Príručka Java

Práca s XML

IT Academy s.r.o., Tomášikova 50/A, 831 04 Bratislava, tel.: 0917/095 406, 0907/375 543
IČO: 46 759 786, DIČ: 2023556766, IČ DPH: SK2023556766

Obsah:

Túto príručku môžete využiť ako pomôcku pri práci s Java. **Je určená výhradne pre absolventov kurzu Java spoločnosti IT Academy a VITA Company.** Bez informovania autora je zakázané príručky používať na webových stránkach alebo rozširovať v tlačovej podobe. **Príručka podlieha autorským právam a jej vlastníkom je spoločnosť IT Academy s.r.o.**

I. Java a XML, SAX

Java a XML

Programovací jazyk **Java** vytvorí spoločne s **XML** veľmi dobrú symbiózu. **Java** je prenositeľný programovací jazyk a **XML** je prenositeľný formát pre **popis dát**.

Najčastejšie akcie vykonávané pri spracovaní **XML**:

- **Načítanie XML elementov**
- **Generovanie nových elementov alebo oprava stávajúcich sa elementov**
- **Zápis do XML dokumentu**
- **Transformácia XML do iných formátov**

Najhoršie, čo môžeme s XML dokumentom pri spracovaní vykonať, je považovať ho za textový súbor a napísať vlastný program, ktorý jednotlivé elementy, atribúty a ich hodnoty postupne získava **analýzou** načítaných textových riadkov.

Túto akciu potrebuje každý a pretože je **XML** jednoznačne **definovaný** formát, existuje už množstvo hotových knižných programov, ktoré vykonajú rozdelenie **XML** dokumentu na jednotlivé časti **(parsing)** za nás. Týmto nástrojom sa hovorí **parsery (parser)** a vo svojej podobe sa nachádzajú v knižniciach každého moderného programovacieho jazyka.

Všeobecné vlastnosti parsera

Bez ohľadu na použitý programovací jazyk by **parsel** mal mať nasledujúce **schopnosti**:

- Čítať **XML dokument** zo súboru alebo všeobecne zo vstupného prúdu. Pri čítaní sa automaticky vykonáva kontrola správnej štruktúrovanosti **(well-formed)** a nevyhovujúci XML dokument je odmietnutý a problém je viac či menej presne lokalizovaný. Voliteľne je možné vykonávať aj validáciu podľa **DTD** alebo **XSD**.
- Behom čítania extrahuje názvy **elementov** a **atribútov** a ich **hodnoty**. Ďalej konzistentne spracováva všetky ďalšie pomocné informácie z **XML** súborov, ktoré sú napr. komentáre, inštrukcie pre spracovanie entity, CDATA sekcie a pod.
- **Parter** cez programátorské rozhranie **(API)** ponúka abstraktný model **XML** dokumentu, tzv. **infoset**. Pomocou infosetu s XML súborom nepracujeme ako s obyčajným textovým súborom, ale posúvame svoju prácu o úroveň vyššie.

Takže pracujeme s jednotlivými časťami XML dokumentu – **elementy, atribúty, textovým obsahom elementov.**

Spôsobov spracovania **XML** dokumentov je veľké množstvo. U každého všeobecného spôsobu je popísané ako **rozhranie (API),** ktorému vyhovuje určitý **parser**. To znamená, že nie každý parser, ktorý nájdeme, je možné použiť pre ľubovoľný spôsob spracovania.

Základné delenie rozhrania je na **prúdové čítanie** (respektíve spracovanie) a na **prácu so stromovou reprezentáciou dokumentu.**

Prúdové čítanie

Niekedy je tento spôsob nazývaný **udalosťami riadené spracovanie.** Jeho typickým predstaviteľom je rozhranie **Simple API for XML** alebo **SAX.**

Princíp spracovania je ten, že parser postupne číta **XML dokument** a pre každú ucelenú časť vyvolá udalosť. Zjednodušene povedané je našou úlohou napísať obsluhy týchto udalostí.

Poznámka: Jedná sa teda o implementáciu podľa **návrhového vzoru** (design pattern) **vydavateľ – odberateľ.**

Výhody tohto spôsobu spracovania sú:

- **Veľká** rýchlosť načítania a **malá** pamäťová náročnosť.
- Jedná sa o všeobecne známe a všade podporované **API** – teraz vo verzii SAX2.

Ak sa budeme pohybovať len na platforme Javy, potom je súčasťou **Java Core API** od **JDK 1,4.**

Základné nevýhody:

- **XML dokument** je spracovávaný sekvenčne, čo prakticky znamená, že pri čítaní nie je možné sa vrátiť. Všetky načítané údaje je nutné buď priebežne spracovávať, alebo ukladať v nami zvolenej organizácii do pamäti. Táto organizácia **nemusí** mať (a veľmi často ani nemá) **priamy** obraz s načítaným XML dokumentom (predovšetkým jeho stromovú štruktúru).
- **Princíp obsluhy udalostí** predstavuje veľmi **nízkoúrovňové** spracovanie, ktoré typicky vyžaduje množstvo pomocných premenných alebo vlastnú nadstavbu.

II. Práca so stromovou reprezentáciou dokumentu

Celý XML dokument sa načíta naraz do stromovej štruktúry v pamäti. To je významný **rozdiel** od predchádzajúceho spôsobu.

Základný predstaviteľ je **Document Object Model** čiže **DOM** navrhnutý **W3C**.

Výhody tohto spôsobu spracovania sú:

- Celý infoset je dostupný v pamäti, čo prakticky znamená, že sa pri spracovaní údajov môžeme ľubovoľne vracať, preskakovať nepotrebné časti – všeobecne **pohybovať sa** po strome.
- Je možné nie len čítanie XML dokumentu, ale aj zápis do neho. Načítaný dokument môžeme **meniť**, generovať nové elementy a na záver **uložiť** späť do XML dokumentu.
- Rovnako ako u SAX sa jedná o všeobecne známe a všade podporované **API** – teraz vo verzii **DOM Level 2**. Ak sa budeme pohybovať len na platforme **Javy**, potom je súčasťou **Java Core API** od **JDK 1,4.**
- Značnou výhodou je dobrá spolupráca s dotazovacím jazykom pre XML nazývaným **XPath (XML Path Langugage).**

Základné nevýhody:

- Malá rýchlosť načítania a veľká pamäťová náročnosť. To je dané tým, že sa v pamäti musí postupne vytvoriť celá **stromová štruktúra**. Nepriaznivým dôsledkom je, že tento spôsob je možné použiť (pri veľkostiach RAM okolo 1GB) len do rady asi 100MB veľkosti XML dokumentu.

- **API** je dosť všeobecné a tým pre niektoré použitie zbytočne zložité a to hlavne pre začiatočníkov.

Poznámka: Okrem týchto dvoch základných rozhraní sa objavuje ďalšie špecializovanejšie rozhranie zjednodušujúce prácu pre konkrétne prípady XML dokumentov. Ich spoločnou nevýhodou je, že nie sú tak všeobecne známe a preto často ani nebývajú súčasťou základných knižníc daného jazyka.

III. Rozhranie parserov pre Javu

JAXP – Java API for XML Processing

Jedná sa o dôležité API rozhranie začlenené do **Java Core API**. Vo verzii JDK 1.5 je verzia JAXP 1.3, vo verzii JDK 1.6 je JAXP 1.4.

JAXP 1.4 definuje celkom **22 balíkov**, v ktorých sú definované Java rozhrania pre širokú oblasť práce s XML dokumentmi.

JAXP neprináša žiadne nové spôsoby spracovania **XML dokumentov**, ale vďaka balíkom rozhrania zjednocuje spôsoby používania už existujúcich parserov. Tým dovoľuje používateľovi, aby používal **jednotným** spôsobom **rôzne spôsoby** spracovania (napr. **SAX** alebo **DOM**) a rôzne parsery (napr. pre DOM sú to **Xerces** alebo **Crimson**).

Tak je zaistené, že pri zmene parseru buď za novú verziu, alebo za iný typ sa v spracovaní XML dokumentov nič **nezmení**. Ak využívame rozhranie **JAXP**, sme odlíšení od drobných rozdielností parserov a ich verzií.

JDOM

Jedná sa o **špeciálne** rozhranie len pre **Javu**, ktoré vzniklo ako snaha pri zjednodušení príliš všeobecného **DOM** s bežnými dokumentmi a dáva možnosť jednoduchšieho spracovania.

IV. Stručný prehľad parserov

Je treba zdôrazniť, že nie všetky možnosti sú dostupné v každom **programovacom jazyku**, prípadne v každej verzii jeho knižnice.

Podľa **spôsobu spracovania XML dokumentov** rozoznávame:

1. **Prúdové čítanie**
    - **Push parsery** – napr. **SAX**. Tu čítanie XML dokumentov prebieha automaticky, po prečítaní určitého úseku generuje parser rôzne typy udalostí, na ktoré v našom programe reagujeme.
    - **Pull parsery** – napr. **StAX**. Čítanie XML prebieha na našu žiadosť po častiach, to znamená, že udalosti generujeme my a parser na ne reaguje vrátením príslušného načítaného úseku XML dokumentu.
2. **Práca so stromovou reprezentáciou dokumentu**
    - **DOM**, kedy dokument v pamäti v podstate **kopíruje** štruktúru XML dokumentov. **JAXB**, kedy je vykonané **objektové mapovanie.**
    - V pamäti pracujeme s objektmi, na ktoré boli automaticky prevedené jednotlivé elementy **XML súboru.**

Podľa možností **validácie XML dokumentov** rozoznávame:

- **Nevalidujúce parsery**, kedy bez ohľadu na existenciu príslušného DTD alebo XSD súboru parser validácie nevykonáva.
- **Validujúce parsery**, ktoré pred spracovaním, XML dokument validujú.

Filozofický prístup k validácii je ten, že XML dokument by ma byť validovaný externe pred tým, než ich začneme spracovávať. To, že sú **dáta** v poriadku (tzn. vyhovujú validácii), by mal zaistiť ten, kto ich **poskytuje**, nie ten kto ich spracováva.

Ten kto dáta poskytuje, má kompletnú možnosť ich opravy. Ak sa zistí **nekonzistentnosť** dát až pri nasledujúcom spracovaní, nemusí byť ich oprava principiálne možná, pretože spracovávateľ pre ňu nemusí mať dostatok informácií. Validácia **spomaľuje** spracovanie.

Poznámka: Validácia je mienená validácia oproti príslušnému schémovému súboru. Kontrolu správnej štruktúrovanosti XML dokumentu vykonáva každý parser a nie je možné ju vypnúť.

Okrem bežne očakávaných funkcií parserov, môžu mať parsery ešte zabudovaný manažér entít. Ten má význam pokiaľ sa **XML dokument** skladá z viac súborov. Manažér potom z týchto jednotlivých súborov zloží výsledný **XML dokument.**

Konkrétne parsery

Každé rozhranie musí byť nakoniec implementované pomocou konkrétneho **parsera**, ktorých je značné množstvo.

Ďalej bude používaný parser **Xerces** (Xerces.apache.org), ktorý je cez **JAXP** súčasťou Java **Core API** od JDK 1.4. Ten sme používali externe už pre validáciu podľa **DTD** či **XSD**.

Xerces zvláda okrem toho základné zmieňované rozhrania **DOM** a **SAX** a tiež dokáže pracovať s mennými priestormi.

V. SAX – Simple API for XML

Jedná sa o **prúdové** spracovanie **XML** dokumentov typu **push-parser.**

SAX2 je definovaný niekoľkými rozhraniami v **Jave Core API** popísanými v balíku **org.xml.sax.helpers.** Používaný parser **„odtienený“** cez **JAXP** je v balíku **javax.xml.parsers**.

Základný postup pri spracovaní

Program s dôsledným využitím JAXP postupne vytvorí objekt k parseru umožňujúcemu podľa API SAX2 prečítať súbor **jedlo.xml.** Žiadnu inú činnosť program nevykonáva. Je ale možné ho použiť ako **verifikátor** (well-formed) – ak nevypíše chybu, súbor **XML** je v poriadku.

Postup pri vytváraní všetkých objektov nutných pre správne **parsovanie** sa bude v ďalších príkladoch takmer bez zmeny opakovať.

Hlavný program

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

public class VerificatorJidloSAX {

    private static final String SUBOR =
“jidlo.xml“;

    public static void main(String[] args) {
        try {
            SAXarserFactory spf =
            SAXParserFactory.newInstance();
            Spf.setValidating(false);
            SAXParser saxLevel1 = spf.newSAXParser
            ();
            XMLReader parser =
            saxLevel1.getXMLReader ();
            parser.setErrorHandler(new
            ChybyZisteneParserem());
            parser.setContentHandler(new
            DefaultHandler());
            parser.parse(SUBOR);
            System.out.println(SUBOR + “ precitany
            bez chyb“);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Význam jednotlivých programových sekvencií:

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

Vytvorí **„obálku“** pre **univerzálny parser**. Tu je možné vykonať nastavenie konfigurácie pre budúci parser. Prakticky sa využíva možnosť nastavenia **validity** podľa **XSD** a spracovanie jemných priestorov. Konkrétny objekt parseru v tejto dobe ešte **neexistuje.**

```
spf.setValidation(false);
```

Validícia nebude vykonávaná.

```
SAXParser saxLevel1 = spf.newSAXParser();
```

Vytvorí SAX parser vyhovujúci Level 1, tj. staršiu verziu **SAX,** označovanú tiež ako **SAX1.** Všimnite si, že v príkaze nie je zrejmé, aký skutočný parser sa použije. My už vieme, že implicitným parserom pre **JAXP JDK 1.6 je Xerces**.

```
XMLReader parser = saxLevel1.getXMLReader();
```

Trieda **org.xml.sax.XMLReader** je nadstavba nad **SAXParse**r spĺňajúca požiadavky **SAX2.** Umožňuje mimo iného definovať vlastnú obsluhu podľa **SAX2.**

Tento spôsob vytvárania parseru vyhovujúcemu **SAX2** cez **SAX1** bol zavedený z dôvodu **spätnej kompatibility**.

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

Jedná sa o nastavenie reakcie na chyby, ktoré nie je nutné uskutočňovať, pokiaľ je XML súbor v poriadku, tj. **validovaný pred spracovaním**. Tým si však nemôžeme byť nikdy istý, takže sa jedná o vhodnú činnosť, ktorá nás v prípade problémov so spracovaním **XML** dokumentom upozorní na prípadné chyby.

Trieda **ChybyZisteneParserom()** sa bude bez akejkoľvek zmeny **opakovať** vo **všetkých ďalších** programoch.

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

Nastavenie obsluhy pre čítanie **XML** dát, tj. ako sa bude XML dokument skutočne spracovávať. Toto nastavenie nerobí nič – **DefaultHandle**r je prázdny adaptér. Pre skutočné spracovanie XML dokumentu sa vytvorí naša obslužná trieda ako potomok triedy **DefaultHander** a v nej sa preferujú potrebné metódy.

