Application Development with XML and Java

Lecture 3

Introduction to SAX, DOM, JAXP

SAX

- SAX is unique among XML APIs in that it models the parser rather than the document.
  - The parser is represented as an instance of the XMLReader interface.
  - The specific class that implements this interface varies from parser to parser.
  - Most of the time you only access it through the common methods of the XMLReader interface.
- A parser reads a document from beginning to end. As it does so it encounters start-tags, end-tags, text, comments, processing instructions, and more. In SAX, the parser tells the client application what it sees as it sees it by invoking methods in a ContentHandler object. ContentHandler is an interface the client application implements to receive notification of document content.

ContentHandler

- The client application will instantiate a client-specific instance of the ContentHandler interface and register it with the XMLReader that's going to parse the document. As the reader reads the document, it calls back to the methods in the registered ContentHandler object.

- The second example is a simple SAX program that communicates with the XML-RPC service It sends the request document using basic output stream techniques and then receives the response through SAX.

SAX Example

- Since SAX is a read-only API, we are using the same code as before to write the request sent to the server.
- The code for reading the response, however, is different. Rather than reading directly from the stream, SAX bundles the InputStream in an InputSource, a generic wrapper for all the different things an XML document might be stored in— InputStream, Reader, URL, File, etc. This InputSource object is then passed to the parse() method of an XMLReader.
- Several exceptions can be thrown at various points in this process. For instance, a IOException will be thrown if the socket connecting the client to the server is broken. A SAXException will be thrown if the org.apache.xerces.parsers.SAXParser class can't be found somewhere in the class path.
- There's no code in this class to actually find the double response and print it on the console. Yet, when run it produces the expected response:

```
C:\XMLJAVA>java FibonacciSAXClient 42
267914296
```

DOM

- The Document Object Model, DOM, is a tree-based API
- DOM programs start off similarly to SAX programs, by having a parser object read an XML document from an input stream or other source.
- SAX parser returns the document broken up into a series of small pieces, DOM method returns an entire Document object that contains everything in the original XML document.
- Information is read from the document by invoking methods on this Document object or on the other objects it contains. This makes DOM much more convenient when random access to widely separated parts of the original document is required.
- It is memory intensive compared to SAX. It is not as well suited to streaming applications
- A second advantage to DOM is that it is a read-write API. Whereas SAX can only parse existing XML documents, DOM can also create them.

DOM Example

- Example 2 is a DOM-based program for connecting to the Fibonacci XML-RPC servlet. The request is formed as a new DOM document. The response is read as a parsed DOM document.
- In DOM the request document is built as a tree. Everything in the document is a node in this tree including text nodes, comments, processing instructions and more.
- Once the Document object has been created and populated, it needs to be serialized onto the URLConnection's output stream.
- When the server receives and parses the request, it calculates and transmits its response as an XML document.
- This document must be parsed to extract the single string we actually want. DOM includes a number of methods and classes to extract particular parts of a document without necessarily walking down the entire tree.

DOM Issues

- It's a little complex, even for very simple problems like this one. However, DOM does have an internal logic; and once you become accustomed to it, you'll find it's actually not that hard to use.
- The second downside to DOM is that it does not expose as much of the information in an XML document as SAX does. These include unparsed entities, notations, attribute types, and declarations in the DTD. Some of this will be provided in DOM Level 3.
- The third downside to DOM is that it's not as complete as SAX. Much of the code in the Example is actually part of the Xerces parser rather than standard DOM. Such parser specific code is virtually impossible to avoid when programming in DOM because DOM doesn't give you any way to create a new XML document, create a new parser, or write a Document onto an output stream.

JAXP

- In Java 1.4, the Crimson XML parser and the SAX2, DOM2, and TrAX APIs are bundled into the standard Java class library. Also a couple of factory classes were included, this is called "Java API for XML Processing".
- When starting a new program the question is whether to choose SAX or DOM. The question is not whether we should use SAX or JAXP, or DOM or JAXP. SAX and DOM are part of JAXP. If we know SAX, DOM, and TrAX, we know 99% of JAXP.
- The only public part of JAXP that isn't part of its component APIs are the factory classes in javax.xml.parsers. These can be used to create new documents in memory and load documents from text files and streams.
- Example 3 is a JAXP client for the XML-RPC server. All the DOM standard code is the same as before except the parser-dependent parts from the org.apache packages

JAXP Example

- The request document is built again as a tree. This time a DocumentBuilderFactory from JAXP does the building instead of the Xerces-specific
- When it becomes time to serialize the Document, the JAXP solution again diverges. Here, FibonacciDOMClient used Xerces-specific classes. FibonacciJAXPClient uses TrAX..
- Finally, parsing the server response is much the same as before. However, this time instead of using the Xerces-specific DOMParser class, we use the same DocumentBuilder that created the request document.
- DocumentBuilder may delegate the parsing to Xerces anyway, depending on which classes are where in your class path, and how certain environment variables are set. However, there's no need for code at this level to know that implementation detail.

