Writing XML with Java

- You don’t need to know any special APIs like DOM or SAX or JDOM. All you have to know is how to System.out.println(). If you want to store your XML document in a file, you can use the FileOutputStream class instead.
- We are going to develop a program that writes Fibonacci numbers into an XML document.
- The principles of XML we will use here will be much more broadly applicable to other, more complex systems.
- The key idea is that data arrives from some source, is encoded in XML, and is then output. Where the input comes from, whether an algorithm, a file, a network socket, user input, or some other source, really doesn’t concern us here.
Fibonacci Numbers

• As far as we know, the Fibonacci series was first discovered by Leonardo of Pisa around 1200 C.E. Leonardo “How many pairs of rabbits are born in one year from one pair?” To solve his problem, Leonardo estimated that rabbits have a one month gestation period, and can first mate at the age of one month, so that each female rabbit has its first litter at two months. He made the simplifying assumption that each litter consisted of exactly one male and one female.

• The rabbits aren’t so important, but the math is. Each integer in the series is formed by the sum of the two previous integers. The first two integers in the series are 1 and 1. The Fibonacci series turns up in some very unexpected places including decimal expansions of \( \pi \).

• It’s very easy to calculate the Fibonacci sequence by computer. A simple for loop will do. For example, this code fragment prints the first 40 Fibonacci numbers:

```java
int low = 1;
int high = 1;
for (int i = 1; i <= 40; i++) {
    System.out.println(low);
    int temp = high;
    high = high+low;
    low = temp;
}
```

• However, the Fibonacci numbers do grow very large very quickly, and exceed the bounds of an int shortly before the fiftieth generation. Consequently, it’s better to use the java.math.BigInteger class instead.
import java.math.BigInteger;
public class FibonacciNumbers {
    public static void main(String[] args) {
        BigInteger low = BigInteger.ONE;
        BigInteger high = BigInteger.ONE;
        for (int i = 1; i <= 10; i++) {
            System.out.println(low);
            BigInteger temp = high;
            high = high.add(low);
            low = temp;
        }
    }
}

- When run, this program produces the first ten Fibonacci numbers:

```
java FibonacciNumbers
1 1 2 3 5 8 13 21 34 55
```

```xml
<?xml version="1.0"?>
<Fibonacci_Numbers>
    <fibonacci>1</fibonacci>
    <fibonacci>1</fibonacci>
    <fibonacci>2</fibonacci>
    <fibonacci>3</fibonacci>
    <fibonacci>5</fibonacci>
    <fibonacci>8</fibonacci>
    <fibonacci>13</fibonacci>
    <fibonacci>21</fibonacci>
    <fibonacci>34</fibonacci>
    <fibonacci>55</fibonacci>
</Fibonacci_Numbers>
```

- To produce this, just add string literals for the `<fibonacci>` and `</fibonacci>` tags inside the print statements, as well as a few extra print statements to produce the XML declaration and the root element start- and end-tags. XML documents are just text, and you can output them any way you’d output any other text document.
Java Example

```java
import java.math.BigInteger;
public class FibonacciXML {
    public static void main(String[] args) {
        BigInteger low = BigInteger.ONE;
        BigInteger high = BigInteger.ONE;
        System.out.println("<?xml version="1.0"?>");
        System.out.println("<Fibonacci_Nume rs>");
        for (int i = 0; i < 10; i++) {
            System.out.print(" <fibonacci>");
            System.out.print(low); System.out.println("</fibonacci>";
            System.out.println("<Fibonacci_Number s>");
            BigInteger temp = high;
            high = high.add(low);
            low = temp;
        }
        System.out.println("</Fibonacci_Numbe rs>");
    }
}
```

XML-RPC

- An XML-RPC request is just an XML document sent over a network socket.
- We will write a very simple XML-RPC client for a particular service.
  - We will just ask the user for the data to send to the server, wrap it up in some XML, and write it onto a URL object pointing at the server.
  - The response will come back in XML as well. Since we haven’t yet learned how to read XML documents, Response will be a text to System.out. Later we’ll pay more attention to the response, and provide a nicer user interface.
- The specific XML-RPC service we’re going to talk to is a Fibonacci generator. The request passes an int to the server. The server responds with the value of that Fibonacci number.
Example

• This request document asks for the value of the 23rd Fibonacci number:
  
  ```xml
  <?xml version="1.0"?>
  <methodCall>
    <methodName>calculateFibonacci</methodName>
    <params>
      <param>
        <value><int>23</int></value>
      </param>
    </params>
  </methodCall>
  ```

  And here’s the response:
  
  ```xml
  <?xml version="1.0"?>
  <methodResponse>
    <params>
      <param>
        <value><double>28657</double></value>
      </param>
    </params>
  </methodResponse>
  ```

Example

• The client needs to read an integer input by the user, wrap that integer in the XML-RPC envelope, then send that document to the server using HTTP POST.