V prípade, že meno súboru obsahuje medzery alebo akcenty bolo v JDK 1.5 nutné namiesto:

```
parser.parse(SUBOR);
```

použiť:

```
parser.parse(new InputSource(new FileReader(newFile(SUBOR))));
```

inak bola vypísaná výnimka:

```
java.net.MalformedURLException: no protocol: jedlo mňam.xml
```

Spracovanie výnimiek v hlavnom programe

Pokiaľ sme si istí, že pri spracovaní XML dokumentu nenastanú problémy, je možné použiť jednoduchú reakciu typu:

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

Potenciálne ale môžu pri spracovaní XML dokumentu vzniknúť **tri typy výnimiek** a v prípade problémov je pre ich lokalizáciu možné použiť hierarchiu výnimiek:

```
try {
    skôr uvedený kód
}
catch (SAXExceptation e) {
    výnimky vzniknuté pri parsovaní XML dokumentu
}
catch (ParserConfigurationException e) {
    výnimky vzniknuté pri nastavovaní vlastnosti
parserov
}
catch (IOExceptation e) {
    výnimky vzniknuté pri čítaní XML dokumentu zo
súboru alebo z prúdu
}
```

Pomocná trieda pre spracovanie chýb

Chyby vzniknuté pri parsovaní XML dokumentu sa podľa **JAXP** delia do troch kategórií popísaných metódami z rozhrania **org.xml.sax.ErrorHandler**. Pokiaľ chceme tieto kategórie rozlišovať, stačí pripraviť vlastnú triedu implementujúcu rozhranie **ErrorHandler**. Touto triedou bude trieda **ChybyZisteneParserom**, ktorá sa bude opakovať v programoch.

Tromi spomenutými kategóriami sú **varovné hlásenia, chyby** a **kritické chyby**, ktoré sú v rozhraní **ErrorHandler** predstavované metódami **warning(), error() a fatalError().** Tie dostanú ako svoj skutočný parameter objekt výnimky typu **org.xml.sax.SAXParseExceptation**.

Z tohto objektu získajú informácie o čísle riadku a stĺpca súboru, v ktorom bol problém nájdený, vrátane popisu chyby ich sformulujú do hlásenia. To sa v prípade varovného hlásenia iba vypíše. V prípade chyby či kritickej chyby sa toto hlásenie predá ako popis novo vygenerovanej výnimky typu **org.xml.sax.SAXException**, zachytená v hlavnom programe.

Pokiaľ čítame **XML dokument** z prúdu, nemajú hodnoty čísla riadkov ani stĺpcov význam a často majú hodnotu **-1.**

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

public class ChybyZisteneParserom implements
ErrorHandler {
// Sformatovanie textu hlaseni
private String textHlasenie(SAXParseException e) {
return e.getSustemId() + "\n"
+ "riadok: " + e.getLineNumber()
+ " stlpec: " + e.getColumnNumber()
+"\n" + e.getMessage();
}
// obsluha varovnych hlaseni
public void warning(SAXParseException e) {
System.out.println("Varovanie: " + textHlaseni(e));
}
// obsluha chyb
public void error(SAXParseException e) throws
SAXException {
throw new SAXException("Chyba: " + textHlaseni(e));
}
// obsluha fatalnych chyb
public void fatalError(SAXParseException e) throws
SAXException {
throw new SAXException("Kriticka chyba: " +
textHlaseni(e));
```

Pokiaľ sa nechceme zaťažovať s prípravou tejto triedy, je možné využiť triedu **org.xml.sax.helpers.DefaultHanler**, ktorá pomocou prázdnych metód implementuje mimo iné aj rozhranie **ErrorHandler**.

VI. Spracovanie parsovaného XML dokumentu

V balíku **org.xml.sax** je niekoľko rozhraní, ktoré je možné využiť pri našom spracovávaní XML súboru. Spracovanie vykonáme tak, že tieto rozhrania implementujeme našimi triedami a objekty týchto tried predáme parseru pomocou jeho set...() metód. Z asi 10 uvedených rozhraní sú potrebné:

- **ErrorHandler** – reakcie na chyby
- **ContentHandler** – spracovanie elementov
- **Attributes** – spracovanie atribútov

Teraz sa budeme zaoberať rozhraním **ContentHandler**. Objekty vyhovujúce tomuto rozhraniu sú predávané parseru pomocou jeho metódy **setContentHandler().**

Trieda **org.sax.helpers.DefaultHandler** z predchádzajúceho príkladu toto rozhranie implementuje ale prázdnymi metódami. Ako už bolo povedané, vo všetkých príkladoch budeme spracovávať súbor **jedlo.xml** s obsahom.

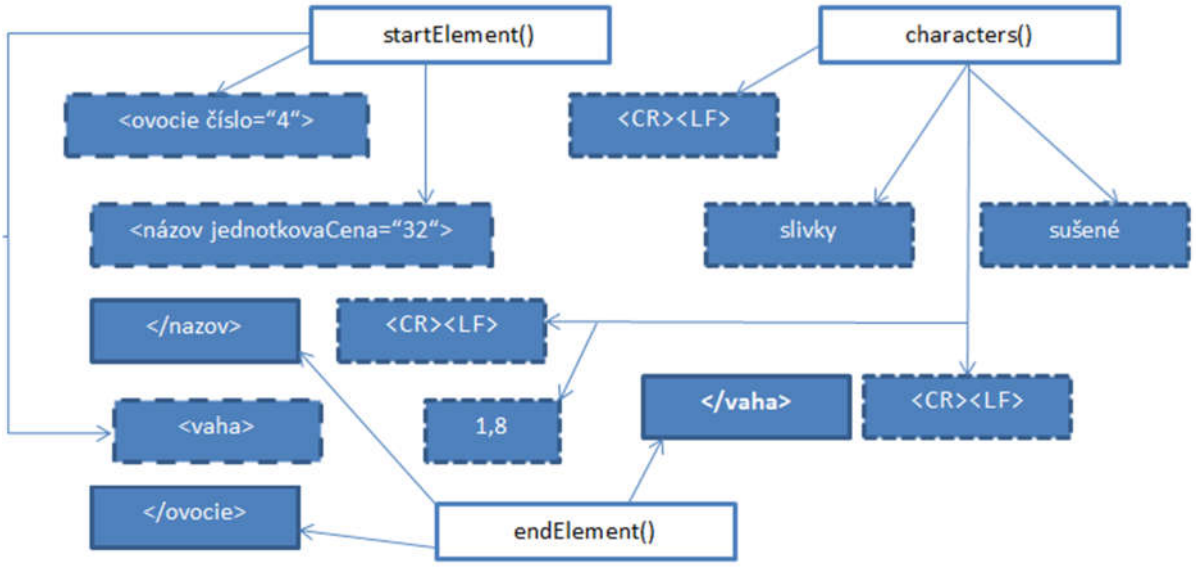
```
<?xml version = "1.0" encoding = "windows-1250"?>
<jidlo>
<ovocie cislo="1">
<nazov jednotkovaCena = "10">jablka</nazov>
<vaha>2.5</vaha>
</ovocie>
<ovocie cislo="2">
<nazov jednotkovaCena = "25">banany</nazov>
<vaha>2</vaha>
</ovocie>
<ovocie cislo = "3">
<nazov jednotkovaCena = "19">grapefruity</nazov>

<vaha>0.75</vaha>
</ovocie>
<ovocie cislo = "4">
<nazov jednotkovaCena = "32">slivky susene</nazov>
<vaha>1.8</vaha>
</ovocie>
</jedlo>
```

Z veľkého počtu metód rozhrania **ContentHandler** využívame iba nasledovné metódy:

- **startDocument()** – udalosť na začiatku dokumentu je vhodnejšia než konštruktor, pretože umožňuje viacnásobné opakovanie čítania dokumentu.
- **startElement()** – je volaná pri načítaní počiatočnej značky a prípadných atribútov.
- **characters** – vracia obsah elementu
- **endElement()** – je volaná pri načítaní koncovej značky.

Posledné tri spomenuté metódy z rozhrania **ContentHandler** aplikované na jeden element <ovocie> majú teda nasledujúci rozsah pôsobnosti:



Obr. 1 Rozsah pôsobnosti metód z rozhrania

Celý problém spracovania konkrétneho XML dokumentu spočíva v tom, že pripravíme svoju triedu zdedenú od **org.xml.sax.helpers.Default-Handler**, do ktorej napíšeme metódy startDocument(), startElement(), characters() a endElement().

VII. Práca s obsahom elementov

Program vypíše celkovú **váhu** nakúpeného ovocia. V hlavnom programe (súbor VahaJedloSAX.java) sa zmenia len riadky:

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

Celé spracovanie je popísané v triede **VahovyHandler**.

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

public class VahovyHandler extends DefaultHandler {
    private static final int VELKOST_BUFFEROV =
    1000;
    private static final String MENO_ELEMENTOV=
    "vaha";
    private double celkovavaha;
    private StringBuffer hodnota = new
    stringBuffer(VELKOST_BUFFEROV)
    private boolean vovnutriVahy;
    public double getCelkovaVaha() {
        return celkovaVaha;
    }
```

```
    public void startDocument() {
        celkovaVaha = 0
    }
    public void startElement(String uri, String
localName,
                     String qName, Attributes atts)
{
        if (qName.equals(MENO_ELEMENTOV) == true) {
            vnutriVahy = true;
            hodnota.setLength(0);
        }
    }
    public void endElement(String uri, String
localName,
                     String qName) {
        if (qName.equals(MENO_ELEMENTOV) == true) {
            vovnutriVahy = false,
            celkovaVaha +=
            double.parseDouble(hodnota.toString());
        }
    }
    public void characters(char[] ch, int start,
int length) {
        if (vovnutriVahy == true) {
            hodnota.append(ch, start, length);
        }
    }
}
```

Význam parametrov metód **startElement()** a **endElement** je:

- **uri** – URI menného priestoru, ak nie je možné menný priestor použiť, tak je uri prázdny reťazec
- **localName** – meno elementu v súvislosti s menným priestorom

Varovanie: Ak nie je možné menný priestor použiť, potom je **localName** prázdny reťazec, nie meno elementu.

- **qName** – úplné (kvalifikované) meno elementu, používame pre XML dokumenty bez menného priestoru
- **atts** – zoznam atribútov

Význam parametrov metódy **characters()** je nutné popísať podrobnejšie. Parameter **ch** predstavuje **pole znakov**, v ktorom je uložená hodnota elementov.

Pri podrobnejšom skúmaní však zistíme, že **ch** obsahuje mnohonásobne viac, než len hodnotu konkrétneho elementu.

- Pre krátke **XML dokumenty** obsahuje celý **textový súbor**. Preto je hodnota aktuálneho elementu určená ďalšími dvoma parametrami. Prvým je štart, ktorý predstavuje index počiatočného znaku aktuálnej hodnoty a druhým je **length** s významom počtu znakov.

Aby situácia nebola jednoduchá, musíme počítať s **dvoma problémami**:

1. Hodnota elementu môže prijať po **častiach,** tzv. v niekoľkých volaných metódach **characters()** postupne. Preto je nutné čiastkové hodnoty ukladať (spojovať) postupne do **StringuBufferov** a čítať celok až v metóde **endElement().**

2. Okrem hodnôt elementov obsahuje **ch** taktiež znaky medzi elementmi (typicky odriadkovanie), ktoré nás nezaujímajú – je nutné testovať, či sme vo vnútri konkrétneho elementu.

   Prakticky to prevedieme pomocou **booleovskej** premennej (v príklade nazvané **vovnutriVahy**), ktorú v **startElement()** nastavíme a v **endElement()** zhodíme.

Poznámka: Pri spracovaní je treba mať na pamäti, že SAX je typu **push-parser,** kedy pripravované metódy nikde nevoláme my, ale volá ich **parser** sám.

VIII. Práca s atribútmi

Z predchádzajúceho príkladu bolo zrejmé, že získanie jedného typu hodnoty z celého **XML dokumentu** je veľmi ľahké. Pokiaľ dokument nie je príliš hierarchicky zložitý, je pre viac typov hodnôt spracovania zložitejší, ale situácia je stále prehľadná.

Program (súbor CenaJedloSAX.java) spracováva **atribúty** a vypočíta **celkovú cenu nákupu**. V hlavnom programe sa opäť **zmení** niekoľko riadkov týkajúcich sa **definície** obslužnej triedy a záverečného **výpisu** výsledkov.

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

V metódach **startElement()** a **endElement()** potrebujeme pomoc booleovských premenných aby sme rozlíšili, v ktorom elemente sa nachádzame. Našťastie je možné previesť pomocou konštrukcie **else-if**, takže zložitosť zápisu sa zvyšuje len **lineárne**.

Spracovanie atribútov je pri základnom použití veľmi jednoduché. Ak poznáme **meno atribútu**, je možné jeho hodnotu získať jednoducho volaním metódy **getValue().**

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

```
public class CenovyHandler extends DefaultHandler {
    private static final int VELKOST_BUFFEROV =
    1000;
    private double celkovaCena;
    private boolean vovnutriVahy;
    private StringBuffer hodnota = new
    StringBuffer(VELKOST_BUFFEROV);
    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) {
        vovnutriVahy = true;
        hodnota.setLength(0);
}
    else if (qName.equals("nazov") == true) {
        jednotkovaCena = Integer.parseInt(
            atts.getValue("jednotkovaCena"));
    }
}
```

```
public void endelement(String uri, String
localName,
                String qName) {
    if (qName.equals("vaha") == true) {
        VovnutriVahy = false;
        Vaha =
        Double.parseDouble(hodnota.toString());
    }
    else if (qName.equals("ovocie") == true) {
        celkovaCena += vaha * jednotkovaCena;
    }
}
public void characters(char[] ch, int start, int
length) {
    if (vovnutriVahy == true) {
        hodnota.append(ch, start, length);
        }
    }
}
```

Prevod infosetu na zoznam objektov

Program uloží všetky hodnoty a atribúty naraz do pamäti a potom ich spracováva. V pamäti budú uložené objekty triedy **Ovocie**, ktorá má pre jednoduchosť veľmi stručný **obsah.**

```
public class Ovocie {
    int cislo;
    String nazov;
    int jednotkovaCena;
    double vaha;
```

```
    public Ovocie(int cislo, String nazov,
                  int jednotkovaCena, double vaha) {
        this.cislo = cislo;
        this.nazov = nazov;
        this.jednotkovaCena = jednotkovaCena;
        this.vaha = vaha;
    }
    public String toString() {
        return "" + cislo + "." + nazov + " – "
            + vaha + " [kg] po "
            + jednotkovaCena + " [Kc] = "
            + vaha * jednotkovaCena + " [Kc]";
    }
}
```

V pamäti budú vzniknuté objekty triedy **Ovocie** uložené v zozname typu **ArrayList<Ovocie>,** z ktorého bude kedykoľvek možné získať potrebné informácie. Bude sa jednať o **celkovú váhu** a **celkovú cenu**. Ďalšou službou bude vytlačenie všetkých načítaných ovocí.

```
import java.util.*;

public class SpracovanieDatVpamati {
    public static void
vytlacvsetko(ArrayList<Ovocie> ar) {
        for (Ovocie o: ar) {
            System.out.println(o);
        }
    }
    public static double
```

```
    celkovaVaha(ArrayList<Ovocie> ar) {
        double celkovaVaha = 0;
        for (Ovocie o: ar) {
            celkovaVaha += o.vaha;
        }
        Return celkovaVaha;
    }
    public static double
celkovaCena(ArrayList<Ovocie> ar) {
        double celkovaCena = 0;
        for (Ovocie o: ar) {
            celkovaCena += o.vaha *
        o.jednotkovaCena;
        }
        return celkovaCena;
    }
}
```

Handler spracovávajúci **XML dokument** bude takmer zhodný z predchádzajúcim handlerom. Jediná podstatná zmena je v metóde **endElement(),** kde z načítaných hodnôt vzniká nový objekt typu **Ovocie** uložený do zoznamu.

