XML Parsing and Regular Expressions

Topics

- XML Parsing
  - DOM
  - SAX
- Regular Expressions

DOM

- Document Object Model (DOM)
- W3C recommendation
- Supports parsing and manipulation of XML documents
- Node/tree oriented interface

DOM Classes

- **The DOM package** `org.w3c.dom`
  - `Document`
    - **Root of document tree**
    - **Provides methods for creating document elements**
  - `Element`
    - **Represents an element, may have attributes**
  - `Node`
    - **Underlying abstraction for DOM**
  - `NodeList`
    - **Ordered collection of nodes**
  - `Text`
    - **Text node, represents the text between tags**

Helper Classes

- The package `javax.xml.parsers`
  - `DocumentBuilderFactory`
    - Provides classes for parsing XML documents or creating DOM object trees
  - `DocumentBuilder`
    - Provides methods for parsing XML documents or creating DOM object trees
- DOM uses SAX classes for parsing

DOM Construction Example

```
private Document createDocument()
{
   Document doc = null;
   try
   {
      DocumentBuilderFactory dbf;
      dbf = DocumentBuilderFactory.newInstance();
      DocumentBuilder db = dbf.newDocumentBuilder();
      doc = db.newDocument();
      Element root = doc.createElement( "presidents" );
      doc.appendChild( root );

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

   return doc;
}
```

DOM Construction Example

```
 private void addPresident( Document doc, int number, String name,
                            int enteredOffice, int exitedOffice,
                            String comment )
{
   Element root = doc.getDocumentElement();
   Element president = doc.createElement( "president" );
   president.setAttribute( "number", ""+number );
   president.setAttribute( "name", name );
   president.setAttribute( "enteredOffice", ""+enteredOffice);
   president.setAttribute( "exitedOffice", ""+exitedOffice );
   Element commentElement = doc.createElement( "comment" );
   Text txt = doc.createTextNode( comment );

   // build tree
   commentElement.appendChild( txt );
   president.appendChild( commentElement );
   root.appendChild( president );
}
```

DOM Parsing Example

```
public static void main( String[] args ) {
   try {
      FileReader reader = new FileReader( args[0] );
      DocumentBuilderFactory dbf;
      dbf = DocumentBuilderFactory.newInstance();
      dbf.setValidating(true); // requires a dtd
      dbf.setIgnoringComments(true);
      dbf.setIgnoringElementContentWhitespace(true);
      dbf.setCoalescing(true);
      DocumentBuilder db = dbf.newDocumentBuilder();
      db.setErrorHandler( new DefaultHandler() );
      InputSource source = new InputSource( reader );
      Document doc = db.parse( source );
      DomPresidentsParser processor = new DomPresidentsParser();
      processor.processDoc( doc );
   }
   catch( Exception e ){
      e.printStackTrace();
   }
}
```

DOM Parsing Example

```
 private void processDoc( Document doc )
 {
    NodeList nodes = doc.getElementsByTagName( "presidents" );
    if( nodes.getLength() == 1 );
    {
       System.out.println("Got a list of presidents (potentially)");
       Element root = (Element)nodes.item(0);
       nodes = root.getElementsByTagName( "president" );
       int presCount = nodes.getLength();
       if( presCount >= 1 )
       {
          System.out.println( "There are presidents in the list" );
          for( int i = 0; i < presCount; i++ )
          {
             Element pres = (Element)nodes.item(i);
             String number = pres.getAttribute( "number" );
             String name = pres.getAttribute( "name" );
             String enteredOffice =
pres.getAttribute("enteredOffice");
             String exitedOffice =
pres.getAttribute( "exitedOffice" );
```

DOM Parsing Example

```
             NodeList commentNodes;
             commentNodes = pres.getElementsByTagName( "comment" );
             String comment = "";
             if( commentNodes.getLength() == 1 )
             {
              // get the text node contained within the comment
element
                Node commentNode = commentNodes.item(0);
                comment = commentNode.getFirstChild().getNodeValue();
             }
             System.out.println( "President (" + number + "): "+
name);
             System.out.println( "  Entered Office: " +
enteredOffice);
             System.out.println( "  Exited Office: " + exitedOffice );
             System.out.println( "  Comment: " + comment );
          }
       }
    }
 }
```

SAX

- Simple API for XML (SAX)
- “de facto” standard for XML parsing
- Supports parsing of XML Documents
- Event oriented

SAX Classes

- The SAX packages `org.xml.sax`
  - `Attribute`
    - Represents an elements set of attributes
  - `ContentHandler`
    - Event handler for the parser
  - `InputSource`
    - Input source for XML entities
  - `SAXException`
  - `SAXParseException`
  - `XMLReader`
    - Interface for parser