SAX in Detail

- The Simple API for XML, SAX, was invented in late 1997/early 1998
- SAX was designed around abstract interfaces rather than concrete classes so it could be layered on top of parsers' existing native APIs.
- The ease with which SAX could be implemented by many parser vendors with very different architectures contributed to its success and rapid standardization.
- SAX has been unofficially ported to several other object oriented languages including C++, Visual Basic, Python, and Perl. The general patterns and names of most functions remain the same.
- In late 1999, work began on SAX2. This was a radical reformulation of SAX that, while maintaining the same basic event-oriented architecture, replaced almost every class in SAX1. The main impetus for this radical shift was the need to make SAX namespace aware.

SAX Parsing

- Parsing is the process of reading an XML document and reporting its content to a client application while checking the document for well-formedness.
- SAX represents parsers as instances of the XMLReader interface. The specific class that implements this interface varies from parser to parser.
  - For example, in Xerces it's org.apache.xerces.parsers.SAXParser.
  - In Crimson it's org.apache.crimson.parser.XMLReaderImpl.
- Most of the time you don't construct instances of this interface directly. Instead you use the static XMLReaderFactory.createXMLReader() factory method to create a parser-specific instance of this class.
- Then you pass InputSource objects containing XML documents to the parse() method of XMLReader. The parser reads the document, and throws an exception if it detects any well-formedness errors

Parsing Example

- Example 4 demonstrates the complete process with a simple program whose main() method parses a document found at a URL entered on the command line. If this document is well-formed, a simple message to that effect is printed on System.out.
- Otherwise, if the document is not well-formed, the parser throws a SAXException.
- If an I/O error such as a broken network connection occurs, then the parse() method throws an IOException. In this case, you don't know whether or not the document is well-formed.
- This program's output is straightforward. For example:

```
%java SAXChecker
  http://www.cafeconleche.org

http://www.cafeconleche.org is well-
  formed.
```

Errors

- However, some readers will encounter a different result when they run this program. In particular, you may get this output:

```
%java SAXChecker
  http://www.cafeconleche.org
org.xml.sax.SAXException: System
  property org.xml.sax.driver not
  specified
```

- What this really means is that the parser has not properly customized its version of the XMLReaderFactory class. Parsers including Xerces and Crimson fail to do this. Consequently we need to set the org.xml.sax.driver Java system property to the fully package-qualified name of the Java class.
  - For Xerces, it's org.apache.xerces.parsers.SAXParser.
  - For Crimson, it's org.apache.crimson.parser.XMLReaderImpl.
  - For other parsers, consult the parser documentation.

Errors

- You can specify a one-time value for this property using the -D flag to the Java interpeter like this:

```
%java -
  Dorg.xml.sax.driver=org.apache.xerces.
  parsers.SAXParser SAXChecker
  http://www.cafeconleche.org/

http://www.cafeconleche.org is well-
  formed.
```

Content Handler

- The ContentHandler interface declares eleven methods. As the parser—that is, the XMLReader—reads a document, it invokes the methods in this interface.
- When the parser reads
  - a start-tag, it calls the startElement() method.
  - some text content, it calls the characters() method.
  - an end-tag, it calls the endElement() method.
  - a processing instruction, it calls the processingInstruction() method.
- The details of what the parser's read, e.g. the name and attributes of a start-tag, are passed as arguments to the method.
- Order is maintained throughout. That is, the parser always invokes these methods in the same order it sees items in the document.
- For example, the parser calls the startElement() method as soon as it's read a complete start-tag. It will not read past that start-tag until the startElement() method has returned.

Implementation

- An example should help make this clearer. We are going to write a very simple program that extracts all the text content from an XML document while stripping out all the tags, comments, and processing instructions.
- This will be divided into two parts, a class that implements ContentHandler and a class that feeds the document into the parser.
- Example 5 is the class that implements ContentHandler. It has to provide all eleven methods declared in ContentHandler.
- However, the only one that's actually needed is characters(). The other ten are do-nothing methods.
- When the parser reads content between tags, it passes this text to the characters() method inside an array of chars. The index of the first character of the text inside that array is given by the start argument. The number of characters is given by the length argument. In this class, the characters() method writes the sub-array of text from start to start+length onto the Writer stored in the out field.