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

```
public class VsetkovpametiHandler exdents
DefaultHandler {
    private static final int VELKOST__BUFFEROV =
    1000;
    private static ArrayList<Ovocie> ar = new
    ArrayList<Ovocie>;
    private StringBuffer hodnota = new
    StringBuffer(VELKOST_BUFFEROV);
    private boolean vovnutriElementu;
    private int cislo;
    private string nazov;
    private int jednotkovaCena;
    private double vaha;
    public ArrayList<Ovocie> getZoznam() {
        return ar;
    }
    public void startDocument() {
        ar.clear();
    }
    public void startElement(String uri, Stirng
    localName,
                        String qName, Attributes
                    atts) {
        if(qName.equals(“ovocie“) == true) {
            cislo =
        integer.parseInt(atts.getValue(“cislo“));
        }
```

```
        else if (qNAme.equals("nazov") == true) {
            jenotkovaCena = Integer.parseInt(
                        atts.getValue("jenotkovaCena"
                    ));
            hodnota.setLength(0);
            vovnutriElementu = true;
        }
        else if (qName.equals("vaha") == true) {
            hodnota.setLength(0);
            vovnutriElementu = true;
        }
    }
    public void endElement(String uri, String
    localName, String qName) {
        if (qName.equals("vaha") == true) {
            Vaha =
            double.parseDouble(hodnota.toString());
            vovnutriElementu = false;
        }
        else if (qName.equals("nazov") == true) {
            Nazov = hodnota.toString();
            vovnutriElementu = false;
        }
        else if (qName.equals("ovocie") == true) {
            Ar.add(new Ovocie(cislo, nazov,
        jednotkovaCena, vaha));
        }
    }
    public void characters(char[] ch, int start, int
    length) {
        if (vovnutriElementu == true) {
            Hodnota.append(ch, start, length);
        }
    }
```

V hlavnom súbore(**VsetkovpamatiSAX.java**) sa opäť zmení len veľmi malá časť:

```
...
VsetkovpamatiHandler ph = new
VsetkovpamatiHandler();
Parser.setContentHandler(ph);
Parser.parse(SUBOR);
ArrayList<Ovocie> ar = ph.getZoznam();
SpracovanieDatVpamati.vytlacVsetko(ar);
System.out.println("Celkova vaha = "
            +
        SpracovanieDatVpamati.celkovaVaha(ar));
System.out.println("Celkova cena = "
            +
        SpracovanieDatVpamati.celkovaCena(ar));
...
```

IX. Spracovanie zložitejšieho XML dokumentu

Doposiaľ uvedený postup je **priamočiary**, nemusí byť však optimálny pre zložitejšie XML dokumenty. Problémy by nastali v prípade, že by XML dokument:

- Mal **veľký** počet elementov. Potom by konštrukcie **else-if** v **startElement()** a **endElement()** boli rozsiahle podľa toho, koľko by existovalo rôznych elementov.
- Mal rovnako pomenované **elementy** vnorené v rôznych elementoch. Príkladom môže byť napr. **element dátum**, ktorý by pri bankovom účte bol súčasťou elementu **založenie Účtu** a potom súčasťou každého z elementov vklad a výber.

  Tu by bola situácia oveľa horšia, než v predchádzajúcom prípade, pretože by bolo **nutné** sledovať o aký dátum sa jedná pomocou zložených **logických** podmienok. Tým by sa obsahy metód **startElement()** a **endElement()** stali značne neprehľadnými.
  V tomto prípade je možné situáciu riešiť napr. tým, že pre každý vnorený element pripravíme samostatnú triedu ako potomka **DefaultHandler** a prepíname ich. Postup má výhodu v tom, že každý takto vzniknutý samostatný **handler** má svoj **startElement(), endElement()** a **characters(),** tj. Ich kód je jednoduchý.

Nevýhodou je, že je nutné vyriešiť **prepínanie** jednotlivých **handlerov**.

Jedným z možných riešení je nájsť záujemcov v súboroch **ViacHandlerovSAX.java**, **OvocieHandler.java**, **NazovHandler.java** a **VahaHandler.java**.

Poznámka: Pre jednoduchý XML dokument ako je **jedlo.xml**, ale nie je na prvý pohľad zrejmý prínos tohto riešenia.

Sofistikovanejší spôsob prepínania handlerov je možné prehliadnuť v súbore **ObezitologiaSAX.java.**

Problematika rôzneho kódovania

XML dokumenty môžu byť, ako je známe, uložené v mnohých rôznych kódovaniach. Pokiaľ má XML dokument v hlavičke uvedené použité kódovanie, a následne správne načítanie záležitosti, o ktorú sa automaticky postará **parser**. Problematika správneho kódovania je teda prenesená na toho, kto dáta pripravuje.

Hlavička XML dokumentu má tvar napr.

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

Pri príprave XML dokumentu alebo chybne uvedenom kódovaní XML dokumentu sa dá problém riešiť pomocou triedy **org.xml.sax.InputSource**. V jej konštruktore je možné nastaviť taký **InputStream**, ktorý vyhovuje použitému kódovaniu.

Nastavenie vlastností parseru

Pri sofistikovanejšom spracovaní XML dokumentu sa dostaneme do situácie, kedy nám prestane vyhovovať štandardné nastavenie parseru.

Ovplyvniť jeho chovanie je možné v **4 úrovniach.**

Prvá úroveň je úroveň **JAXP,** kedy trieda **javax.xml.parsers.SAXParserFactory** dáva priamo k dispozícii metódy pre ovplyvnenie dvoch najpoužívanejších vlastností parseru.

Jedná sa o:

- **Zapnutie/vypnutie validácie** – metóda setValidating(bolean validating)
- **Použitie menných priestorov** – metóda setNamespaceAware(boolean awareness)

Ak si nevystačíme s týmito 2 nastaveniami, používame z triedy **SAXParserFactory** metódu:

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

kde **name** je meno vlastnosti a **value** je zapnutie či vypnutie tejto vlastnosti.

Pokiaľ použijeme meno všeobecných vlastností, tj. vlastností podporovaných parserom **JAXP**, dostávame sa na **druhú úroveň** nastavení chovania parserov. Opäť sa jedná o celkom prenositeľné nastavenie, ktoré nie je závislé na konkrétnom parseri.

Zoznam mien všeobecných vlastností je možné ako **„SAX2 Standard Feature Flags“** nájsť na:

Jdkl.5.0/docs/api/org/xml/sax/package-summary.html alebo

Jdkl.6.0/docs/api/org/xml/sax/package-summary.html

Pri nastavovaní vlastností je treba dávať pozor na to, že tieto mená sú v tvare **URL**, ktoré sa musí presne opísať.

Takže napr. typická **chyba**, pri nastavovaní podpory menných priestorov je:

```
spf.setFeature(“namespaces“, true); // CHYBA
```

pričom správne nastavenie má byť

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

Poznámka: Toto nastavenie dáva rovnaký výsledok ako predtým uvedené **spf.setNamespaceAware(true);** z prvej úrovne.

Tretia úroveň nastavenia vlastnosti je **neprenositeľná**, pretože už nastavujeme špeciálne vlastnosti pre konkrétny používaný **parser**.

Poznámka: Je asi zrejmé, že túto akciu nie je dobré vykonávať bez vážneho dôvodu.

Opäť používame metódu **setFeature()** triedy **SAXParseFactory**, samozrejme názvy vlastností musíme získať z dokumentácie konkrétneho **parseru**. Napríklad pre **Xerces** je možné zoznam vlastností nájsť na:

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

A vykonávanie validácie podľa **XSD** schémy sa zapne príkazom:

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

Keď si prehliadneme uvedené vlastnosti, zistíme, že niektoré sú i iného typu než **zapnúť/vypnúť**. Práca s nimi prestavuje **štvrtú** úroveň nastavenia. Vykonávame ju pomocou metódy triedy **javax.xml.parsers.SAXParser**

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

Napríklad nastavenie, kedy sa pre validáciu oproti **XSD** bude používať súbor **jedlo.xsd**, ktorý nie je pripojený do **XML** súboru **jedlo.xml**:

```
sp.setProperty(
        “http: //
        java.sun.com/xml/jaxp/properties/schemaSour
        ce“,
        new File(“jedlo.xcd“));
```

X. Validácia oproti DTD alebo XSD

Môže sa stať, že nebude možné validovať XML dokument externe, pred tým než ho začneme spracovávať. Napríklad pokiaľ XML dokumenty prichádzajúce po sieti alebo sú generované on-line pod iným programom. Potom je možné dodať do nášho programu validáciu podľa **DTD** alebo **XSD**.

Nasledujúci program **ValidatorSAX.java** je možné využiť ako **externý** validátor, kedy pri **validnom** XML dokumente vypíše len **počet** jeho elementov.

Použitie:

- iba kontrola správnej štruktúrovanosti

```
>java validatorSAX jedlo.xml
jedlo.xml: 13 elementov 8 atributov
```

- **DTD** je pripojené v súbore

```
>java ValidatorSAX jedlo-dtd.xml dtd
jedlo-dtd.xml: 13 elementov 8 atribútov
```

- **XSD** je pripojené v súbore

```
>java ValidatorSAX jedlo.xml xsd jedlo.xsd
jedlo.xml: 13 elementov 8 atributov
```

V prípade **nevalidného** XML dokumentu vypíše napríklad:

```
>java ValidatorSAX jedlo-xsd-chyba.xml xsd
chyba: file/// D:/java-a-xml/priklady/kap05/jedlo-
xsd-chyba.xml
riadok: 7 stlpec: 23
cvc-datatype-valid.1.2.1: "velka" is not valid
value for
"double".
```

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

public class ValidatorSAX {

  public static void main(String[] args) {
    if (args.length < 1) {
      System.out.println("Syntaxa: "
                         + "java ValidatorSAX
                         <subor.xml>"
                         + "[dtd | xsd] [subor.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].toLoerCase().equals("xsd") =
    true) {
      xsd = true;
    }
  }
  try {
    SAXParserFactory spf =
    SAXParserFactory.newInstance();
    spf.setValidating(true);
    spf.setFeature(http://xml.org/sax/features/na
    mespaces, true);
      // spf.setNamespaceAware(true); // rovnaka
      uloha
    if (dtd == false && xsd == false) {
      spf.setValidating(false);
    }
    else if (xsd == true) {
       spf.setFeature(
         "http://apache.org/xml/features/validat
         ion/schema,true);
    }
```

```
    SAXParser sp = spf.newSAXParser();
    if (xsd == true) {
      sp.setProperty(
       "http://java.sun.com/xml/jaxp/properties/s
      chemaLanguage,
       "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
    ValidaciaChybyZistenaParserom());
    PocetElementovHandler h = new
    PocetElementovHandler());
    parser.setContentHandler(h);

    parser.parse(args[0]);
    System.out.println(args[0] +": "
    +h.getVysledok());
  }
```

```
catch (Exception e) {
  String s = e.getMessage();
  int i =
  s.indexOf(ValidaciaChybyZistenaParserom.KON
  IEC);
  if (i> 0) {
  s = s.substring(0, i);
  }
  System.out.println(s);
  }
  }
  }
class PocetElementovHandler extends
DefaultHandler {
   private int pocetElementov = 0;
   private int pocetAtributov = 0;
   public void startElement(String uri,
   String localName, String qName, Attributes
   atts) {
     pocetElementov++;
     pocetAtributov += atts.getLength();
   }
   public String getVysledok() {
     return "" +pocetElementov + " element"
            +pocetAtributov + " atributov";

   }

}
```

Pomocná konštanta **KONIEC** triedy ValidaciaChybyZistenaParserom je využívaná preto, aby sa z textu chybového hlásenia dali odrezať rušivé výpisy o lokalizácií vzniknutej výnimky.

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

public class ValidaciaChybyZistenaParserom
implements ErrorHandler {
  public static final String KONIEC =
  "koniecHlaseni";
// Sformatovanie textu hlasenia
private String textHlasenia(SAXParseException
e) {
  return "System ID: " + e.getLineNumber() +
  "\n" +
        "riadok: " + e.getLineNumber() + "
        stlpec: " +
        e.getColumnNumer() +"\n" +
        e.getMessage() + KONIEC;
}

// obsluha varovnych hlaseni
public void warning(SAXParseException e ) {
  System.out.println("Varovanie: "+ text
  hlasenia());
}

// obsluha chyb
```

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

    // obsluha fatalnych chyb
    public void fatalError(SAXParserException e)
    throws SAXException {
      throw new SAXException("Fatalna chyba: "+
      textHlaseni());
    }
  }
```

Pokiaľ máme nainštalované **JWSDP**, je možné podobný profesionálny program pre validáciu prezerať v :

\ jwsdp-2.0\jaxp\samples\sax\SAXLocalNameCount.java

XI. Práca s mennými priestormi

Spracovávať XML dokument s mennými priestormi je veľmi jednoduché - oproti predchádzajúcim spôsobom sa prakticky nič nemení. Kľúčové je zapnúť pri **SAXParserFactory** spracovanie menných priestorov príkazom:

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

Bez tohto nastavenia je aj XML dokument s mennými priestormi čítaný akoby v ňom menné priestory **neboli**. Budeme spracovávať súbor **jedlo-ns.xml**, ktorý začína:

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

  <potrava:ovocie cislo = "1">
    <potrava:nazov potrava:jednotkovacena =
    "10">jablka
    </potrava:nazov>
    <potrava:vaha>2.5</potrava:vaha>
  </potrava:ovocie>
```

Ako je známe, atribúty môžu, ale aj nemusia mať **označenie** menného priestoru – porovnaj číslo verzus **potrava:jednotkovaCena**.