SAX Parsing Example

```
public class SaxPresidentsParser implements ContentHandler {
   private String tmpValue;

   public static void main( String[] args ) {
      try {
         FileReader reader = new FileReader( args[0] );
         InputSource source = new InputSource( reader );
         SAXParserFactory spf = SAXParserFactory.newInstance();
         spf.setValidating( true ); // requires a dtd
         SAXParser sp = spf.newSAXParser();

         XMLReader parser = sp.getXMLReader();
         parser.setContentHandler( new SaxPresidentsParser() );
         parser.setErrorHandler( new DefaultHandler() );
         parser.parse( source );
      }
      catch( Exception e ) {
         e.printStackTrace();
      }
   }
```

SAX Parsing Example

```
   public void startElement( String namespace, String localName,
                             String tag, Attributes attrs )
   throws SAXException {
      if( "presidents".equals(tag) ) {
         System.out.println("Got a list of presidents
(potentially)" );
      }
      else if( "president".equals( tag ) )
      {
         String number = attrs.getValue( "number" );
         String name = attrs.getValue( "name" );
         String enteredOffice = attrs.getValue( "enteredOffice" );
         String exitedOffice = attrs.getValue( "exitedOffice" );
         System.out.println( "President (" + number + "): " + name );
         System.out.println( "  Entered Office: " + enteredOffice );
         System.out.println( "  Exited Office: " + exitedOffice );
      }
      else if( "comment".equals( tag ) ) {
         tmpValue = "";
      }
   }
```

SAX Parsing Example

```
   public void endElement(String namespace,String localName,String
tag)
   throws SAXException {
      //System.out.println( "endElement()" );
      if( "comment".equals( tag ) )
      {
         System.out.println( "  Comment: " + tmpValue );
      }
      tmpValue = null;
   }
```

SAX Parsing Example

```
/**
 * Accumulates characters from the tag body, these will be used
 * later to construct the value for the value tag.
 *
 * @param buf buffer of characters
 * @param offset start offset into buffer
 * @param len length of buffer
 */
public void characters (char buf [], int offset, int len)
throws SAXException
{
   //System.out.println( "characters()" );
   if( tmpValue == null )
      tmpValue = new String(buf, offset, len);
   else
      tmpValue += new String(buf, offset, len);
}
```

Break

Regular Expressions

- What are regular expressions?
  - A specification of textual pattern for matching
    - Expression syntax is not Java
    - Exact syntax varies from implementation to implementation
  - Originally used in UNIX tools
- The simplest expressions are string literals

Characters

c	The literal character
\u*nnnn*	Character with the given hex value
\x*nn*	Character with the given hex value
\t	Tab
\n	Newline
\r	Return
\f	Form feed
\a	Alert
\e	Excape character

Boundary

^	Beginning of a line
$	End of a line
\b	Word boundary
\B	Non-word boundary
\A	Beginning of input
\G	End of previous match
\z	End of input
\Z	End of input, except for final terminator

Miscellaneous

- **Set Operations**

XY	*X* followed by *Y*
`X|Y`	*X* or *Y*

- **Escapes**

`\`*c*	Escape character
`\Q...\E`	Verbatim

- **Groups**

`(`*X*`)`	Capture the string in as a group
`\n`	Back reference, match of group *n*

Quantifiers (Greedy)

- Read entire input and work backward looking for match

X?	*X*, Once or not at all
*X**	*X*, Any number of occurrences
X+	*X*, One or more occurrences
X`{n}`	*X*, Exactly `n` occurrences
X`{n,}`	*X*, At least `n` occurrences
X`{n,m}`	*X*, At least `n` but not more than `m` occurrences

Quantifiers (Reluctant)

- **Begin at beginning of input and read until a match is found**

X`??`	**_X_, Once or not at all**
X`*?`	**_X_, Any number of occurrences**
X`+?`	**_X_, One or more occurrences**
X`{n}?`	**_X_, Exactly `n` occurrences**
X`{n,}?`	**_X_, At least `n` occurrences**
X`{n,m}?`	**_X_, At least `n` but not more than `m` occurrences**

Quantifiers (Possessive)

- Require entire input to match

X?+	*X*, Once or not at all
*X**+	*X*, Any number of occurrences
X++	*X*, One or more occurrences
X`{n}+`	*X*, Exactly `n` occurrences
X`{n,}+`	*X*, At least `n` occurrences
X`{n,m}+`	*X*, At least `n` but not more than `m` occurrences

Character Classes

[Character class, a set of character alternatives, enclosed in brackets

- Used to specify ranges

```
[a-z]
```

^ As first character denotes complement

```
[^a-z]
```

Character Class Operations

- Unions
  - Use nested character classes

    `[a-c[A-C]]`, matches `a`, `b`, `c`, `A`, `B` or `C`
- Intersections
  - Use nested classes and `&&` intersection operator

    `[a-m&&[k-z]]`, matches `k`, `l`, or `m`
- Subtraction
  - Negate an intersected class

    `[a-i&&[^d-f]]`, matches `a`, `b`, `c`, `g`, `h`, or `i`

Predefined Character Classes

`.`	**Any character except line termination**
`\d`	**A digit** `[0-9]`
`\D`	**A non-digit** `[^0-9]`
`\s`	**Whitespace** `[ \t\n\r\f\x0B]`
`\S`	**Non-whitespace** `[^ \t\n\r\f\x0B]`
`\w`	**Word character** `[A-Za-z0-9_]`
`\W`	**Non-word character** `[^A-Za-z0-9_]`
`\p{`*name*`}`	**The named class**
`\P{`*name*`}`	**The complement of the named class**