Implementation

- By itself the TextExtractor class does nothing. There's no code in the class to actually invoke any of the methods or parse a document. Although code to do this could be placed in a main() method in TextExtractor, you caould place it in a class of its own called ExtractorDriver which is shown in Example 6.
- The main() method in this class performs the following steps:
  1. Build an instance of XMLReader using the XMLReaderFactory.createXMLReader() method.
  2. Construct a new TextExtractor object.
  3. Pass this object to the setContentHandler() method of the XMLReader.
  4. Pass the URL of the document you want to parse (read from the command line) to the XMLReader's parse() method.
- There's still no code that actually invokes the characters() or any other method in the TextExtractor class! The code that actually calls these methods is hidden deep inside the class library.

Ignorable white space

- One of the more obscure parts of the XML 1.0 specification is the perhaps misleadingly named "ignorable white space". This is white space that occurs between tags in places where the DTD does not allow mixed content. For example :

```
<?xml version="1.0"?>
<!DOCTYPE methodCall [
<!ELEMENT methodCall (methodName,
  params)>
<!ELEMENT params (param+)>
<!ELEMENT param (value)>
<!ELEMENT value (string)>
<!ELEMENT methodName (#PCDATA)>
<!ELEMENT string (#PCDATA)>
]>
<methodCall>
  <methodName>lookupSymbol</methodName>
      <params>
            <param>
                  <value>
                        <string>
                        Red Hat
                        </string>
                  </value>
            </param>
      </params>
</methodCall>
```

Ignorable white space

- This example has quite a bit of white space just for indenting. In particular, the spaces, carriage returns, and line feeds between <methodCall> and <methodName>, </methodName> and <params>, <params> and <param>, <param> and <value>, </value> and </param>, </param> and </params>, and </params> and </methodCall> only exist for indenting.
- Furthermore, the DTD says that these elements cannot contain #PCDATA, and therefore it's known that this white space is ignorable. Thus a validating parser will not pass these white space characters to the characters() method. Instead it passes them to the ignorableWhiteSpace() method.
- A non-validating parser might do the same, or it might pass the ignorable white space to the characters() method instead.
- If this matters, make sure you use a validating parser.

Further Exercises

- Have a look at:

http://www.cafeconleche.org/books/xmljava/chapters/ch06s05.html

Try to implement the user interface. Copy the example and try to run it.

DOM

- The Document Object Model, DOM for short, is an abstract data structure that represents XML documents as trees of nodes.
- Different interfaces in the org.w3c.dom package represent elements, attributes, parsed character data, comments, and processing instructions.
  - The root of the tree is a Document object that represents a complete well-formed document.
  - A parser reads an XML document from a stream and builds a Document object representing that XML document.
  - The client program calls the methods of Document and the other DOM interfaces to navigate the tree and extract information from the document.
  - Programs can even create completely new documents from scratch in memory which are then written into an XML file.

Historical Overview

- DOM is language neutral.
- DOM bindings exist for most OO languages including Java, JavaScript, C++, and Perl.
- DOM 0 only applied to HTML documents and only in the context of JavaScript.
- DOM 1 achieved some level of compatibility across browsers. The naming conventions feel a little wrong to a Java programmer, but DOM 1 does provides a solid core of functionality that covers maybe 75% of what programmers want to do when processing XML.
- DOM Level 2 cleaned up the DOM Level 1 interfaces. The big change was namespace support in the Element and Attr interfaces.
- All significant XML parsers that support DOM, support DOM Level 2.
- DOM Level 3 is is also going to add a lot more support for DTDs and schemas. But despite all its new features and functionality, DOM3 will not replace DOM2. Everything that works today in DOM2 will continue to work the same way in DOM3.

DOM Modules

DOM2 is divided into 14 modules in 8 different packages.

Core: org.w3c.dom
- The basic interfaces that can be used to represent any SGMLish, hierarchical tree-structured document

XML: org.w3c.dom
- The additional sub-interfaces of Node just for XML documents including

HTML: org.w3c.dom.html
- Interfaces designed specifically to represent the parts of an HTML document

Views: org.w3c.dom.views
- The AbstractView and DocumentView interfaces used to associate different views with one document. For instance, applying two style sheets to one XML document could produce two views.

StyleSheets: org.w3c.dom.stylesheets
- Basic interfaces for representing style sheets

CSS: org.w3c.dom.css
- Interfaces that specifically represent CSS style sheets

DOM Modules

CSS2: org.w3c.dom.css
- Provides shortcut methods for setting all the different CSS2 style properties.

