF
  - spouštění např.:

```
java -jar batik-rasterizer.jar krivky.svg -m application/pdf
```

- *SVG Font Converter* – konvertuje True Type fonty do SVG formátu, který ukládá do SVG dokumentu ve formě definice (element `<defs>`)
  - SVG dokument s textem pak vypadá na jakémkoliv počítači stejně
  - spouštění

```
java -jar batik-ttf2svg.jar arial.ttf -o mujArial.svg
```

- `SVG Pretty Printer` – formátování SVG dokumentu
  - dá SVG dokumentu „štábní kulturu“
    - pokud SVG dokument připravujeme ručně nějakým XML editorem, pak je dodatečné formátování většinou zbytečné
    - je to velmi užitečný nástroj, pokud SVG dokument připravujeme např. XSLT transformací
  - umožňuje řadu drobných „vychytávek“ typu různé konce řádek (CRLF či LF), maximální délka řádky, apod.
  - spouštění

```
java -jar batik-svgpp.jar krivky.svg formatovane-krivky.svg
```

12.3.2. Programování pomocí knihoven Batiku

- v Java programu můžeme samozřejmě využít všech výše zmíněných nástrojů
- kromě nich máme k dispozici několik velmi zajímavých možností

12.3.2.1. Generování SVG dokumentu pomocí Java Graphics2D

- idea je, že kreslíme v Javě pomocí běžně známých nástrojů ze třídy `java.awt.Graphics2D` a tento obrázek je automaticky generátorem SVG převeden do SVG dokumentu
  - výhoda – tímto způsobem lze do SVG relativně velmi snadno (změna hlavičky metody `paint()`) převést grafiku vytvořenou v Java programech
- příklad ukáže, jak vytvořit soubor `ctverec.svg` s červeným čtvercem

```
public class Swing2DdoSVG {

  // funkce pro kresleni
  public static void paint(Graphics2D g2d) {
    g2d.setPaint(Color.red);
    g2d.fill(new Rectangle(10, 10, 100, 100));
  }

  public static void main(String[] args) throws IOException {
    // instance DOMImplementation
    DOMImplementation domImpl =
           GenericDOMImplementation.getDOMImplementation();

    // instance org.w3c.dom.Document.
    String svgNS = "http://www.w3.org/2000/svg";
    Document document = domImpl.createDocument(svgNS, "svg", null);

    // instance SVG generatoru bude ukladat do DOM
    SVGGraphics2D svgGenerator = new SVGGraphics2D(document);

    // paint() zapisuje do generatoru
    paint(svgGenerator);

    Writer zapis = new OutputStreamWriter(
                    new FileOutputStream("ctverec.svg"),
                    "UTF-8");
    boolean useCSS = false;       // nepouzivat CSS styly attributu
    svgGenerator.stream(zapis, useCSS);
  }
}
```

12.3.2.2. Práce se SVG obrázkem

- využijeme v případě, že chceme obrázek nejen zobrazit, ale mít i možnost reagovat na události uživatele nad tímto obrázkem
  - typický případ je zobrazení mapy, nad kterou pak program poskytuje další služby
- složitost této akce je v rozporu s jednoduchostí používání ostatních částí Batiku a přesahuje rámec tohoto textu
  - bohužel i dokumentace Batiku k této části je velmi strohá

12.4. Generování SVG pomocí XSLT

- vytváření SVG pomocí XSLT je jedním z častých způsobů
  - lze využít všech možností XSLT
    - v příkladu je to např. volba měřítka na ose Y
- pro XML soubor `jidlo.xml`