Named Character Classes

`[Lower]`	**ASCII lowercase** `[a-z]`
`[Upper]`	**ASCII uppercase** `[A-Z]`
`[Alpha]`	**ASCII alphabetic** `[A-Za-z]`
`[Digit]`	**ASCII digit** `[0-9]`
`[Alnum]`	**ASCII alphabetic or digit** `[A-Za-z0-9]`
`[XDigit]`	**Hex digits** `[0-9A-Fa-f]`
`[ASCII]`	**All ASCII alphabetic** `[\x00-\x7F]`
`[Blank]`	**Space or tab** `[ \t]`
`[Space]`	**Whitespace** `[ \t\n\r\f\x0B]`

Using Regular Expressions

- A dedicated package `java.util.regex`
- Just two classes
  - `Pattern`
    - Compiles the regular expression string
    - Serves as a factory for `Matcher` objects
  - `Matcher`
    - Performs matching operations against its `Pattern`

Pattern Class

- **No accessible constructor**
- **Static methods**

```
Pattern compile(String regex)
Pattern compile(String regex, int flags)
boolean matches(String regex, CharSequence input)
```

- **Instance methods**

```
int flags()
Matcher matcher(CharSequence input)
String pattern()
String[] split(CharSequence input)
String[] split(CharSequence input,int limit)
```

Matcher Class

- **No accessible constructor**
- **Finding matches**

```
boolean find()
boolean find(int)
boolean matches()
```

- **Operating on matches**

```
int start()
int start(int group)
int end()
int end(int group)
String group()
String group(int group)
int groupCount()
```

Matcher Class

- **Manipulating input**

```
String replaceAll(String)
String replaceFirst(String)
```

Example

```
Pattern pat = Pattern.compile("foo");
Matcher matcher = pat.matcher("foobar");
if (matcher.find()) {
   System.out.println( "'" + matcher.group() + "' @ "
                          + matcher.start() + "..”
                          + matcher.end() );
} else {
   System.out.println("No match found.”);
}
// [output] 'foo' @ 0..3
```

Example

```
pat = Pattern.compile("(bar)(foo)");
matcher = pat.matcher("foobarfoobar");
if (matcher.find()) {
   System.out.println( "'" + matcher.group() + "' @ "
                     + matcher.start() + ".." + matcher.end() );
   System.out.println( "   '" + matcher.group(1) + "' @ "
                     + matcher.start(1) + ".." + matcher.end(1) );
   System.out.println( "   '" + matcher.group(2) + "' @ "
                     + matcher.start(2) + ".." + matcher.end(2) );
} else {
   System.out.println("No match found.");
}
// [output] 'barfoo' @ 3..9
// [output]    'bar' @ 3..6
// [output]    'foo' @ 6..9
```

Example

```
pat = Pattern.compile("([bc]ar)(foo)\\1");
matcher = pat.matcher("carfoobarfoobarfoocar");
if( matcher.find() ) {
   System.out.println( "'" + matcher.group() + "' @ "
                            + matcher.start() + ".." + matcher.end() );
   System.out.println( "   '" + matcher.group(1) + "' @ "
                            + matcher.start(1) + ".." + matcher.end(1) );
   System.out.println( "   '" + matcher.group(2) + "' @ "
                            + matcher.start(2) + ".." + matcher.end(2) );
} else {
   System.out.println("No match found.");
}
// [output] 'barfoobar' @ 6..15
// [output]    'bar' @ 6..9
// [output]    'foo' @ 9..12
```

Options

- Specified when compiling expression or embedding flag in expression
  - `CASE_INSENSITIVE`
    - Case insensitive processing, US-ASCII characters
    - Embedded flag expression `(?i)`
  - `COMMENTS`
    - Permits white space and comments starting with # continuing to end of line
    - Embedded flag expression `(?x)`
  - `DOTALL`
    - The `.` expression matches all characters (including line termination)
    - Embedded flag expression `(?s)`

Options

- `MULTILINE`
  - The `^` and `$` match lines only, not entire input
  - Embedded flag expression `(?m)`
- `UNICODE_CASE`
  - In combination with `CASE_INSENSITIVE`, use Unicode letter case for matching
  - Embedded flag expression `(?u)`
- `UNIX_LINES`
  - Only recognizes `'\n'` as line terminator
  - Embedded flag expression `(?d)`
- `CANON_EQ`
  - Accepts canonical equivalence of Unicode characters

Pattern Errors

- **The** `PatternSyntaxException` **class is thrown when syntax errors a encountered**

```
String getDescription()
int getIndex()
String getPattern()
String getMessage()
```

  - **A string containing all three**

`String` Support

- **The `String` class has a few methods for processing regular expressions**

```
boolean matches( String regex )
String[] split(String regex, int limit)
String[] split( String regex )
```