Events: org.w3c.dom.events
- The interfaces in these classes establish a generic system that allows event listeners to be attached to nodes. Events nodes can respond to user interface events like mouse clicks, etc.

UIEvents: org.w3c.dom.events
- The UIEvent interface signals when a node represented on the screen in some form of GUI has received the focus, lost the focus, or been activated.

MouseEvents: org.w3c.dom.events
- The MouseEvent interface signals when, where, and with which keys pressed the user has clicked the mouse.

MutationEvents: org.w3c.dom.events
- It signals that a node has been added, removed, or modified in the document.

DOM Modules

HTMLEvents: org.w3c.dom.events

- Uses the base DOMEvent interface to report a dozen events specific to web browsers including load, abort, error, submit, reset, resize, scroll, etc

Traversal: org.w3c.dom.traversal

- Provides simple utility classes for performing common operations on a tree such as walking the entire tree or filtering out nodes that meet certain conditions.

Range: org.w3c.dom.ranges

- This extends DOM to cover sections of documents that don't neatly match element boundaries. For instance, it would be useful for indicating the section of text that the user has selected with the mouse.
- Aside from the core and XML modules, not all DOM implementations support all of these modules.
  - Most Java implementations do support the traversal module. The events module is not uncommon.

Application Specific

- A number of XML applications have built useful application-specific DOMs by extending the standard DOM interfaces.
- Where the generic DOM would use an Element object, a WML-specific DOM might use a WMLOptionElement or a WMLPElement or a WMLPostfieldElement object, as appropriate for the actual type of element it represents.
- These custom subclasses and subinterfaces have all the methods and properties of the standard interfaces, as well as other methods and properties appropriate only for their type.
- For example, A WML p element has align, mode, and xml:lang attributes, like this:

```
<p align="center" mode="wrap"
  xml:lang="en"> Hello! </p>
```

- Therefore, the WMLPElement interface has getter and setter methods for those three attributes

```
public void setMode(String mode);
public void setAlign(String align
   );
public void setXMLLang(String lan
   g);
public String getMode();
public String getAlign();
public String getXMLLang();
```

- An application specific DOM can enforce application specific rules such as "The mode attribute must have one of the values wrap or nowrap," though currently this isn't very common.
- The big issue for most of the application specific DOMs is parser support. To read these documents, we need not only a custom DOM but also a custom parser that knows how to generate the application specific DOM. That's a little harder to come by.
- With some effort, the Xerces DOM parser can be configured to produce HTML DOM Document objects for well-formed HTML and XHTML.

Trees

- According to DOM, an XML document is a tree made up of nodes of several types.
- The tree has a single root node, and all nodes in this tree except for root have a single parent node.
- Furthermore, each node has a list of child nodes. In some cases, this list of children may be empty, in which case the node is called a leaf node.
- There can also be nodes that are not part of the tree structure. For instance, each attribute node belongs to one element node but is not considered to be a child of that element.
- Recursion works very well on DOM data structures, as it does on any tree.

Trees

- Besides its tree connections, each node has a local name, a namespace URI, and a prefix; though for several kinds of nodes, these may be null.
- Each node also has a *node name*.
  - For an element or attribute, the node name is the prefixed name.
  - For other named things like notations or entities, the node name is the name of the thing.
  - For nodes without names such as text nodes, the node name is the value from the following list matching the node type:

    #document
    #comment
    #text
    #cdata-section
    #document-fragment
- Finally each node has a string value. For like text nodes and comments, this tends to be the text of the node. For attributes, it's the normalized value of the attribute. For everything else, including elements and documents, the value is null.

Node Types

- DOM divides nodes into twelve types, seven of which can potentially be part of a DOM tree:
  - Document nodes
  - Element nodes
  - Text nodes
  - Attribute nodes
  - Processing instruction nodes
  - Comment nodes
  - Document type nodes
  - Document fragment nodes
  - Notation nodes
  - CDATA section nodes
  - Entity nodes
  - Entity reference nodes
- However, of these twelve, the first seven are by far the most important; and a tree built by an XML parser will often contain only the first seven.