Spracovanie bude prebiehať v súboroch **VypisVsetkehoNSJedloSAX.java** a **VypisVsetkehoNSHandler.java**, z ktorého uvedieme iba metódu **startElement().**

```
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)
                    + ", c=value = " +
                    atts.getValue(i));
   }
}
```

Pri zapnutom **spf.setNameSpaceAware(true)**; vypisuje:

```
1.    start    uri    =    http://www.kiv.zcu.cz/~
herout/xml/jedlo-sada,~
 localName = jedlo, qName = potrava:jedlo
2.    start    uri    =    http://www.kiv.zcu.cz/~
herout/xml/jedlo-sada,
 localName = ovocie, qName = potrava:ovocie
  atts uri = , localName = cislo, qName = cislo,
  type = CDATA, value = 1
```

```
3. start uri = http://www.kiv.zcu.cz/~herout/xml/jedlo-sada,
 localName = nazov, qName = potrava:nazov
  atts uri = http://www.kiv.zcu.cz/~herout/xml/jedlo-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/jedlo-sada,
 localName = nazov, qName = potrava:nazov
4. start uri = http://www.kiv.zcu.cz/~herout/xml/jedlo-sada,
 localName = vaha, qName = potrava:vaha
2. ch = 2.5, start = 204, length = 3
```

Pri vypnutom spf.setNamespaceAware(false); vypisuje:

```
1. start uri = localName = , qName = potrava:jedlo
  ats uri = , localName = , qName = xmlns: potrava,
  type = CDATA,
 value = http://www.kiv.zcu.cz/~ herout/xml/jedlo-sada
2. start uri = , localName = , qName = potrava:ovocie
 atts uri = , localName = qName = cislo, type = CDATA, value = 1
3. start uri = , localName, qName = potrava:nazov
  atts uri = , localName = , qName = potrava:jednotkovaCena,
 type= CDATA, value= =
```

```
1. ch = jablka, start = 162, length = 6
1. end uri = , localName = , qName = potrava:nazov
4. start uri = , localName = , qName = potrava:vaha
2. ch = 2.5, start = 204, length = 3
```

Z výpisu je vidieť, že **qName** vždy obsahuje presne to, čo je v XML dokumente uvedené, takže pre jednoduchšie prípady je možné používať iba **qName**. Pokiaľ sa nejedná o takto triviálny príklad, musí sa testovať uri a **localName**, pretože prefix si každý autor XML dokumentu môže voliť ako chce.

Typ atribútu **getType(i)** nemá vo väčšine prípadov praktický význam.

XII. DOM – Document Object Model

Základné informácie

Jedná sa o štandard pripravený konsorciom **W3C**. Ako už vieme, parser **DOM** načíta celý **XML** dokument do pamäte a vytvorí tam **stromovú objektovú reprezentáciu**. Objektom stromovej štruktúry sa hovorí **uzly**.

Prístup, počas ktorého je všetko potrebné naraz v pamäti, umožňuje, aby DOM slúžil aj pre zmenu alebo vytváranie nových XML dokumentov. Všetko potrebné pre prácu s DOM nájdeme **v Java Core API**. Spôsob pomenovania balíkov je analogický spôsob popísaný v **SAX**, vrátane konkrétnej implementácie pomocou **JAXP.**

Triedy zaisťujúce vlastné parsovanie nájdeme v **javax.xml.parsers** – jedná sa o **DocumentBuilderFactory** a **DocumentBuilder**.

Rozhranie popisujúce možnosti jednotlivých uzlov nájdeme v balíku **org.w3c.dom**. Tu je popísané pomocou rozhrania viac ako 10 typov uzlov, pričom pre bežné používanie ich stačí približne päť.

- **Node** – základný prvok a predok nasledujúcich potomkov
- **Document** – počiatočný uzol
- **Attr** – atribúty
- **Element** – elementy
- **Text** – hodnoty elementov

Atribúty tvoria samostatné uzly, ktoré ale nie sú potomkami príslušného elementu. Pre pripomenutie si znova ukážeme XML dokument, ktorý bude spracovávaný súbor **jedlo.xml**:

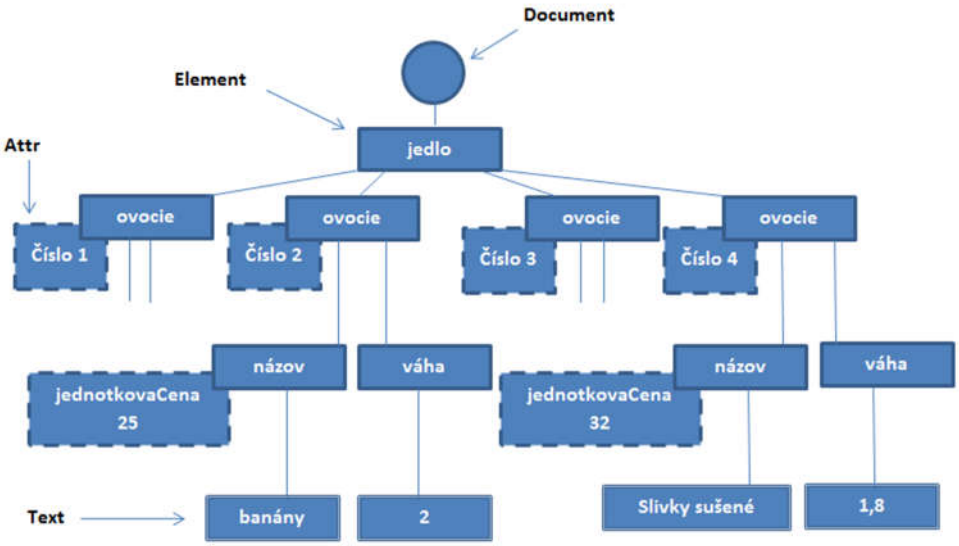
```
<?xml version = "1.0" encoding = "windows-1250"?>
<jedlo>
  <ovocie cislo = "1">
    <nazov jednotkovaCena = "10">jablka</nazov>
    <vaha>2.5</vaha>
  </ovocie>

  <ovocie cislo = "2">
    <nazov jednotkovaCena = "25">banany</nazov>
    <vaha>2</vaha>
  <ovocie/>

  <ovocie cislo = "3">
    <nazov jednotkovaCena =
    "19">grapefruity</nazov>
    <vaha>0.75</vaha>
  </ovocie>

  <ovocie cislo ="4">
    <nazov jednotkovaCena = "32">slivky
    susene</nazov>
    <vaha>1.8</vaha>
</ovocie>
</jedlo>
```

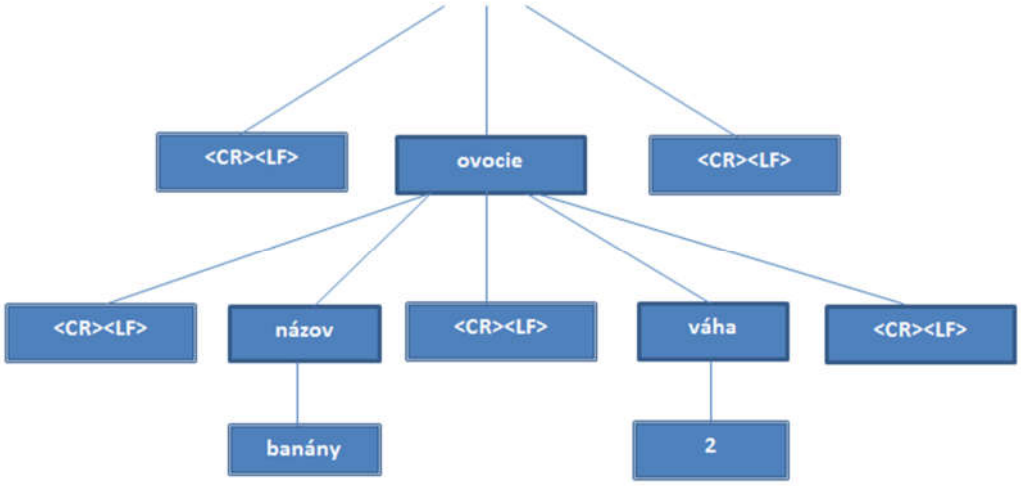
Po načítaní bude v pamäti nasledujúca štruktúra uzlov:



Obr. 2 Štruktúra uzlov

Rovnako ako pri SAX parser spracováva a ukladá aj konce riadkov a formátovacie medzery vnútri elementov. Tieto biele znaky ukladá do uzlov typu **Text** a s týmito uzlami je treba pri práci so stromom dokumentu počítať.

V skutočnosti vyzerá element ovocie takto:



Obr. 3 Element ovocie

Existuje aj rozhranie **CharacterData**, principiálne podobné **characters()** zo SAX, ktoré však nie je potrebné využívať. Používa sa jeho **potomok** – uzol typu Text, v ktorom je už hodnota elementu kompletná.

Problémy SAX-u s postupným načítaním hodnoty elementu tu neexistujú.

Kolekcia uzlov

Okrem uzlov **Node** a jeho potomkov sú veľmi užitočné ešte dve kolekcie uschovávajúce skupinu uzlov. Prvá je kolekcia podobná známemu zoznamu **java.util.List** názvom **NodeList**. Predstavuje usporiadaný zoznam elementov, ku ktorým sa pristupuje napríklad pomocou indexu.

Druhá je kolekcia podobná **Map** s názvom **NameNodeMap**. Jedná sa o asociatívne pole a používa sa pre uloženie mien atribútov a ich hodnôt, pričom prístup je typický pomocou mena atribútov.

Základné použitie DOM

Rovnako, ako to bolo pri SAX si najprv ukážeme postup vytvárania kostry programu, pričom sa postupy vytvárania objektov bude v ďalších programoch takmer identicky opakovať.

Program vytvorí objekty, ktoré umožnia metódou DOM prečítať súbor **jedlo.xml.** Program nevykonáva žiadnu činnosť a dá sa použiť ako **verifikátor** – pokiaľ nevypíše chybu, je XML súbor v poriadku.

```
import java.xml.parsers.*;
import org.x3c.dom.*;

public class VerifikatorJedloDOM {
  private static final String SUBOR = "jedlo.xml";

  public static void main(String[] args) {
    try {
      DocumentBuilderFactory dbf =
                DocumentBuilderFactory.newInstance()
                ;
      dbf.setValidating(false);
      DocumentBuidler builder =
      dbf.newDocumentBuilder();
      builder.setErrorHandler(new
      chybyZisteneParserom());
      // nacitanie dokumentu do pamäti
      Document doc = builder.parse(SUBOR);
      System.out.println(SUBOR + "precitany bez
      chyb");
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Vytváranie objektov DOMu je vďaka JAXP veľmi **podobné** SAXu:

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

Najskôr sa vytvorí **"obálka"** pre univerzálny parser, ktorý ďalej umožňuje nastavenie **validácie** podľa XSD a spracovanie menných priestorov.

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

Validácia sa **nebude** vykonávať.

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

Vytvorenie **vlastného** parseru – opäť nie je zrejmé, aký skutočný parser sa použije.

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

Pretože DOM využíva vnútorné pre parsovanie SAX, je nastavenie reakcie na chyby zhodné so SAX. Trieda **ChybyZisteneParserom** je celkom rovnaká ako pri SAX, rovnaké sú aj reakcie na výnimky a možné rozlíšenie troch typov výnimiek.

V prípade, že meno súboru obsahuje medzery a/alebo akcenty, je v **JDK** nutné namiesto:

```
builder.parse(SUBOR);
```

použiť:

```
builder.parse(new File(SUBOR));
```

XIII. Spracovanie parsovaného XML dokumentu

Pretože po úspešne vykonanom parsovaní sú všetky **dáta** uložené **v pamäti**, existuje v porovnaní so SAX omnoho viac možností, ako ich spracovať.

Práca s obsahom elementu

V hlavnom programe pre výpočet celkovej váhy sa v metóde **main()** zmení iba riadok výpisu:

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

Okrem toho pridávame **metódu**:

```
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.parserDouble(s);
  }
  return celkovaVaha;
}
```

Význam jednotlivých príkazov je nasledovný:

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

Uloží do zoznamu všetky elementy váha.

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

Hodnota element je **null**! To je typická chyba. Je treba získať potomka, ktorým je **Text** a potom až jeho **hodnotu**.

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

Známa metóda **trim()** nás zbavuje prípadných okrajových bielych znakov, ktoré by boli súčasťou hodnoty elementu:

```
<vaha>
  2.5
</vaha>
```

Je veľmi nebezpečné robiť implicitné predpoklady o prvom či poslednom potomkovi, čiže používať metódu **e.getFirstChild().** V XML dokumente môžu byť totiž okrem vyššie spomenutého odriadkovania aj napríklad komentáre:

```
<ovocie cislo = "1">
  <nazov jednotkovaCena = "10">jablka</nazov>
  <vaha><!—celkom dost→2.5/vaha>
</ovocie>
```

Bezpečný spôsob získania hodnoty elementu je napríklad:

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

XIV. Metódy rozhrania Node

Pred zložitejším príkladom je nutné uviesť, aké možnosti nám dávajú jednotlivé uzly. Pretože **Node** je predkom všetkých ďalších uzlov, je možné používať jeho metódy aj v prípade, že vieme, že typ skutočného uzlu je napríklad **Element**.

Metódy pre získanie informácie o uzle

- **short getNodeType()** – typ uzlu-

Väčšinou sa porovnáva s konštantami Node.ATTRIBUTE_NODE, Node.COMMENT_NODE, Node.ELEMENT_NODE, Node.TEXT_NODE.

- **String getNodeName()** – meno uzlu, napríklad „váha“
- **String getNodeValue()** – hodnota uzlu, napríklad „2.5“

Metóda pre pohyb v hierarchii hore

Pokiaľ budete vytvárať programy, v ktorých je častý pohyb po jednotlivých uzloch, je veľmi vhodné zoznámiť sa s možnosťou jazyka **XPath**. Jeho použitie značne pohyb po uzloch uľahčí.

- **Node getParentNode()** – presun na rodičovský uzol

Metódy pre horizontálny pohyb

- **Node getPreviousSibling()** – bezprostredne predchádzajúci súrodenec
- **Node getNextSibling()** – bezprostredne nasledujúci súrodenec

Pokiaľ neexistujú požadovaní súrodenci, tak obe metódy vracajú **null**.

Metódy pre pohyb v hierarchii dolu

- **boolean hasChildNodes()** – existujúci potomkovia?
- **Node getFirstChild()** – prvý potomok
- **Node getLastChild** – posledný potomok

Metódy pre presun na prvého alebo posledného potomka je nutné používať veľmi opatrne. Je treba vždy skúsiť, či potomok, ktorý je vracaný je ten očakávaný. Lepší spôsob je použiť metódu **NodeList getChildNodes(),** ktorá vracia zoznam potomkov. Tento zoznam je možné prezerať pomocou **dvoch metód**:

- **int getLength()** – vracia počet uzlov
- **Node item(int index)** – vracia uzol v poradí, pričom prvý má index 0

Metódy pre prácu s atribútmi

Najskôr je vhodné metódou **boolean hasAttributes()** zistiť, či atribúty existujú. Pokiaľ atribúty existujú, získame ich asociatívne pole metódou **NameNodeMap getAttributes**(). Trieda **NameNodeMap** má metódy:

- **int getLength()** – počet atribútov
- **Node gtNameItem** – vráti uzol atribútu podľa mena
- **Node item** – vráti uzol atribútu podľa poradia

Získavanie hodnôt atribútov nie je priamočiare. Prvé úskalie spočíva v metóde **item**. Pokiaľ pre to nemáme vážny dôvod, potom túto metódu zásadne **nepoužívame**. Poradie atribútov sa totiž v XML môže ľubovoľne **meniť**.

Druhý problém je v tom, že používaná metóda **getNamedItem** vracia nie očakávaný **String** s hodnotou atribútu, ale objekt uzlu, z ktorého je nutné hodnotu získať volaním **getNodeValue().** Typický postup teda je napríklad:

```
String hodnota = namedNodeMap.getNamedItem(
                                "jednotkovaCena").getNodeVa
                                lue();
```

Pokiaľ je uzol pretypovaný na **Element**, je možné použiť **Attr getAttributeNode** a potom **String getValue().**

```
String hodnota = nodeElement.getAttributeNode(
                               "jednotkovaCena").getValue();
```

Vyhľadávanie uzlov a práca s atribútmi

Metóda **getNodePodlaMena()** predstavuje bezpečný spôsob získania očakávaného uzla, ktorá vyrieši problém s neočakávanými uzlami typu **Text**, či **Comment**.

Táto metóda však nevyrieši problém, pri ktorom je komentár vložený medzi známky hodnoty elementu, napríklad:

```
<vaha>2<!–kila-->.5/vaha>
```

Pokiaľ by bolo možné, že sa takýto zápis v spracovávanom XML dokumente vyskytne, odporúča sa použiť jednoduchý spôsob, kedy na samom začiatku spracovávania automaticky odstránime všetky komentáre.

Program **cenaJedloDOM.java** spracováva atribúty a vypočíta **celkovú cenu nákupu**. V metóde **main()** sa zmení iba riadok výpisu:

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

Okrem toho musíme pridať metódu:

```
private static double getCelkovaCena(Document doc)
{
  NodeList nl = doc.getElementsByTagName("ovocie");
  double celkovaCena = 0.0;
  for (int I = 0; i<nl,getLength(); i++) {
     Node e = nl.item(i);
     If (e.getNodeType() == Node.ELEMENT_NODE) {
       celkovaCena += getCenaOvocia(e);
     }
  }
  return celkovaCena;
}
private static int getJednotkovaCena(Node ovocie) {
  Node nazov = getNodePodlaMena(ovocie, "nazov");
  NamedNodeMap nnm = nazov.getAttributes();
  String hodnota = nnm.getNamedItem(
                        "jednotkovaCena").getNodeVal
                        ue();

/*druhy sposob
  Attr a = ((Element)
  nazov).getAttributeNode("jednotkovaCena");
   String hodnota = a.getValue();
*/
   return Integer.parseInt(hodnota);
}
```

```
private static int getCenaOvocia(Node ovocie) {
  double jdnotkovaCena = getJednotkovaCena(ovocie);
  Node vaha = getNodePodlaMena(ovocie, "vaha");
  Node text = getNodePodlaMena(vaha, "#text");
  String hodnota = text.getNodeValue().trim();
  return Double.parseDouble(hodnota) *
  jednotkovaCena;
}

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

Problém vkladaných elementov

Postup použitý v predchádzajúcom príklade v metóde getNodePodlaMena90 sa môže zdať prehnane opatrný, pretože v súbore **jedlo.xml** sú predsa všetky elementy potomkov na jasne definovateľných pozíciách. Takže prečo by ich mal parser meniť?

Parser pozície uzlov nemení, ale vkladá ďalšie textové uzly, vzniknuté odriadkovaním medzi elementmi pôvodného XML dokumentu. Toto môže byť pri spracovaní značný problém, pretože rôzne spôsoby odriadkovania a formátovania medzerami nie je možné úplne zachytiť.

Pokiaľ si vypíšeme potomkov prvého elementu ovocia, dostaneme:

Potomkovia ovocia:

```
#text
nazov
#text
vaha
#text
```

V tomto prípade **#text** predstavuje odriadkovanie a odsadzovanie medzier. Vkladanie týchto textových zabrániť použitím metódy:

```
Void setIgnoringElementContentWhitespace(Boolean
whitespace) triedy DocumentBuilderFactory, tj.
```

Prakticky:

```
dbf.setIgnoringElementContentWhitespace(true);
```

Funkčný program s automatickým odstraňovaním odriadkovavacích uzlov:

```
public class PotomkoviaOvociaDOM {
  private static final String SUBOR =
  "jedlo/dtd.xml";
```

```
public static void main(String[] args) {
  try {
     DocumentBuiderFactory dbf =
                DocumentBuilderFactory.newInstance()
                ;
     dbf.setValidating(false);
     dbf.setIgnoringElementContentWhitespace(true);
     DocumentBuilder builder =
     dbf.newDocumentBuilder();
     builder.setErrorHandler(new
     ChybyZisteneParserom());
     Document doc = builder.parse(SUBOR);
     System.out.println("Potomkovia ovocia: ");
     tlacPotomkovia(doc);
  }
  catch (Exception e) {
    e.printStackTrace();
  }
 }

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

Vypíše:

```
Potomkovia ovocia:
Nazov
Vaha
```

Situácia ale v praxi nie je taká jednoduchá, pretože celý princíp funguje správne iba pokiaľ je možné XML dokument **validovať** oproti **DTD**. Bez podpory **DTD** alebo **XSD** súboru celý systém nefunguje, bez ohľadu na zapnutú alebo vypnutú validáciu.

Použitie **setIgnoringElementContentWhitespace()** je závislé v konečnom dôsledku na validačných súboroch, tj. Nie je všeobecné. Odporúčanie je teda také, že tento spôsob je lepšie nepoužívať. Potom máme na výber dve možnosti riešenia:

- **Uzly** hľadať **„opatrnou“** metódou
- Ihneď po načítaní XML dokumentu najskôr **odstrániť** všetky „odriadkovavacie“ uzly.

XV. Automatické odstránenie komentárov

Funguje iba pre JDK respektíve pre **JAXP**.

Možnosť automatického odstránenia komentára je veľmi podobná ako odstránenie „odriadkovacích“ uzlov. Rozdiel je v tom, že odstránenie komentárov nie je závislé na existencií schémového súboru. Tento spôsob je možné teda použiť vždy.

Odstránenie komentáru zapneme zavolaním metódy **setIgnoringComments()** z triedy **DocumentBuilderFactory**.

Čítaný súbor jedlo-komentar2.xml začína takto:

```
<?xml version = "1.0" encoding = "windows-1250"?>
<jedlo>
  <ovocie cislo = "1">
    <nazov jednotkovaCena = "10">jablka</nazov>
    <!—celkom velka vaha-->
    <vaha>2.5</vaha>
  </ovocie>
```

Zdrojový kód programu **PotomkoviaOvociaKomentareDOM.java** začína takto:

```
public class PotomkoviaOvociaKomentarDOM {
  private static final String SUBOR = "jedlo-
  komentar2.xml":
```

```
public static void main(String[] args) {
  try {
     DocumentBuilderFactory dbf =
                DocumentBuilderFactory.newInstance()
                ;
     dbf.setValidating(false);
     dbf.setIgnoringComments(true);
     DocumentBuilder builder =
     dbf.newDocumetBuilder();
     builder.setErrorHandler(new
     ChybyZisteneParserom());
     Document doc = builder.parse(SUBOR);
     System.out.println("Potomkovia ovocia: ");
     tlacPotomkovia(doc);
  }
  catch (Exception e) {
    e.printStackTrace();
  }
}
```

Pre nastavenie **dbf.setIgnoringComments(false);** program vypíše:

Potomkovia ovocia:

```
#text
Nazov
#text
#comment
#text
Vaha
#text
```

Pre nastavenie **dbf.setIgnoringComments(true);** program vypíše:

Potomkovia ovocia:

```
#text
Nazov
#text
Vaha
Text
```

Prevod infosetov zo zoznamu objektov

Pokiaľ potrebujeme uložiť hodnoty z XML dokumentu do pamäti, je situácia viac-menej bezproblémová. Program uloží všetky hodnoty a atribúty naraz do pamäti a potom ich spracováva, pričom triedy **Ovocie** a **SpracovanieDatVPamati** sú celkom rovnaké , ako v SAX.

V metóde **main()** v súbore **VsetkoVPamatiDOM.java** sa zmení:

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

ArrayList<Ovocie> ar =
UlozeniDoPamati.getZoznam(doc);
SpracovanieDatvPamati.tlacVsetko(ar);
System.out.println("Celkova vaha = "
        + SpracovanieDatVPamati.celkovaVaha(ar));
System.out.println("Celkova cena = "
        + SpracovanieDatVPamati.celkovaCena(ar));
```

Trieda pre ukladanie dát do pamäti má obsah:

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

public class UlozenyDoPamati {
  private static ArrayList<Ovocie> ar = new
  ArrayList<Ovocie>();
```

```
  public static ArrayList<Ovocie>
  getZoznam(Document doc) {
    NodeList nl =
    doc.getElementsByTagName("ovocie");
    for (int i = 0; i<nl.getLength(); i++) {
       Node e = nl.item(i);
       if (e.getNodeType() == Node.ELEMENT_NODE) {
         ar.add(vztvorOvocie)(e));
       }
    }
    return ar;
}

private static Ovocie vztvorOvocie(Node Ovocie) {
  String hodnota =
     ((Element)
     ovocie).getAttributeNode("cislo").getValue();
  int cislo = Integer.parserInt(hodnota);
  int jednotkovaCena = getJednotkovaCena(ovocie);
  String nazov = getNazov(ovocie);
  double vaha = getVahaOvocie(ovocie);
  return new Ovocie(cislo, nazov, jednotkovaCena,
  vaha);
}

private static int getJednotkovaCena(Node ovocie) {
  Node nazov = getNodePodlaMena(Ovocie, "nazov");
  NamedNodeMap nnm = nazov.getAttributes();
  String hodnota =
      nnm.getNamedItem("jednotkovaCena").getNodeVal
      ue();
  return Integer.parseInt(hodnota);
}
```

```
  private static String getNazov(Node ovocie) {
    Node nazov = getNodePodlaMena(ovocie, "nazov");
    Node text = getNodePodlaMena(nazov, "#text");
    return text.getNodeValue();
  }

  private static double getVahaOvocia(Node ovocie) {
    Node vaha = getNodePodlaMena(ovocie, "vaha");
    Node text = getNodePodlaMena(vaha, "#text");
    String hodnota = text.getNodeValue().trim();
    return Double.parseDouble(hodnota);
  }

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

XVI. Prechod stromom dokumentu

V niektorých prípadoch sa môžu hodiť ďalšie spôsoby **prechodu stromom**. Typický je prípad, keď nepoznáme dopredu detailné štruktúry spracovávaného XML dokumentu. Ďalší prípad je, pokiaľ je **hierarchická štruktúra** zložitá a my potrebujeme získať iba niektoré informácie.

Rozhrania s príslušnými metódami sú definované v **DOM2** a patria do balíku **org.w3c.dom.traversal.**

Balík org.w3c.dom.traversal popisuje **dve rozhrania**:

1. **NodeIterator** – pozerá na všetky uzly ako by boli v jednom zozname bez hierarchickej **štruktúry**. Prechod cez tieto uzly je potom lineárne od prvého k poslednému.
2. **TreeWalker** – prechod uzlami je realizovaný hierarchicky.

Obidva spôsoby majú možnosť použiť vlastný filter, ktorým sa môžeme zbaviť nepotrebných dát. **Filter**, ktorý musí implementovať rozhranie **NodeFilter**, je veľmi často využívaný.

Posledné rozhranie tohto balíku je **DocumentTraversal**, pomocou neho sa vytvára objekt **NodeIterator** a **TreeWalker** takto:

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

alebo:

```
Treewalker tw = ((DocumentTraversal) doc).createTreeWalker(…);
```

Obe metódy create...() majú rovnaké **štyri parametre:**

1. **Node root** – koreňový uzol
2. **Int whatToShow** – alé typy uzlov sa budú zobrazovať

- Využívajú sa konštanty z **NodeFilter** napr.:

SHOW_ALL, SHOW_COMMENT, SHOW_ELEMENT, SHOW_TEXT...

- **Konštanty** sa dajú sčítať, napr nastavenie:

NodeFilter,SHOW_COMMENT + NodeFilter.SHOW_TEXT

Konštanta **NodeFilter.SHOW_ATTRIBUTE** má význam iba pokiaľ je koreňovým uzlom **atribútový uzol**. Prakticky sa teda nepoužíva, pretože atribúty nie sú zaradené v dokumentovom strome.

3. **NodeFilter filter** – zjemnenie parametru **whatToShow**, ktorý spôsobí, že z vybraných typov uzlov sa budú dať vybrať iba niektoré.

NodeFilter má iba jednu metódu **public short acceptNode** – tá môže vracať tri rôzne hodnoty:

- **NodeFilter.FILTER_ACCEPT** – tento uzol je povolené ďalej spracovávať.
- **NodeFilter.FILTER_SKIP** – tento uzol sa ďalej nespracováva, ale jeho prípadný potomkovia áno.
- **NodeFilter.FILTER_REJECT** – tento uzol ani jeho potomkovia sa ďalej nespracovávajú.

4. **Boolean entityReferenceExpansion** – povolenie expanzie **entit**. Má zmysel iba pre dokumenty využívajúce DTD. Pre bežné dátovo orientované dokumenty sa tento parameter nastavuje na **false**.

Program **NodeIteratorDOM.java** využíva možnosť **NodeIterator**. Vypíše najskôr názvy všetkých elementov a ich prípadných hodnôt a potom rovnakou metódou názvy a hodnoty elementov názov a váha. Všetky uzly sú v jednom zozname, ktorý prezeráme pomocou metódy **nextNode().** Táto metóda vracia **null**, pokiaľ je už celý zoznam spracovaný.

