

# JSP - XML

---

## JSP Structure

JSP Engine runs on the server (e.g. <http://jakarta.apache.org/tomcat/>)

The JSP file is translated into a .java servlet which is then compiled into a .class file. The HTML text in the JSP translates into a bunch of out.println(...) statements -- when run, they produce the output of the JSP.

The first access is slow; the later ones are very fast -- it's just running the .class file  
This is why JSP error messages can be obscure -- the system observes a problem in the .java, but has to report the error in terms of the .jsp.

## Basic JSP

Basic features: include files, refer to bean values

Syntax is general, but not that nice `<jsp:getproperty id="bean" property="title" />`

Runs pretty fast

## Problems

Iteration -- How do you write a JSP that loops through an array of values and produces some HTML for each

Conditional -- It would be nice to have an IF statement to control what goes in the output

Better syntax -- could we have a syntax more friendly to the non-programmer community (i.e. people who are familiar with HTML and that's it)

## JSP Tag Libraries

A JSP "tag libraries" defines a new tag type which will be available to JSPs

There is work to define a standard tag library -- doesn't exist yet, but it'll be an important part of future JSPs

You can also define a custom, domain specific tag library for your application

The tag library code is written in Java

The syntax is better....

```
<cs193i:magic />  
<cs193i:dblookup key="foo" />  
<cs193i:bookTitle />
```

Looping could look like this...

```
<cs193i:loop name="bean" property="titles">  
  <td>  
    <cs193i:item property="title" />  
  </td>  
</cs193i:loop>
```

# XML Introduction

Just a format standard

XML is just a way of describing a bunch of bindings in a textual form. It's a simple thing, but by being standard, it makes many boring incompatibilities go away.

Trying to make things work together more easily -- a data exchange format.

It makes compatibility easier -- it does not make it zero effort however!

<http://www.xml.org/>

## DTD

DTD -- formal structure description -- meta.

The parser or other tool can formally check that a document meets the DTD structure definition. In theory, just regular code does not need to worry about structure errors -- it's handled by the parser/DTD system.

## Backward/Forward/Round-Trip

With intelligent use of XML, a new version of an app will be able to read the docs of the old version -- just don't get confused if certain new nodes are not there

With intelligent use of XML, an old version of an app may be able to read the new docs -- just ignore nodes you don't understand.

Round-trip (this is hard), the versions can read each others docs, and write them out again without affecting the other version's state. This requires the old version ignore the new nodes, yet preserve them and write them back out again after editing.

## Strict XML

Bad standards: TIFF, RTF, HTML somewhat-- different vendors implement it different ways, which destroys the network effect advantage of having a standard

XML has learned the lesson: behavior in all cases is defined. Where, in C, the def might say that a behavior in a weird case, like divide by 0, "undefined", XML will say that a correct implementation **must** throw an error and halt.

## Nodes

Nodes

Like HTML tags

`<dots> ..... </dots>`

Can enclose other things with the `<node>...</node>` form

A node can start and end with one tag like this: `<node/>`

## Attributes

A node can have name/value attributes defined inside its tag

`<node foo="bar" pi="3.14">`

## 1. Text Markup

Can have free form text between the `<node>blah blah blah</node>` pairs -- like HTML

## 2. Tree Shape

If just have nodes inside the nodes with no free text, then it's just a big tree. This is the way I have used XML in programming projects -- save out the internal state as an XML tree

## Attributes vs. Child Nodes

Suppose we want to have a "dot" that stores an x and a y

There is **not** wide agreement about which of these is better, but I prefer the "attribute" way for fixed number of children

If a node can have an arbitrary number of children, then certainly the `<parent>`

`<child>..</child> <child>..</child> </parent> style is best`

Attributes can be used where the number of children is fixed

### 1. Attributes

XML

```
<dot x="27" y="13">
```

### 2. Children

XML

```
<dot>
  <x>27</x>
  <y>13</y>
</dot>
```

## XML Example

The "Dots" XML format -- a set of (x,y) points

root node : "dots" -- parent of dot nodes

child nodes : "dot" -- each with "x" and "y" attributes

```
<?xml version="1.0" encoding="UTF-8"?>
<dots>
  <dot x="72" y="101" />
  <dot x="170" y="164" />
  <dot x="184" y="158" />
  <dot x="194" y="146" />
  <dot x="191" y="133" />
  <dot x="164" y="84" />
  <dot x="119" y="89" />
</dots>
```

## Java XML

JAXP project

<http://java.sun.com/xml/>

SAX

Simple parser

DOM

Tree of nodes

Can iterate over the tree to look at the nodes

Can edit the tree: add/remove nodes

Jar Files

jaxp.jar and crimson.jar -- the code we're using from Jaxp

These are in the cs108/jars directory

## Leland Compiling

Add /usr/class/cs108/jar/jaxp.jar and /usr/class/cs108/jar/crimson.jar to your CLASSPATH