Example

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/css"
  href="xml-rpc.css"?>
<!-- It's unusual to have an xml-
  stylesheet processing instruction in
  an XML-RPC document but it is legal,
  unlike SOAP where processing
  instructions are forbidden. -->
<!DOCTYPE methodCall SYSTEM "xml-
  rpc.dtd">
<methodCall>
  <methodName>getQuote</methodName>
  <params>
      <param>
      <value><string>RHAT</string>
      </value>
      </param>
  </params>
</methodCall>
```

- The document node representing the root of this document has four child nodes in this order:
  - A processing instruction node for the xml-stylesheet processing instruction
  - A comment node for the comment
  - A document type node for the document type declaration
  - An element node for the root methodCall element

Element Nodes

- Each element node has a name, a local name, a namespace URI and a prefix.It also contains children.

For example, consider this value element:

`<value><string>RHAT</string></value>`

- When represented in DOM, it becomes a single element node with the name value. This node has a single element node child for the string element. Thie string element node has a single text node child containing the text RHAT.

Or consider this para element:

```
<db:para
  xmlns:db="http://www.example.com/"
  xmlns="http://namespaces.cafeconle
  che.org/"> Or consider this
  <markup>para</markup> element:
  </db:para>
```

Example cont.

- In DOM it's represented as an element node with the name db:para, the local name para, the prefix db, and the namespace URI http://www.example.com/. It has three children:
  - A text node containing the text Or consider this
  - An element node with the name markup, the local name markup, the namespace URI http://namespaces.cafeconleche.org/, and a null prefix.
  - Another text node containing the text element:.
- White space is included in text nodes, even if it's ignorable.

Attribute Nodes

- An attribute node has a name, a local name, a prefix, a namespace URI, and a string value.
- The value is normalized as required by the XML 1.0 specification. That is, entity and character references in the value are resolved, and all white space characters are converted to a single space.
- An attribute node also has children, all of which are text and entity reference nodes forming the value of the attribute.
- Attributes are *not* considered to be children of the element they're attached to. Instead they are part of a separate set of nodes.

For example, consider this Quantity element:

```
<Quantity amount="17" />
```

- This element has no children, but it does have a single attribute with the name amount and the value 17.

Leaf Nodes

- Only document, element, attribute, entity, and entity reference nodes can have children. The remaining node types are much simpler.

Text nodes

- Text nodes contain character data from the document stored as a String. Characters like & and < that are represented in the document by predefined entity or character references are replaced by the actual characters they represent.

Comment nodes

- A comment node has a name (which is always #comment), a string value (the text of the comment) and a parent (the node that contains it). That's all. For example, consider this comment:

```
<!-- Don't forget to fix this! -->
```

- The value of this node is Don't forget to fix this! . The white space on either end is included.

DOM Parsers

- DOM is defined almost completely in terms of interfaces rather than classes.
- Different parsers provide their own custom implementations of these standard interfaces. This offers a great deal of flexibility.
- DOM isn't quite as broadly supported as SAX, but most of the major Java parsers provide it including Crimson, Xerces, XML for Java, the Oracle XML Parser for Java, and GNU JAXP.
- DOM is not complete to itself. Almost all significant DOM programs need to use some parser-specific classes.
- DOM programs are not too difficult to port from one parser to another, but a recompile is normally required.
- JAXP, the Java API for XML Processing, fills in a few of the holes in DOM by providing standard parser independent means to parse existing documents, create new documents, and serialize in-memory DOM trees to XML files.

Example

- Because DOM depends so heavily on parser classes, its performance characteristics vary widely from one parser to the next.
- Speed is something of a concern, but memory consumption is a much bigger issue for most applications. Almost all DOM implementations use more space for the in-memory DOM tree than the actual file on the disk occupies. Generally the in-memory DOM trees range from three to ten times as large as the actual XML text.
- Some parsers including Xerces offer a “lazy DOM” that leaves most of the document on the disk, and only reads into memory those parts of the document the client actually requests.
- **Example 1** is a simple program that uses Xerces to check documents for well-formedness. You can see that it depends directly on the org.apache.xerces.parsers.DOMParser class.

JAXP Example

- The lack of a standard means of parsing an XML document is one of the holes that JAXP fills. If your parser implements JAXP, then instead of using the parser-specific classes, you can use the **javax.xml.parsers.DocumentBuilderFactory** and **javax.xml.parsers.DocumentBuilder** classes to parse the documents.

The basic approach is as follows:

  – Use the static DocumentBuilderFactory.newInstance() factory method to return a DocumentBuilderFactory object.
  – Use the newDocumentBuilder() method of this DocumentBuilderFactory object to return a parser-specific instance of the abstract DocumentBuilder class.
  – Use one of the five parse() methods of DocumentBuilder to read the XML document and return an org.w3c.dom.Document object.

- Example 2 uses JAXP to check documents for well-formedness.

The node interface

- If you have time try examples 9.8 and 9.11 from:

  http://www.cafeconleche.org/books/xmljava/chapters/ch09s07.html