V prípade, že by sme sa potrebovali v zozname uzlov vrátiť, môžeme využiť metódu **Node previousNode().**

```
import java.io.*;
import java.util.*;
import java.xml.parsers.*;
import org.w3c.dom.*;
import org.w3c.dom.traversal.*;
import org.xml.sax.*;

public class NodeIteratorDOM {
  private static final String SUBOR = "jedlo.xml";

public static void main(String[] args) {
  try {
     DocumentBuilderFactory dbf =
                DocumentBuilderFactory.newInstance()
                ;
     dbf.setValidating(false);
     DocumentBuilder builder =
     dbf.newDocumentBuilder();
     builder.setErrorHandler(new
     chybyZisteneParserom());
     Document doc = builder.parse(SUBOR);
```

```
    // vypis vsetkych elementov a textu
    NodeIterator ni =
          ((DocumentTraversal) doc).createNodeIterator(
            doc.getDocumentElement(),
            NodeFilter.SHOW_ELEMENT +
            NodeFilter.SHOW_TEXT,
            null, false);
    vypisZoznam(ni);
    System.out.println("Filtrovany vstup");
    ni = ((DocumentTraversal) doc).createNodeIterator(
            doc.getDocumentElement(),
            NodeFilter.SHOW_ELEMENT +
            NodeFilter.SHOW_TEXT,
            new Filter(), false);
    vypisZoznam(ni);
  }
    catch (Exception e) {
      e.printStackTrace();
    }
}
    private static void vzpisZoznam(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 Filter implements NodeFilter {
    public short acceptNode(Node n) {
      if (n.getNodeName().equals("ovocie") == true) {
        return NodeFilter.FILTER_SKIP;
      }
      else if (n.getNodeType() == Node.TEXT_NODE
          && n.getNodeValue().trim().length() == 0) {
        // odriadkovanie
        return NodeFilter.FILTER_SKIP;
      }
      return NodeFilter.FILTER_ACCEPT;
  }
}
```

Vypíše:

jedlo:
ovocie:
nazov: jablka
vaha: 2.5
...

Filtrovaný vstup
jedlo: nazov: jablká
vaha: 2.5
nazov: banány
vaha: 2
nazov: grapefruity
vaha: 0.75
nazov: slivky sušené
vaha: 1.8

Rozhranie **TreeWalker** nám z tohto pohľadu dáva možností omnoho viac, pretože udržuje hierarchickú štruktúru uzlov. V nej sa môžeme pohybovať pomocou metód **Node nextNode()** a **Node previousNode()** ako v prípade **NodeIterato**. Tu sa ale priechod uskutočňuje postupne po jednotlivých vetvách stromov.

Ďalšie možnosti pohybu po dokumente sú rovnaké, ako má **Node**, tj. známe **metódy**:

- **Node parentNode()**
- **Node nextSibling()**
- **Node previousSibling()**
- **Node firstChild()**
- **Node lastChild()**

Výhodou **TreeWalker** je, že tieto uzly sú už „prefiltrované“ príslušným **NodeFilter**, takže obsahujú iba to, čo nás zaujíma.

Automatické odstránenie odriadkovaných uzlov

Uzly sa dajú odstrániť využitím **NodeIterator**. Využijeme najskôr nastavenie pri vzniku **NodeIterator** tak, aby boli spracovávané iba textové uzly. Je potrebné nastaviť filtre tak, aby prepustili iba tie uzly, ktoré sú buď prázdne alebo obsahujú **biele znaky**. Potom už stačí vracaný zoznam týchto uzlov iba prejsť a nájdené uzly vymazať.

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

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

Po zavolaní **odstranMedzeryANoveRiadky(doc();** sú všetky odriadkovávacie uzly z DOM odstránené.

Zápis dokumentu

Značnou **výhodou DOM** oproti SAX je možnosť načítaný dokument zapísať na disk. Presnejší výraz je **serializovať**, čo predstavuje **zápis** XML dokumentu **do prúdu**. Pre túto činnosť poskytuje JAXP veľmi účinnú podporu, pretože **DOM strom** je možné serializovať niekoľkými spôsobmi.

V skutočnosti pri serializácií prebieha **transformácia**, ktorá predstavuje ďalšiu silnú zbraň XML. Pre transformáciu je možné využiť jazyk XLS. Pre serializáciu sa používajú triedy JAXP z balíku **java.xml.transform**. Jedná sa o triedy **TransformerFactory** a **Transformer** používané v rovnakom duchu, ako obdobné triedy pri vytváraní SAX či DOM parseru.

Ďalej potrebujeme triedu **DOMSource** z balíku **xml.transform.dom** a triedu **StreamResult** z balíku **javax.xml.transform.stream**. Konkrétny transformačný program, ktorý v JDK vykonáva transformáciu, je **Xalan**. Ďalší často používaný transformačný program je **Saxon**.

Ovplyvnenie transformačnej triedy

Pomocou JAXP je možné ovplyvniť spôsob činnosti triedy **Transformer**. K tomuto účelu sa využíva metóda **SetOutputProperty()** tejto triedy s dvoma formálnymi parametrami. Prvým je **meno** nastavované transformačnej vlastnosti a druhým je **hodnota** tejto vlastnosti.

Skutočným prvým parametrom sú konštanty triedy **javax.xml.transform.OutputKeys**, ktorých je v JDK celkovo 10. Pre naše účely sú významné nasledujúce:

- **OMIT_XML_DECLARATION**

Pokiaľ je druhým skutočným reťazec **„yes“,** potom nebude vo výslednom XL dokumente uvedená jeho hlavička. Súbor teda bude začínať napríklad:

```
<jedlo>
  <ovocie cislo = "1">
```

Toto nastavenie nie je príliš užitočné, pretože našou snahou by malo byť vytvárať **XML dokumenty** vrátane ich hlavičky. Jedným z rozumných použití je vytvorenie dočasného XML dokumentu bez hlavičky.

Druhou možnosťou je reťazec **„no“** a toto nastavenie spôsobí, že výsledný XML dokument bude obsahovať hlavičku. Súbor teda bude začínať napríklad:

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

Implicitné nastavenie tejto vlastnosti je:

```
setOutputProperty(outputKeys.OMIT_DECLARATION,
"no");
```

- **ENCODING**

Druhým skutočným parametrom je reťazec udávajúci požadované kódovanie. Toto kódovanie potom bude použité pre výstupný **XML dokument**.

Implicitné nastavenie tejto vlastnosti je prevzaté z hlavičky vstupného XML dokumentu. Ak tam nebolo uvedené, prípadne pokiaľ **DOM** vznikol inak ako čítaním XML dokumentu je implicitné nastavenie:

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

Varovanie:

Pre JDK 1,6 toto nastavenie funguje pri transformácii, ale bohužiaľ sa neprejaví zápisom do hlavičky vzniknutého XML dokumentu.

- **STANDALONE**

Druhým skutočným parametrom je reťazec „yes“ alebo „no“.

Varovanie: Pre JDK 1,6 toto nastavenie bohužiaľ nefunguje.

- **INDENT**

Druhým skutočným je reťazec **„yes“** alebo **„no“.**

- **METHOD**

Druhým skutočným parametrom je reťazec **„xml“** alebo **„text“.** Pre **„xml“** sa vykoná očakávaná transformácia 1:1.

Nastavenie skutočného parametra na „text“ je veľmi užitočné v prípade, kedy chceme z XML dokumentu získať len hodnoty elementov.

Napríklad pre nastavenie:

```
setOutProperty(OutputKeys.METHOD, “text“);
```

Dostaneme výstupný súbor:

jablká
2,5
Banány
2
Grapefruity
0,75
Slivky sušené
1,7

XVII. Ukážka zápisu do súboru

Pokiaľ načítaný XML dokument nemeníme, prípadne nevytvárame úplne nový dokument, nemá zápis tohto načítaného XML dokumentu do iného dokumentu 1: veľký zmysel.

Poznámka: Jedná sa vlastne o vytvorenie kópie súboru - výpočtovo zložitého a časovo náročného.

Existujú však najmenej dva prípady, kedy sa tento zápis dá zmysluplne využiť. Prvým z nich je prípad, kedy má mať **výstupný XML súbor** iné **kódovanie** (samozrejme vrátane hlavičky XML dokumentu) ako vstupný súbor. Druhým prípadom je **zmena štýlu odsadzovania**.

V tejto časti sa bude jednať o zmenu výstupného kódovania. Program prečíta súbor **jedlo.xml** (v kódovaní Windows-1250) a uloží ho do súboru **jedlo-UTF-8.xml**. Bohužiaľ sa však budú líšiť riešenia pre JDK 1.5 a JDK 1.6.

Príklad:

Ukážka riešení pre JDK 1.5

```
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_5 {
    private static final String SUBOR = "jedlo.xml";
    private static final String VYSTUPNY_SUBOR =
"jedlo-UTF-8.xml";

    public static void main(String[] args) {
        try {
            DocumentBuilderFactory dbf =
                    Document
                BuilderFactory.newInstance();
            dbf.setValidating(false);

            DocumentBuilder builder =
            dbf.newDocumentBuilder();
            Builder.setErrorHandler(new
            ChybyZisteneParserom());
            Document doc = builder.parse(SUBOR);

            TransformerFactory tf =
            TransformerFactory.newInstance();
            Transformer zapisovac =
            tf.newTransformer();
            Zapisovac.setOutputProperty(OutputKeys.ENC
            ODING, "UTF-8");
            Zapisovac.tranform(new DOMSource(doc),
                        New StreamResult(
                        New File(VYSTUPNY_SUBOR)));
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Pre JDK 1.5 je vhodné používať ako výstupné kódovanie **UTF-8**, kde budú znaky zapísané podľa očakávaní, čo pre iné kódovanie nie je splnené.

Pri použití kódovania Windows-1250 dostaneme z originálu:

áčďéeíňóršťúuýž

Výstup používajúci XML zápis pomocou kódových bodov **Unicode** (zapísaných navyše netypicky v desiatkovej sústave) pre niektoré znaky:

áčďéěíňóřšťúůýž

V prípade, že použijeme kódovanie ISO-8859-2 dostaneme rovnaký výsledok (sú to Unicode kódové body):

áčďéěíňóřšťúůýž

Podobne dopadne výsledok pre kódovania **UTF-16LE** a **UTF-16BE**:

á č ď é ěíň óř š ť ó ř š ť ú ů ý ž

Vo všetkých prípadoch je z hľadiska **XML dokumentu** všetko v poriadku.

Príklad:

Ukážka riešení pre JDK 1.6.

V tomto prípade bude tiež ukázané rozsiahlejšie použitie vyššie popísaných **konštánt** z **OutputKeys.**

```
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 SUBOR =
“jedlo.xml“;
    private static final String VYSTUPNY_SUBOR =
“jedlo-UTF-8.xml“;
```

```
public static void main(String[] args) {
    try {
        DocumentBuilderFactory dbf =
                Document
            BuilderFactory.newInstance();
        dbf.setValidating(false);

        DocumentBuilder builder =
        dbf.newDocumentBuilder();
        Builder.setErrorHandler(new
        ChybyZisteneParserom());
        Document doc = builder.parse(SUBOR);

        String kodovanie = "UTF-8";

        TransformerFactory tf =
        TransformerFactory.newInstance();
        Transformer zapisovac =
        tf.newTransformer();
        zapisovac.setOutputProperty
        (OutputKeys.OMIT_XML_DECLARATION,
                                "yes");
        zapisovac.setOutProperty(OutputKexs.MET
        HOD, "xml");
        DOmResult DOMbezHlavicky = new
        DOmResult();
        zapisovac.transform(new DOMSource(doc),
        DOMbezHlavicky);

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

```
                zapisovac.setOutputProperty(OutputKeys.
                ENCODING, kodovanie);
                // nefunguje
                zapisovac.setOutputProperty(OutputKeys.
                STANDALONE, “yes“);
                // nastavenie standalone=“yes“
                Document doc2 = (Document)
                DOMbezHlavicky.getNode();
                doc2.setXmlStandalone(true);

                zapisovac.transform(new DOMSource(doc),
                                    new StreamResult
                                    new
                                    File(VYSTUPNY_SUBOR)));
            }
            catch (Exception e) {
                e.printStackTrace();
            }
        }
}
```

Načítanie XML súboru je rovnaké ako pre JDK 1.5. Potom je prevedená prvá **transformácia**, samozrejme s nastavením:

```
(OutputKeys.OMIT_XML_DECLARATION, “yes“)
```

Tá spôsobí, že **dočasný** DOM **nebude** mať **hlavičku**. Nad týmto DOM sa prevedie **druhá transformácia**, pri ktorej sa zmení výstupné kódovanie a tiež nastaví požiadavka na vytvorenie hlavičky, do ktorej už bude zvolené výstupné kódovanie správne zapísané.

Tým sa ale práca nekončí, pretože by sa v hlavičke použilo **implicitné** standalone="no". To nie je možné prepísať pomocou očakávaného:

```
(OutputKeys.OutputKeys.STANDALONE, "yes")
```

A musí sa preto použiť trik, v ktorom sa nastaví **standalone** pomocou metódy **setxmlStandalone(true);**

Pokiaľ by sme nepoužili tento komplikovaný postup dvojitej transformácie, dostali by sme výstupný XML súbor v požadovanom kódovaní, ale v hlavičke bolo uvedené kódovanie pôvodného vstupného XML dokumentu, co je samozrejme hrubá chyba.

XVIII. Problematika odriadkovania a odsadzovania

Pri spracovaní XML dokumentu **nezáleží** na odriadkovaní a odsadení elementov. Tak napr. nasledujúce dva XML dokumenty nesú tú istú **informáciu**:

```
<jedlo><ovocie cislo="4"><nazov
jednotkovaCena="32">slivky</nazov><vaha>1.8</vaha><
/ovocie></
jedlo>
a
<jedlo>
<ovocie cislo="4">
<nazov jednotkovaCena="32">slivky</nazov>
<vaha>1.8</vaha>
</ovocie>
</jedlo>
```

Samozrejme **druhý** príklad je **prehľadnejší**.

Ak vytvárame **nový** XML dokument, je zaistenie správneho odriadkovania a odsadenia nepríjemná práca. Túto činnosť je možné vykonať **automaticky**, čo je možné zaistiť nastavením **príslušných transformačných vlastností**.

Teoreticky by malo stačiť len **nastavenie**:

```
Zapisovac.setOutputProperty(OutputKeys.INDENT,
"yes");
```

Ktoré zaistí **odriadkovanie elementov** a ich **odsadení**.

Prakticky ale často potrebujeme nastaviť aj „**hĺbku odsadenia“** alebo **počet** vkladaných odsadzovacích **medzier**.

Príklad:

Ukážka riešení pre JDK 1.5

Hĺbku odsadenia (dve medzery) je možné nastaviť volaním:

```
tf.setAttribute(“indent-number“, new Integer(2));
```

Jedná sa o metódu triedy **TransformerFactory** nie Trasformer ako to bolo doteraz.