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

- XSLT šablona `jidlo-graf.xsl`

```
<?xml  version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="2.0">

  <xsl:output method="xml"
    encoding="UTF-8"/>

  <xsl:variable name="xSloupec" select="100"/>
  <xsl:variable name="yGraf" select="200"/>
```

```
<xsl:variable name="okraj" select="50"/>
<xsl:variable name="fontNadpis" select="30"/>

<!-- odvozene konstanty -->
<xsl:variable name="xSloupecGraf" select="($xSloupec)*0.7"/>
<xsl:variable name="maxCena"
          select="max(for $x in //ovoce
               return $x/vaha * $x/nazev/@jednotkovaCena)"/>
<xsl:variable name="yMeritko"
          select="(($yGraf) - ($fontNadpis)*2)div($maxCena)"/>

<xsl:template match="jidlo">
  <xsl:variable name="xMax" select="count(ovoce)*$xSloupec"/>
  <!-- svg korenovy element -->
  <svg xmlns="http://www.w3.org/2000/svg"
       width="{$xMax+$okraj}px" height="{$yGraf+$okraj}px">
    <g id="bar" transform="translate(10,10)">
      <!-- svisla osa -->
      <line stroke="black" x1="0" y1="0" x2="0" y2="{$yGraf}"/>
      <!-- vodorovna osa -->
      <line stroke="black" x1="0" y1="{$yGraf}"
            x2="{$xMax}" y2="{$yGraf}"/>

      <text font-family="SansSerif" font-size="{$fontNadpis}"
            font-weight="bold" fill="blue" text-anchor="middle"
            x="{($xMax)div(2)}" y="{$fontNadpis}">
        Útrata za ovoce
      </text>

      <!-- spolecne nastaveni fontu pro popis ovoci -->
      <g font-family="SansSerif" font-size="{($fontNadpis)*0.5}"
        font-weight="normal" text-anchor="middle">

      <xsl:for-each select="ovoce">
        <xsl:variable name="xStred"
             select="(@cislo - 1)*$xSloupec + ($xSloupec)*0.5"/>

        <!-- popis na ose X -->
        <text x="{$xStred}" y="{$yGraf + ($fontNadpis)*0.5}">
          <xsl:value-of select="nazev"/>
        </text>
        <!-- cena sloupec -->
        <xsl:variable name="yVyska"
               select="nazev/@jednotkovaCena*vaha*$yMeritko"/>
        <xsl:variable name="barva" select="(@cislo)*40"/>

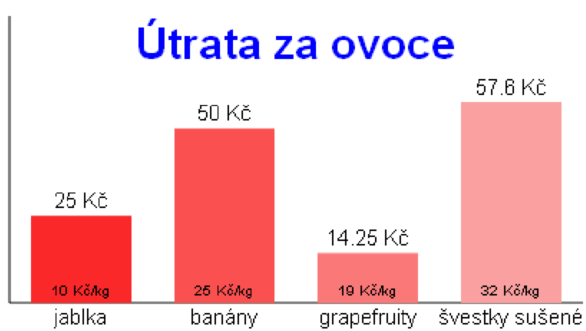
        <line x1="{$xStred}" y1="{($yGraf)-$yVyska}"
         x2="{$xStred}" y2="{$yGraf}" stroke="rgb(250, {$barva}, {$barva})"
          stroke-width="{$xSloupecGraf}" stroke-linecap="butt"/>
        <text x="{$xStred}" y="{($yGraf)-5}"
             font-size="{($fontNadpis)*0.33}">
          <xsl:value-of select="nazev/@jednotkovaCena"/> Kč/kg
        </text>
```

```
          <text x="{$xStred}" y="{($yGraf)-($yVyska)-5}"
                font-size="{($fontNadpis)*0.5}">
            <xsl:value-of select="nazev/@jednotkovaCena*vaha"/> Kč
          </text>
        </xsl:for-each>
        </g>
      </g>
    </svg>
  </xsl:template>

</xsl:stylesheet>
```

■ vytvoří soubor `jidlo-graf.svg`

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg" width="450px" height="250px">
  <g id="bar" transform="translate(10,10)">
    <line stroke="black" x1="0" y1="0" x2="0" y2="200"/>
    <line stroke="black" x1="0" y1="200" x2="400" y2="200"/>
    <text font-family="SansSerif" font-size="30" font-weight="bold"
          fill="blue" x="200" y="30" text-anchor="middle">
      Útrata za ovoce </text>
    <g font-family="SansSerif" font-size="15" font-weight="normal"
       text-anchor="middle">
      <text x="50" y="215">jablka</text>
      <line x1="50" y1="139.23611111111111" x2="50" y2="200"
            stroke="rgb(250, 40, 40)"
            stroke-width="70" stroke-linecap="butt"/>
      <text x="50" y="195" font-size="9.9">10 Kč/kg </text>
      <text x="50" y="134.23611111111111" font-size="15">25 Kč </text>
      <text x="150" y="215">banány</text>
      <line x1="150" y1="78.47222222222223" x2="150" y2="200"
            stroke="rgb(250, 80, 80)"
            stroke-width="70" stroke-linecap="butt"/>
      <text x="150" y="195" font-size="9.9">25 Kč/kg </text>
      <text x="150" y="73.47222222222223" font-size="15">50 Kč </text>
      <text x="250" y="215">grapefruity</text>
      <line x1="250" y1="165.36458333333334" x2="250" y2="200"
            stroke="rgb(250, 120, 120)"
            stroke-width="70" stroke-linecap="butt"/>
      <text x="250" y="195" font-size="9.9">19 Kč/kg </text>
      <text x="250" y="160.36458333333334" font-size="15">14.25 Kč </text>
      <text x="350" y="215">švestky sušené</text>
      <line x1="350" y1="60" x2="350" y2="200"
            stroke="rgb(250, 160, 160)" stroke-width="70"
            stroke-linecap="butt"/>
      <text x="350" y="195" font-size="9.9">32 Kč/kg </text>
      <text x="350" y="55" font-size="15">57.6 Kč </text>
    </g>
  </g>
</svg>
```


Útrata za ovoce
25 Kč
50 Kč
14.25 Kč
57.6 Kč
10 Kč/kg
25 Kč/kg
19 Kč/kg
32 Kč/kg
jablka
banány
grapefruity
švestky sušené