• In Java the simplest way to post a document is through the `java.net.HttpURLConnection` class. You get an instance of this class by calling a URL object’s `openConnection()` method, and then casting the resulting `URLConnection` object to `HttpURLConnection`.

• Once you have an `HttpURLConnection`, you set its protected `doOutput` field to true using the `setDoOutput()` method and its request method to POST using the `setRequestMethod()` method.

• Then you grab hold of an output stream using `getOutputStream()` and proceed to write your request body on that.
What is an XML Parser

The parser is a software library (a Java class) that reads the XML document and checks it for well-formedness. Client applications use method calls defined in the parser API to receive or request information the parser retrieves from the XML document.

- The parser shields the client application from all the complex and not particularly relevant details of XML including:
  - Transcoding the document to Unicode
  - Assembling the different parts of a document divided into multiple entities.
  - Resolving character references
  - Understanding CDATA sections
  - Checking hundreds of well-formedness constraints
  - Maintaining a list of the namespaces in-scope on each element.
  - Validating the document against its DTD or schema
  - Associating unparsed entities with particular URLs and notations
  - Assigning types to attributes

What is a Parser?

- Defining Parser Responsibilities
- Evaluating Parsers

Validation

- Validating v. Nonvalidating Parsers

XML Interfaces

- Object v. Tree Based Interfaces
- Interface Standards: DOM, SAX

Java XML Parsers
What is an XML Parser?

An XML Parser enables your Java application or Servlet to more easily access XML Data.
Defining Parser Three Responsibilities

1. Retrieve and Read an XML document
   - For example, the file may reside on the local file system or on another web site.
   - The parser takes care of all the necessary network connections and/or file connections.
   - This helps simplify your work, as you do not need to worry about creating your own network connections.

2. Ensure that the document adheres to specific standards.
   - Does the document match the DTD?
   - Is the document well-formed?

3. Make the document contents available to your application.
   - The parser will parse the XML document, and make this data available to your application.

Why use an XML Parser?

- If your application is going to use XML, you could write your own parser.
- But, it makes more sense to use a pre-built XML parser.
- This enables you to do build your application much more quickly.

Parser Evaluation

- When evaluating which XML Parser to use, there are two very important questions to ask:
  - Is the Parser validating or non-validating?
  - What interface does the parser provide to the XML document?
XML Validation

- Validating Parser
  - a parser that verifies that the XML document adheres to the DTD.
- Non-Validating Parser
  - a parser that does not check the DTD.
- Lots of parsers provide an option to turn validation on or off.

Performance and Memory

Questions:
- Which parser will have better performance?
- Which parser will take up less memory?
- Validating parsers:
  - more useful
  - slower
  - take up more memory
- Non-validating parsers:
  - less useful
  - faster
  - take up less memory
- Therefore, when high performance and low-memory are the most important criteria, use a non-validating parser.
- Examples:
  - Java applets
  - Palm Pilot Applications
  - Huge XML Documents
XML Interfaces

- Broadly, there are two types of interfaces provided by XML Parsers:
  - Object/Tree Interface
  - Event Based Interface

Object / Tree Interface

- Definition: Parser reads the XML document, and creates an in-memory “tree” of data.
- For example:
  - Given a sample XML document on the next slide, what kind of tree would be produced?