Druhým problémom je, že toto nastavenie v jednoduchej podobe **nefunguje** správne pre akcentované znaky v UTF-8. Pre **správnu** funkčnosť je potrebné použiť komplikovanejšie nastavenie výstupu.

```
import java.io.*;
import javax.nio.charset.*;
import org.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 OdsadenieVystupuDOM1_5 {
    private static final String SUBOR = “jedlo-
    neodsadene2.xml“;
    private static final String VYSTUPNY_SUBOR =
    “jedlo-odsadene2.xml“;

    public static void main(String[] args) {
        try {
            DocumentBuilderFactory dbf =
            DocumentBuilderFactory.newInstance();
            Dbf.setValidating(false);
            DocumentBuilder builder =
            dbg.newDocumentBuilder();
            Builder.setErrorHandler(new
            ChybyZisteneParserom());
            Document doc = builder.parse(SUBOR);

            TransformerFactory tf =
            TransformerFactory.newInstance();
            Tf.setAttribute(“indent-number“, new
            Integer(2));
            Trnasformer zapisovac =
            tf.newTransformer();
            Zapisovac.setOutputProperty(OutputKeys.
            INDENT, “yes“);
            String kodovanie = “UTF-8“;
            zapisovac.setOutputProperty(“encoding“,
            kodovanie);
```

```
            System.out.println(zapisovac.getOutputP
            roperties());

            OutputStreamWriter osw = new
            OutputStreamWriter(
                            new FileOputputStream(
                            new
                            File(VYSTUPNY_SUBOR)),
                            Charset.forName(kodovan
                            ie));
            zapisovac.transform(new DOMSource(doc),
            new StreamResult(osw));
            osw.close();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Významy jednotlivých príkazov:

```
tf.setAttribute("indent-number", new Integer(2));
```

Zaistí odsadenie ponorených neodsadených elementov o 2 medzery.

Poznámka: V JDK 1.4 to bol príkaz priamo ovplyvňujúci Xalan.

```
zapisovac.setOutputProperty(
    "{http://xml.apache.org/xalan}indent-amount",
"2");
zapisovac.setOutputProperty(OutputKeys.INDENT,"yes"
);
```

Zaistí odriadkovania neodriadkovaných elementov vrátane koreňového.

Komplikovaný príkaz:

```
OutputStreamWriter osw = new OutputStreamWriter(
                    new FileOutputStream(
                    new File(VYSTUPNY_SUBOR));
                    Charset.forName(kodovanie));
```

Zaistí, že výstup bude zapisovaný znakovo do **Writteru** a použité výstupné kódovanie bude skutočne dodržané. Bez tohto príkazu nie je výstupné kódovanie v poriadku.

Príklad:

Ukážka riešení pre JDK 1.6

Hĺbku odsadenia (dve medzery) je možné nastaviť **volaním**:

```
zapisovac.setOutputProperty(http://xml.apache.org/x
alan}indent-amount, "2");
```

Alebo lepšie **prenositeľnejšie**:

```
zapisovac.setOutputProperty(http://xml.apache.org/xslt}indent-amount, "2");
```

čím sa situácia vrátila k riešeniu v JDK 1.4.

Riešenie funkčnej pre JDK 1.5 pomocou:

```
tf.setAttribute("indent-number", new Integer(2));
```

tu nefunguje.

Naopak problém s nesprávnou funkčnosťou pre akceptované znaky v **UTF-8** tu nie je a preto nie je uvedený.

```
import java.io.*;
import org.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 OdsadzovanieVystupuDOM1_6 {
    private static final String SUBOR = "jedlo-neodsadene.xml";
    private static final String VYSTUPNY_SUBOR = "jedlo-odsadene.xml";
```

```
    public static void main(String[] args) {
        try {
            DocumentBuilderFactory dbf =
            DocumentBuilderFactory.newInstance();
            dbf.setValidating(false);
            DocumentBuilder builder =
            dbf.newDocumentBuilder();
            Builder.setErrorHandler(new
            ChybyZisteneParserom());
            Document doc = builder.parse(SUBOR);
            TransformerFactory tf =
            TransformerFactory.newInstance();
            Transformer zapisovac =
            tf.newTransformer();
            zapisovac.setOutputProperty(OutputKeys.IND
            ENT, "yes");
//          zapisovac.setOutputProperty(
//
    "{http://xml.apache.org/xalan}indent-amount",
//                  "2");
            zapisovac.setOutputProperty(
                    "{http://xml.apache.org/xslt}inde
                nt-amount", "2");

            zapisovac.transform(new DOMSource(doc),
                                new StreamResult(
                                new
                                 File(VYSTUPNY_SUBOR)));
        }
        catch (Exception e) {
            e.prinStackTrace();
        }
    }
}
```

Zmena odsadenia XML dokumentu

Ak má **vstupný XML súbor** nevyhovujúce odsadenie (napr. príliš veľké – preddefinované je často 4 medzery), je možné ho vo výstupnom súbore upraviť podľa našich potrieb.

Pri riešení tohto prípadu je nutné si uvedomiť, že nové riadky a odsadenie elementov pri načítaní XML dokumentu predstavujú v DOM **textové uzly**. Tých sa dokážeme zbaviť využitím metód triedy **Document-Traversal.**

Dokument bez týchto uzlov sa potom už **známym** spôsobom zapíše s potrebným počtom odsadených medzier.

Celé riešenie je teda v kombinácii už známych riešení a v prípade záujmu si ich môžete prehliadnuť v súbore **ZmenaOdsadeniaVystupuDOM1_6.java**.

XIX. Modifikácia dokumentu

Významnou výhodou **DOM** oproti **SAX** je možnosť načítaný dokument meniť (zmenšovať, zväčšovať, opravovať).

Poznámka: Dobrou vlastnosťou je, že sa nemusíme starať napr. o správny zápis znakov typu & (tj. &) alebo <(tj. <), alebo to za nás pri zápise vykoná automaticky **transformačná trieda.**

Poznámka: V ďalšom výklade sa obmedzíme len na najčastejšie akcie, tj.zmena hodnoty elementu či atribútu a pridanie atribútu či elementu. Ďalšie akcie typu **„pridanie komentára“** sú analogické a príslušné metódy je možné ľahko dohliadať v dokumentácii k **Java Core API.**

Zmena hodnoty už existujúceho elementu či atrubútu

Všeobecne použiteľnou metódou pre zmeny je metóda triedy **Node**.

```
void setNodeValue(String nodeValue)
```

Ktorú ale treba volať len na správne typy uzlov (tzn. prakticky len typu Text). Je teda v podstate vhodnejšie používať metódy konkrétnych uzlov pre dve najbežnejšie činnosti:

- **Zmena hodnoty elementu** – jedná sa o objekt triedy **Element**, kedy použijeme metódu:

```
Text replacewholeText(String content)
```

- **Zmena hodnoty atribútu** – jedná sa o objekt triedy **Element**, kedy použijeme metódu

```
void setattribute(String name, String value)
```

- **Zmena hodnoty atribútu** – jedná sa o objekt triedy **Attr**, kedy použijeme metódu:

```
void setValue(String value)
```

Odstránenie uzla

Je možné odstrániť ľubovoľný uzol, ale je potrebné si uvedomiť, že túto činnosť musíme vykonať z nadradeného uzla – nie je možné zrušiť „samého seba“.

V prípade uzlov **Element** alebo **Text** musíme byť v rodičovskom uzle element a potom použijeme **metódu**:

```
Node removeChild(Node oldChild)
```

Pre odstránenie atribútu musíme byť v uzle **Element**, ku ktorému odstraňovaný **atribút** patrí a použijeme:

```
Void removeAttribute(String name)
```

Vkladanie nových uzlov

Jedná sa o najťažšie akcie, kedy je najskôr nutné vytvoriť nový uzol jednou z nasledujúcich metód triedy org.w3c.dom.Document:

- **Element createElement(String tagName)**
- **Text createTextNode(string data)**
- **Attr createAttribute(String name)**

Poznámka: Atribúty je lepšie vytvárať na hotových elementoch metódou `void setAttribute(String name, String value)`

Po vytvorení nového uzla je možné s ním ďalej pracovať, tzn. umiestniť ho na požadované miesto v infosete, napríklad pomocou **metód**:

- `Node insertBefore(Node newChild, Node refChild)` – vkladať
- `Node replaceChild(Node newChild, Node oldChild)` – zamieňať
- `Node appendChild(Node newChild)` – pridá uzol ako posledný do zoznamu detí

Poznámka: Už na tejto úrovni práce je možné zaisťovať odsadenie v budúcom výstupnom XML súbore.

Pre a za pridávaný element je možné pridať ešte uzol **Text** s odriadkovaním a odsadením, kedy sa pre odriadkovanie použije len “ \r“ spôsobí nadbytočný zápis textu

Prakticky je to väčšinou zbytočná činnosť, pretože jednotné odsadenie je možné zaistiť automaticky skôr opísanými spôsobmi.

Príklad zmeny XML dokumentu a jeho zápisu

Príklad ukáže najbežnejšie zmeny dokumentu a jeho následný zápis. Všetky zmeny sú dostatočne komentované v metóde **ZmenDokument().**

```
import java.io.*;
import org.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 ZmenaJedloDOM {
    private static final String SUBOR =
    “jedlo.xml“;
    private static final String VYSTUPNY_SUBOR =
    “jedlo-zmena.xml“;

    public static void main(String[] args) {
        try {
            DocumentBuilderFactory dbf =
                DocumentBuilderFactory.newInstance(
                );
             dbf.setValidating(false);
             DocumentBuilder builder =
             dbf.newDocumentBuilder();
             Builder.setErrorHandler(new
             ChybyZisteneParserom());
             Document doc = builder.parse(SUBOR);

             zmenDokument(doc);

             TransformerFactory tf =
             TransformerFactory.newInstance();
             Transformer zapisovac =
             tf.newTransformer();
             zapisovac.setOutputProperty(OutputKeys.
             ENCODING,
             “Windows-1250“);
```

```
            zapisovac.tranform(new DOMSource(doc),
                               new StreamResult(
                               new
                                 File(VYSTUPNY_SUBOR)));
        }
        catch (Exception e) {
            e.printStackTrace();
    }
}

    private static void zmenDokument(Document doc)
{
        Node jedlo = doc.getDocumentElement();
        // odstranenie <ovocie cislo="1">
        NodeList nl =
        doc.getelementsByTagName("ovocie");
        Node ovocie1 = nl.item(0);
        jedlo.removeChild(ovocie1);

        // odstranenie atributu cislo="2"
        Element ovocie2 = (Element) nl.item(0)
        ovocie2.removeAttribute("cislo");

        // zmen jednotkovaCena="25" na "15"
        NodeList nlNazov =
        doc.getElementByTagName("nazov");
        Element nazov = (Element) nlNazov.item(0);
        nazov.setAttribute("jednotkovaCena", "15");
```

```
        // pridanie atributu <vaha
        jednotka="kg">2</vaha>
        NodeList nlVaha =
        doc.getElementsBYTagName("vaha");
        Element vaha = (Element) nlvaha.item(0);
        vaha.setAttribute("jednotka", "kg");

        // pridanie elementu <vzhlad>cerstvy &
        vonavy</vzhlad>
        Element vzhlad =
        doc.createElement("vzhlad");
        Text text = doc.createTextNode("čerstvý &
        voňavý");
        ovocie2.appendChild(vzhlad);
        vzhlad.appendChild(text);

        // odriadkovanie za elementom </vzhlad>
        Text textCRLF = doc.createTextNode("\n ");
        Ovocie2.appendChild(textCRLF);
    }
}
```

Vytvorí XML súbor:

```
<?xml version="1.0" encoding="Windows-1250"
Standalone="no"?><jedlo>

    <ovocie>
        <nazov jednotkovaCena="15">banány</nazov>
        <vaha jednotka="kg">2</vaha>
    <vzhlad>čerstvý &amp; voňavý</vzhlad>
    </ovocie>
    <ovocie cislo="3">
        <nazov
        jednotkovaCena="19">grapefruity</nazov>
        <vaha>0.75</vaha>
    </ovocie>
    <ovocie cislo="4">
        <nazov jednotkovaCena="32">slivky
        sušené</nazov>
        <vaha>1.8</vaha>
    </ovocie>
</jedlo>
```

Poznámka: Tento výstup je pre JDK 1.6.

XX. Vytváranie nového dokumentu

Doteraz **DOM** vždy vždy vznikol **načítaním** XML dokumentu. Nie je to však jediný spôsob jeho vzniku. Ak by sme potrebovali vytvoriť úplne **nový** XML dokument je to tiež možné.

K vytvoreniu úplne nového DOM v pamäti potrebujeme **inštanciu triedy**, ktorá spĺňa rozhranie **org.w3c.dom.DOMImplementation**. Tu môžeme získať použitím príkazov **JAXP**:

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

DocumentBuilder builder = dbf.newDocumentBuilder();

DOMImplementation impl =
builder.getDOMImplementation();
```

Rozhranie **DOMImplementation** potom poskytuje **metódu**:

```
Document createDocument(String namespaceURI, String
rotQName,

DocumentType doctype)
```

Kde skutočné parametre **namespaceURI** a **doctype** majú väčšinou hodnotu **null.**

Klonovanie uzlov

Ak vytvárame **XML dokument** , v ktorom sa uzly opakujú len so zmenenými hodnotami atribútov či elementov, môže byť vhodné pripraviť si jeden vzorkový uzol a ostatné uzly potom vytvárať klonovaním vzorového uzla.

Práca so zmenou hodnôt a atribútov je síce stále veľká, ale v prípade zložitejších uzlov bude zrejme menší, ako opätovné kompletné vytváranie uzla.

Pre vlastné klonovanie použijeme metódu z triedy **Node**:

```
Node cloneNode(boolean hlbokaKopia)
```

Ak má skutočný parameter hlbokaKopia hodnotu **true**, vytvorí sa kópia uzlu so všetkými prípadnými potomkami, co väčšinou chceme.