CS108/jar is the long-term home for many java .jar files

1. Edit .cshrc to have one long setenv line for the CLASSPATH (not two lines)...

```
setenv CLASSPATH /usr/class/cs193i/tomcat/lib/servlet.jar
:/usr/class/cs108/jar/jaxp.jar:/usr/class/cs108/jar/crimson.jar:.
```

2. Run the .cshrc

```
%> source .cshrc
```

3. Check the CLASSPATH

```
elaine0:~> echo $CLASSPATH
/usr/class/cs193i/tomcat/lib/servlet.jar:/usr/class/cs108/jar/jaxp.jar:/usr/class/cs
108/jar/crimson.jar:.
```

## SAX Parsing

Faster than DOM

Does not build the whole tree

Reads the doc from start to end

Sends notifications at each node

## Notifications

startElement(name, attrs) -- <dot>  
endElement(name) -- </dot> -- often you can ignore this one  
characters() -- called for characters between

## Notification / State Machine Strategy

Do something on each notification --e.g. startElement()

Have ivars to keep track of a state across notifications -- earlier nodes that affect later nodes

Assume the structure is correct -- let the parser notice structural errors for you. In this case, notice that if there were <foo>...</foo> nodes added in the doc, we just ignore them. Also, we don't check that the <dots> node is there.

## XMLDotReader.java

```
// XMLDotReader.java
// (updated to use SAX-2 which has slightly different prototypes
import java.io.*;
```

```

import java.util.*;

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

/**
This is a simple class that can read state out of an XML file
using a SAX state-machine parser.

http://java.sun.com/xml/jaxp-1.1/docs/api/

In this case, we support data like this, where the flip
node switches x,y...

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

<dots>
  <dot x="81" y="67" />
  <dot x="175" y="122" />
  <flip>
    <dot x="175" y="122" />
    <dot x="209" y="71" />
  </flip>
  <dot x="209" y="71" />
</dots>

Based on the JAXP sample code, updated for SAX-2 where the
prototypes for for startElement() and endElement() changed.
*/
public class XMLDotReader extends DefaultHandler
{

    public static void main (String argv [])
    {
        if (argv.length != 1) {
            System.err.println ("Usage: cmd filename");
            System.exit (1);
        }

        try {
            XMLDotReader xr = new XMLDotReader();
            InputStream in = new BufferedInputStream( new FileInputStream(new
File(argv[0])));
            xr.read(in);
        } catch (Throwable t) {
            t.printStackTrace ();
        }

    }

    /**
    Read the XML in the given file
    */

```

```

public void read(InputStream stream) {
    try {
        SAXParserFactory factory = SAXParserFactory.newInstance();
        SAXParser saxParser = factory.newSAXParser();
        clear();
        saxParser.parse(stream, this);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}

```

```

public XMLDotReader() {
    clear();
}

```

```

// State we keep track of -- like a state machine,
// where startElement() etc. keep getting called
private int x;
private int y;
private boolean flip;

```

```

public void clear() {
    x = -1;
    y = -1;
    flip = false;
}

```

```

//=====
// SAX DocumentHandler methods
//=====

```

```

public void startDocument ()
throws SAXException
{
    //System.out.println("startDocument");
}

```

```

public void endDocument ()
throws SAXException
{
    //System.out.println("startDocument");
}

```

```

/**
    Called for each node
    -look at localName and atts to see the node state
    -process that node if appropriate
    -or, update our state to affect future calls to startElem()
    or characters()

```

```

NOTE: With SAX-2, the arguments to startElement() changed
*/

```

```

public void startElement (String namespaceURI, String localName,
                        String rawName, Attributes atts)
throws SAXException
{
    //System.out.println("start element:" + localName);
    if (localName.equals("dot")) {
        x = Integer.parseInt(atts.getValue("x"));
        y = Integer.parseInt(atts.getValue("y"));

        if (flip) {
            int temp = x; x = y; y = temp;
        }

        // do something with our x,y state (could wait for endElement)
        System.out.println(x + ", " + y);
    }
    else if (localName.equals("flip")) {
        flip = true;
    }
}

// Called at the end of each element --
// arguments changed with SAX-2
public void endElement(java.lang.String uri,
                    java.lang.String localName,
                    java.lang.String qName)
throws SAXException
{
    //System.out.println("end element:" + localName);

    if (localName.equals("flip")) {
        flip = false;
    }
}

// Called for characters between nodes
public void characters (char buf [], int offset, int len)
throws SAXException
{
    //String s = new String(buf, offset, len);
    //s = s.trim();
    //if (!s.equals("")) {
        //System.out.println("characters:" + s);
    //}
}
}

```

## HW4 XML Strategy

Here's a suggested way to do it

Start with the XMLDotReader source code (in the hw directory)

Change the XML reader object so it has a Vector to store the usernames and a Vector to store the passwords.

The servlet can create an instance of the XML reader and have it parse the XML the first time doGet() is called

Check a username/password pair by looking for the corresponding pair in the Vectors