Sample XML Document

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE WEATHER SYSTEM "Weather.dtd">
<WEATHER>
  <CITY NAME="London">
    <HI>20</HI>
    <LOW>12</LOW>
  </CITY>
</WEATHER>
```
Event Based Parser

- Definition: Parser reads the XML document, and generates events for each parsing event.
- For example:
  - Given the same XML document, what kind of events would be produced?
XML Parsing Events

- Events generated:
  1. Start of <Weather> Element
  2. Start of <CITY> Element
  3. Start of <HI> Element
  4. Character Event: 20
  5. End of </HI> Element
  6. Start of <LOW> Element
  7. Character Event: 12
  8. End of </LOW> Element
  9. End of </CITY> Element
  10. End of </WEATHER> Element

Event Based Interface

- For each of these events, your application implements "event handlers."
- Each time an event occurs, a different event handler is called.
- Your application intercepts these events, and handles them in any way you want.
Performance and Memory

- Questions:
  - Which parser is faster?
  - Which parser takes up less memory?

- Tree based:
  - slower
  - takes up more memory

- Event based:
  - faster
  - takes up much less memory

- Therefore, when high performance and low-memory are the most important criteria, use an event-based parser.

- Examples:
  - Java applets
  - Palm Pilot Applications
  - Parsing Huge Data files

XML Interface Standards

- Standards are important:
  - Easier to create XML applications
  - You can swap parsers as your application evolves.

- There are two main XML Interface standards:
  - Tree Based: Document Object Model (DOM)
  - Event Based: Simple API for XML (SAX)
### DOM / SAX

#### DOM
- Document Object Model
- Tree Based Interface
- Developed by the W3C
- Supports both XML and HTML
- Originally specified using an IDL (Interface Definition Language).
  - Hence, DOM Versions exist for Java, JavaScript, C++, Perl, Python.
- In this course, we will be studying JDOM (which is similar to DOM.)

#### SAX
- Simple API for XML
- Event Based
- Developed by volunteers on the xml-dev mailing list.
- [http://saxproject.org](http://saxproject.org)
- More on this next lecture…
Java XML Parsers

- Apache Xerces
  - Validating and Non-validating Options
  - Supports DOM and SAX.
  - http://xml.apache.org

- For a full list of XML Parsers, go to http://www.xmlsoftware.com/parsers

- Note that XML Parsers also exist for lots of other languages: C/C++, JavaScript, Python, Perl, etc.

- Most parsers support both DOM and SAX, and most have options for turning validation on or off.
Output Without Markup

• The clients for the XML-RPC server we have seen simply printed the entire document on the console. Now we will extract just the answer and strip out all the markup.

• That is, the user interface will look something like this:
  ```
  C:\XMLJAVA>java FibonacciClient 9
  34
  ```

• From the user’s perspective, the XML is completely hidden. The user neither knows nor cares that the request is being sent and the response received in an XML document.

• In fact, the user may not even know that the request is being sent over the network rather than being processed locally. All the user sees is the very basic command line interface.

Parsing Known XML

Talking to specific server you know details about the server like:

– the root element is methodResponse.
– the methodResponse element contains a single params element that in turn contains a param element.
– the param element contains a single value element.
– the value element contains a single double element that in turn contains a string representing a double. If the server returns a value with a type other than double, you’d probably respond by throwing an exception, just as you would if a local method you expected to return a Double instead returned a String.

All of this is specified by the XML-RPC specification. If any of this is violated in the response you get back from the server, then that server is not sending correct XML-RPC. You’d probably respond to this by throwing an exception.
Parsing Known XML

• The main point is this: most programs you write are going to read documents written in a specific XML vocabulary.
• They are not going to be designed to handle absolutely any well-formed document that comes down the pipe.
• However, you do need to make some assumptions about the format of your documents before you can reasonably process them.
• It’s simple enough to hook up an InputStreamReader to the document, and read it out.