Príklad:

Program vytvorí **jedlo-nove.xml** s dvoma tými istými elementmi **<ovocie>** pričom ten druhý vznikol klonovaním. V príklade je tak vidieť, že je automaticky zaistené odriadkovanie a odsadenie.

```
import java.io.*;
import java.nio.charset.*;
import org.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 NoveJidloDOM {
    private static final String VYSTUPNY_SUBOR =
"jedlo-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(VYSTUPNY
                               SUBOR)));
        }
        catch (Exception e) {
        e.printStackTrace();
    }
}
```

```
    private static Document vytvorDocument() throws
    Exception {
        DocumentBuildrFactory dbf =
                DocumentBuilderFactory.newInstance();
        dbf.setValidating(false);
        DocumentBuilder builder =
            dbf.newDocumentBuilder();
        DOMImplementation impl =
            builder.getDOMImplemantation();
        Document doc = impl.createDocument(null,
            "jedlo", null);
        Node jedlo = doc.getDocumentElement();
        Element ovocie = doc.createElement("ovocie");
        ovocie.setAttribute("cislo", "10");
        Element nazov = doc.createElement("nazov");
        Nazov.setAttribute("jednotkovCena", "30");
        Text textNazov = doc.createTextNode("slivky");
        nazov.appendChild(textNazov);
        Element vaha = doc.createElement("vaha");
        Text textVaha = doc.createTextMpde("1,5");
        vaha.appendChild(textVaha);
        ovocie.appenChild(nazov);
        ovocie.appendChild(vaha);
        Element ovocieklon = (Element)
ovocie.cloneNode(true);
        jedlo.appendChild(ovocie);
        jedlo.appendChild(ovocieklon);
        return.doc;
    }
}
```

Vytvorí pre JDK 1.6:

```
<?xml version="1,0" encoding="Windows-1250"
standalone="no"?>
<jedlo>
    <ovocie cislo="10">
        <nazov jednotkovaCena="30">slivky</nazov>
        <vaha>1.5</vaha>
    </ovocie>
        <ovocie cislo="10">
        <nazov jednotkovaCena="30">slivky</nazov>
<vaha>1.5</vaha>
    </ovocie>
</jedlo>
```

XXI. Validácia novovytvoreného alebo meneného dokumentu

Pokiaľ XML dokument v pamäti **vytvárame** alebo **programovo meníme,** je zrejmé, že môžeme spraviť **chybu**.

Ak máme k dispozícii príslušný **schémový súbor**, je v takýchto prípadoch užitočné si výsledok zmien programovo zvalidovať ešte pred zapísaním do XML súboru.

Druhou možnosťou je zapísať výsledok do XML súboru a ten **zvalidovať externe**. Pre validáciu sa používajú triedy z balíka **javax.xml.validation**. Objekty tried z tohto balíka sa vytvárajú podobným spôsobom cez rozhranie **JAXP** ako to bolo pri **SAX** a **DOM**.

Pomocou **SchemaFactory** a konštanty **javax.xml.XMLConstants.w3c_XML_SCHEMA_NS_URI** sa stanoví, že sa bude pracovať s XSD schémovým súborom. Pomocou takto vytvoreného objektu a príslušného schémového súboru sa vytvorí objekt triedy **Schema**, a pre túto schému sa potom vytvorí konkrétny validátor ako objekt triedy **Validator**.

Ten potom použije metódu **Validate(),** jej parametrom môže byť niekoľko rôznych zdrojov. V našom prípade je to objekt triedy **DOMSource**, ale môže to byť napr. i objekt triedy **StreamSource** alebo **JAXSource**.

Pokiaľ je validácia neúspešná, vypíše sa navyše rozsiahle **chybové hlásenie**, ktoré obsahuje výpis množstva výnimiek. Ak sa chceme výpisu výnimiek zbaviť, použijeme triedu **ValidaciaChybyZisteneParserom**.

Bohužiaľ pri validácii došlo k rozdielu medzi JDK 1.5 a JDK 1.6, preto budú uvedené nasledovné príklady oddelene.

Príklad:

V prípade JDK 1.5 je možné validáciu použiť podľa očakávania.

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

public class ValidaciaNoveJedloDOM1_5 {
    private static final String VALIDACNY_SUBOR =
“jedlo.xsd“;

    public static void main(String[] args) {
        try {
            Document zdroj = vytvorDocument();
            SchemaFactory sf =
            SchemaFactory.newInstance(
                    XMLConstants.w3C_XML_SCHEMA_NS_
                URI);
            validator val = sch.newValidator();
            val.setErrorHandler(new
            validaciaChybyZisteneParserom());
            val.validate(new DOMSource(zdroj));
            System.out.println(“Validacia prebehla
            uspesne“);
        }
```

```
        catch (SAXExpection e) {
            System.out.println("Chyba validacie:");
            // orezanie vypisu
            String s = e.getMessage();
            Int i =
            s.indexOf(ValidaciaChybyZisteneParserom
            .KONEC);
            if (i > 0) {
                s = s.substring(0, i);
            }
            System.out.println(s);
        }
        catch (Exception e) {
            e.printStachTrace();
        }
    }
    private static Document vytvorDocument() throws
Exception {
    ...
```

V prípade **bezchybného** DOM vypíše:

```
Validacia prebehla uspesne.
```

V prípade výskytu chyby(nezmyselná hodnota "ABC10" atribútu číslo) vypíše:

```
Chyba validacie:
Chyba: System ID: null
Riadok: -1 stĺpec: -1
cvc-datatype-valid.1.2.1.: ´ABC10´ is not a valid
value for ´integer´.
```

Kde sa podivné (-1) informácie o riadku a stĺpci vyskytujú, pretože **nie je validovaný** súbor (s jasnou informáciou o príslušnom riadku či stĺpci), ale **DOM** objekt v pamäti.

Validácia v JDK 1.6

Ak sa pokúsime spustiť predchádzajúci program v JDK 1.6, dostaneme nasledujúci **výpis**, pričom nezáleží na tom, či je DOM v poriadku alebo nie.

```
Chyba validacie:
Chyba: System ID: null
Riadok: -1 stĺpec: -1
cvc-complex-type.2.4.a: Invalid content was found
starting with element ´ovocie´. One of ´{ovocie}´
is expected.
```

Tento chybný výpis je spôsobený tým, že validácia v JDK 1.6 vyžaduje použitie menných priestorov. Tie samozrejme v mnohých jednoduchých XML dokumentoch používať nemusíme a nechceme.

Poznámka: Druhým efektom týchto príkladov bude ukážka, ako je možné validátorovi odovzdať iný zdroj, ako doposiaľ používaný **DOMSource.**

Príklad:

Validácia XML súboru. V tomto príklade budeme programovo validovať **XML súbor** načítaný pomocou **DOM**.

Celé riešenie spočíva v tom, ako sa nastaví objekt triedy **DocumentBuilderFactory,** konkrétne v použití metódy **setNamespaceAware().** Ak bude jej skutočný parameter **false** (čo je tiež implicitné nastavenie tohto objektu), dostaneme podobne „nezmyselné“ chybné hlásenie ako v predchádzajúcom prípade:

```
Chyba validacie:
Chyba: System ID: null
Riadok: -1 stĺpec: -1
cvc-elt.1: Cannot find the declaration of element
´jedlo´.
```

Pre správny priebeh validácie je nutné použiť volanie:

setNamespaceAware();

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

```
public class ValidaciaNoveJedloDOM1_6 {
    private static final String SUBOR =
"jedlo.xml";
//      private static final String SUBOR = "jedlo-
chyba-validacia.xml";
    private static final String VALIDACNY_SUBOR =
"jedlo.xsd";

    public static void main(String[] args) {
        try {
            DocumentBuilderFactory dbf =
                    DocumentBuilderFactory.newInsta
                        nce();
            dbf.setNamespaceAware(false);
            DocumentBuilder builder =
        dbf.newDocumentBuilder();
            Document zdroj = builder.parse(SUBOR);
            SchemaFactory sf =
        SchemaFactory.newInstance(
                        XMLConstants.w3C_XML_SCHEMA
                    _NS_URI);
            Schema sch = sf.newSchema(new
            File(VALIDACNY_SUBOR));
            validator val = sch.newValidator();
            val.setErrorHandler(new
            validaciaChybyZisteneParserom());
            val.validate(new DOMSource(zdroj));
            System.out.println("Validacia prebehla
            uspesne");
        }
```

```
        catch (SAXExpection e) {
            System.out.println("Chyba validacie:");

            String s = e.getMessage();
            Int i =
            s.indexOf(ValidaciaChybyZisteneParserom
            .KONEC);
            if (i > 0) {
                s = s.substring(0, i);
            }
            System.out.println(s);
        }
        catch (Exception e) {
            e.printStachTrace();
        }
    }
}
```

V prípade bezchybného DOM vypíše:

```
Validacia prebehla uspesne.
```

V prípade výskytu **chyby** (nezmyselná hodnota "ABC10" atribútu číslo v súbore jedlo-chyba-validacia.xml) vypíše:

```
Chyba validacie:
Chyba: System ID: null
Riadok: -1 stĺpec: -1
cvc-datatype-valid.1.2.1.: ´ABC1´ is not a valid
value for ´integer´.
```

Príklad:

V prípade validovania novovytvoreného DOM by sa mohlo zdať, že postup s nastavením SetNamespaceAware(true); je možné použiť analogicky a to v metóde **vytvorDocument().** Bolo by to elegantné riešenie, ale nepodarilo sa to overiť.

Preto bolo nutné použiť náhradné riešenie a to v metóde **konverzia()** transformovať už známym spôsobom DOM na objekt typu **Reader** (v skutočnosti je to **XML dokument** zapísaný v jednom reťazci). Potom už stačí len upraviť skutočný parameter metódy **validate()** a validácia prebehne podľa očakávania.

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

public class ValidaciaNoveJedloDOM1_6 {
    private static final String VALIDACNY_SUBOR =
“jedlo.xsd“;
```

```
    public static void main(String[] args) throws
Exception {
        try {
            Document zdroj = vytvorDocument();
            Reader r = konverzia(zdroj);

            SchemaFactory sf =
        SchemaFactory.newInstance(
                        XMLConstants.w3C_XML_SCHEMA
                    _NS_URI);
            Schema sch = sf.newSchema(new
            File(VALIDACNY_SUBOR));
            validator val = sch.newValidator();
            val.setErrorHandler(new
            validaciaChybyNajdeneParserom());
            val.validate(new DOMSource(zdroj));
            System.out.println("Validacia prebehla
            uspesne");
        }
        catch (SAXExpection e) {
            System.out.println("Chyba validacie:");

            String s = e.getMessage();
            int i =
            s.indexOf(ValidaciaChybyNajdeneParserom
            .KONIEC);
            if (i > 0) {
                s = s.substring(0, i);
            }
            System.out.println(s);
        }
```

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

    private static Reader konverzia(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.printStachTrace();
    }
    return null;
}

private static Document vytvorDocument() throws
Exception {
...
```

V prípade bezchybného DOM vypíše:

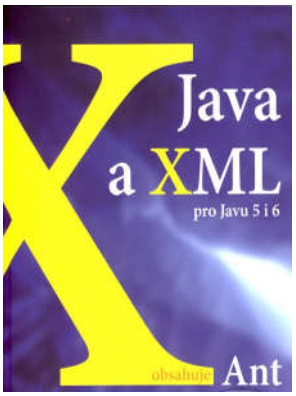
```
Validacia prebehla uspesne.
```

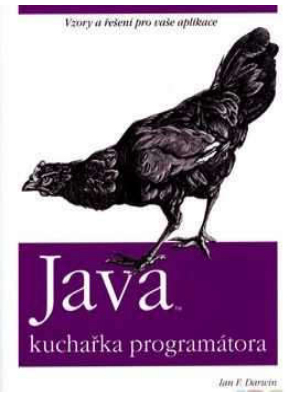
V prípade výskytu **chyby** (nezmyselná hodnota “ABC10“ atributu cislo) vypíše:

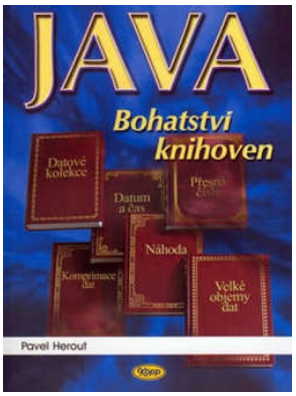
```
Chyba validacie:
Chyba: System ID: null
Riadok: -1 stĺpec: -1
cvc-datatype-valid.1.2.1.: ´ABC10´ is not a valid
value for ´integer´.
```

XXII. Odporúčaná literatúra a zdroje

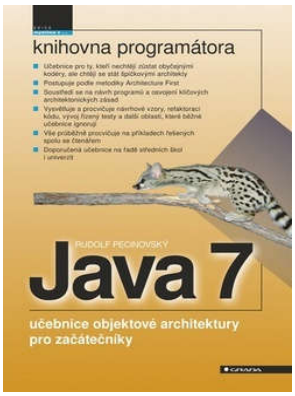
1. **Java a XML -** Pavel Herout
2. **Java kuchařka programátora -** Ian F.Darwin
3. **Java Bohatství knihoven -** Pavel Herout
4. **Java – grafické užívatelské přostredí a čestina -** Pavel Herout





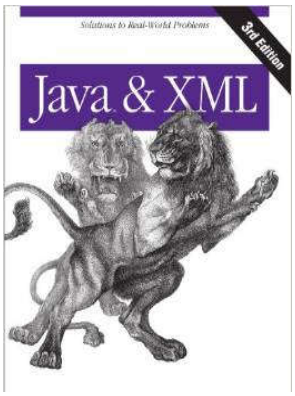


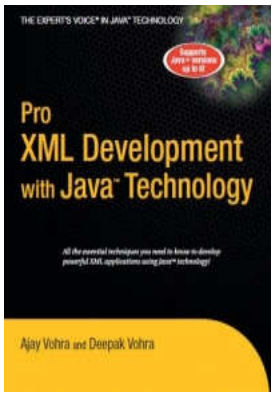
5. **Mistrovství Java -** Herbert Schildt
6. **Java 7 -** Rudolf Pecinovský
7. **1001 tipů a triků pro jazyk Java -** Bogdan Kiszka
8. **Učebnice jazyka Java 5.v. -** Pavel Herout

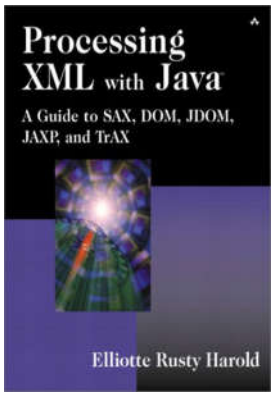


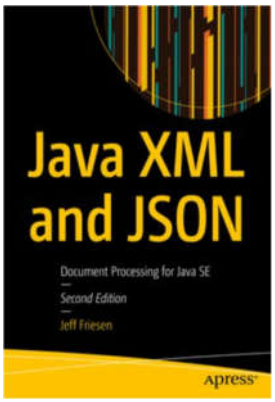
Zahraničná literatúra

1. **Java and XML –** Brett McLaughlin
2. **Pro XML Development with Java Technology –** Ajay Vohra
3. **Processing XML with Java -** E.Rusty Harrold
4. **Java XML and JSON -** Jeff Friesen









5. **Optimizing Java -** Benjamin J Evans
6. **Effective Java -** Joshua Bloch
7. **Java Performance -** Scott Oaks
8. **Head First Java -** Kathy Sierra, Bert Bates