```java
public printXML(InputStream xml) { int c; while ((c = xml.read()) != -1) System.out.write(c); }
```

• To actually extract the information you need to determine which pieces of the input you actually want and separate those out from all the rest of the text.

Example

• The readFibonacciXMLRPCResponse() method in the first example does exactly this by
  – reading the entire XML document into a StringBuffer
  – converting the buffer to a String,
  – using the indexOf() and substring() methods to extract the desired information.
  – The main() method connects to the server using the URL and URLConnection classes.
  – sends a request document to the server using the OutputStream and OutputStreamWriter classes.
  – passes InputStream containing the response XML document to the readFibonacciXMLRPCResponse() method.
What can go wrong

- This stream- and string-based solution is far from robust. In particular, it will fail if:
  - The document returned is encoded in UTF-16 instead of UTF-8
  - An earlier part of the document contains the text "<value><double>"m, even in a comment.
  - The response is written with line breaks between the value and double tags like this:
    `<value> <double>28657</double> </value>`
  - There’s extra white space inside the double tags like this:
    `<double >28657</double >`

- And this is a simple example where we just want one piece of data that's clearly marked up. The more data you want from an XML document, and the more complex and flexible the markup, the harder it is to find using basic string matching or even the regular expressions introduced in Java 1.4.

XML Parser

- Straight text parsing is not the appropriate tool with which to navigate an XML document.
- The structure and semantics of an XML document is encoded in the document’s markup, its tags and its attributes; and we need a tool that is designed to recognize and understand this structure as well as reporting any possible errors in this structure.

  This tool is called an XML parser.

- The parser is a software library (in Java it’s a class) that reads the XML document and checks it for well-formedness.
- One of the original goals of XML was that it be simple enough that a DPH be able to write an XML parser. The exact interpretation of this requirement varied from person to person. On one extreme, the DPH was assumed to be a Web designer without any formal training in programming who was going to do it in a weekend. On the other the DPH was assumed to be Larry Wall and he was allowed two months for the task. The middle ground was a smart grad student and a couple of weeks.
XML Parser

- The parser shields the client application from all the complex and not particularly relevant details of XML including:
  - Transcoding the document to Unicode
  - Assembling the different parts of a document divided into multiple entities.
  - Resolving character references
  - Understanding CDATA sections
  - Checking hundreds of well-formedness constraints
  - Maintaining a list of the namespaces in scope on each element.
  - Validating the document against its DTD or schema
  - Associating unparsed entities with particular URLs and notations
  - Assigning types to attributes
- The most important decision you'll make at the start of an XML project is the application programming interface (API) you'll use. Many APIs are implemented by multiple vendors, so if the specific parser gives you trouble you can swap in an alternative, often without even recompiling your code.

Choosing an API

- There are two major standard APIs for processing XML documents with Java, the Simple API for XML (SAX) and the Document Object Model (DOM).
- In addition there are a host of other, somewhat idiosyncratic APIs including, JDOM, dom4j, ElectricXML, and XMLPULL.
- Each specific parser generally has a native API that it exposes below the level of the standard APIs. For instance, the Xerces parser has the Xerces Native Interface.
- Each of these APIs has its own strengths and weaknesses.
Choosing a Parser

• When choosing a parser library many factors come into play. These include what features the parser has, how much it costs, which APIs it implements, how buggy it is, and last and certainly least how fast the parser parses.

Features
– Parsers can be roughly divided into three categories:
  – Fully validating parsers
  – Parsers that do not validate, but do read the external DTD
  – Parsers that read only the internal DTD subset and do not validate.
– There’s also a fourth category of parsers that reads the instance document but do not perform all the mandated well-formedness checks. Technically such parsers are not allowed by the XML specification, but there are still a lot of them.

• Parsers also differ in their support for subsequent specifications and technologies. All parsers worth considering support namespaces and automatically check for namespace well-formedness as well as XML 1.0 well-formedness.
• Xerces and Oracle are Java parsers that support schema validation.
• Some parsers also provide extra information not required for normal XML parsing. For instance, at your request, Xerces can inform you of the ELEMENT, ATTLIST, and ENTITY declarations in the DTD. Crimson will not do this, so if you need to read the DTD you’d pick Xerces over Crimson.
Choosing a parser

API support
- Most of the major parsers support both SAX and DOM. (Xerces and Crimson).
- The other APIs including JDOM and dom4j generally don’t provide parsers of their own. Instead they use an existing SAX or DOM parser to read a document which they then convert into their own tree model. Thus they can work with any convenient parser. The notable exception here is ElectricXML which does include its own built-in parser.

License
- One often overlooked consideration when choosing a parser is the license under which the parser is published. Most parsers are free however, license restrictions can still get in the way.

Correctness
- An often overlooked criterion for choosing a parser is correctness, how much of the relevant specifications are implemented how well. Although no parser is perfect, some parsers are definitely more reliable than others.
- The fact is that every single parser has sooner or later exhibited significant conformance bugs. Most of the time these fall into two categories:
  - Reporting correct constructs as errors
  - Failing to report incorrect syntax.
- Unnecessarily rejecting well-formed documents prevents you from handling data others send you.
- If a parser fails to report incorrect XML documents, will cause problems for people and systems who receive the malformed documents.
Choosing a Parser

Efficiency

• The last consideration is efficiency, how fast the parser is and how much memory it uses.

• Efficiency should be your last concern when choosing a parser. As long as you use standard APIs and keep parser-dependent code to a minimum, you can always change the underlying parser later.

• The speed of parsing tends to be dominated by I/O considerations. If the XML document is served over the network, it’s entirely possible that the speed with which data can move over the network is the bottleneck, not the XML parsing at all.

• In situations where the XML is being read from the disk, the time to read the data can still be significant even if it’s not quite the bottleneck it is in network applications.

Available Parsers

Xerces

• This is a very complete, validating parser that has the best conformance to the XML 1.0 and Namespaces in XML specifications.

• It fully supports the SAX2 and DOM Level 2 APIs, as well as JAXP.

• Xerces-J is highly configurable and suitable for almost any parsing needs.

• Xerces-J is also notable supporting the W3C XML Schema Language.

• The Apache XML Project publishes Xerces-J under the very liberal open source Apache license.

• Essentially, you can do anything you like with it except use the Apache name in your own advertising.

• Xerces-J 1.x was based on IBM’s XML for Java parser, whose code base IBM donated to the Apache XML Project.
Available Parsers

Crimson

- Crimson, previously known as Java Project X, is the parser Sun bundles with the JDK 1.4.
- Crimson supports more or less the same APIs and specifications Xerces does—SAX2, DOM2, JAXP, XML 1.0, Namespaces in XML, etc.—with the notable exception of schemas.

Piccolo

- Yuval Oren’s Piccolo is the latest entry into the parser arena. It is a very small, very fast, open source, non-validating XML parser. However, it does read the external DTD subset in order to apply default attribute values and resolve external entity references. Piccolo supports the SAX API exclusively. It does not have a DOM implementation.