

# XML

---

## XML -- Hype and Reality

Extensible Markup Language

Just a textual data 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 -- creating real value.

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

XML helps with the basic problem of structure and parsing. To be compatible, two applications still need to agree on the **meaning** of the data, but at least the parsing is taken care of.

DTD -- meta description of a class of XML documents.

Formal description of the allowed structure for a class of XML documents

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.

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

<http://www.w3.org/XML>

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

## XML Tags

Tags -- meta content in the text. Like HTML tags

Here is some text `<red>like this</red>` surrounded by tags

Tags are case sensitive `<foo>` and `<FOO>` may be treated differently

Between the tags there may be raw text and/or more tags

Tags with nothing in between them, `<tag></tag>`, can be written as `<tag />`

Tag attributes

Stores a binding inside of a tag

May use single quote (') or double quote ("), but some sort of quote is required

```
<dot x="72" y="13" />
```

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

# Special Characters

A few characters have meta-meaning in XML, and so must always be encoded.

Note that the encodings end with a semi-colon (;)

< encode as: &lt;  
 > encode as: &gt;  
 & encode as: &amp;  
 " encode as: &quot;  
 ' encode as: &apos;

## 1. XML Text Strategy

XML can be used like HTML -- large blocks of text with tags sprinkled around to mark sections of text.

```
<foo>And here is some <b>text</b></foo>
```

The resulting structure is somewhat free-form

## 2. XML Tree Strategy

Tree shaped

Can write XML without free text between tags, except the innermost (leaf) tags. In that case, the structure is like a tree.

e.g.

```
<person>
  <name>Hans Gruber</name>
  <id>123456</id>
  <username>hans</username>
</person>
```

In my experience, tree form is the most common use of XML in programming projects -- save out the internal state as an XML tree

## Tags vs. Attributes

Suppose we want to have a "dot" that stores an x and a y -- x and y are "sub-data" of the dot itse.f

How should the sub-data be stored in its <dot> ...

### 1. Attribute Method

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

### 2. Tag Method

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

## Tags vs. Attributes style

There is **not** complete agreement about exactly when to use tags vs. attributes. I prefer the "attribute" way where possible, since it seems simpler.

Attributes work best if the sub-data is either present or not, but there are never multiple instances of the sub-data. It is best if the sub-data itself is short.

```
<dot x='6' y='13' />
```

If a node can have an arbitrary number of children, then tags are the best way

```
<parent> <child>..</child> <child>..</child> <child>..</child>
</parent>
```

The tag method is also appropriate if the data is lengthy...

```
<description>How did our constructed suburban landscape
come to be so unpleasant, and what to do about it.
The Geography of Nowhere is a landmark work in growth
of the New Urbanism movement.</description>
```

## Dots 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

As a further complication, there may be <flip>...</flip> tags around some of the <dot> nodes, and in that case the x and y should be interpreted as swapped.

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

```
<dots>
  <dot x="1" y="2" />
  <dot x="3" y="4" />
  <flip>
    <dot x="5" y="6" />
    <dot x="7" y="8" />
  </flip>
  <dot x="9" y="10" />
</dots>
```

## Java XML Support

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 -- in Java 1.3, these contain the XML code. In Java 1.4, the XML classes are part of the basic default, and no jar files are required.

## SAX Strategy

Subclass off `DefaultHandler`, and implement the "notification" methods.

`startElement()` -- use the `qName` string which is the name of the tag

`endElement()`

`characters()` -- char data between tags, may be called multiple times to cover all the text. May be called just with whitespace.

Notifications are sent to the `DefaultHandler` object to mark the start of a node, end of a node, characters, etc.

1. Take action at the time of the notification, or...
2. Or, store state in ivars to be used in a later notification -- e.g. the "flip" boolean
3. Or, store state in ivars, so that later, a client can send some sort of `getXXX()` message to retrieve the data.

Simplest to assume that XML document is correct and well formed

## Aside: Bad SAX Design

If you read the docs for `startElement()`, it is confusing when to use the `qName` and when not since it depends on this "namespace" feature which is rarely used.

Parameters:

`attributes` - The specified or defaulted attributes.

`localName` - The local name (without prefix), or the empty string if Namespace processing is not being performed.

`qName` - The qualified name (with prefix), or the empty string if qualified names are not available

What the heck does that mean? It's a great example of a bad client oriented design.

"Client oriented design" dictates that if a client just wants to do something simple, the interface should meet that need without burdening the client with details they don't care about. The SAX interface does a bad job of this. For simple uses I recommend that you ignore namespaces and just use the `qName`.

I trust that a more client oriented interface for SAX will emerge in the future. The correct design should make it easy to just see the tag name in simple cases, and there should be extra code that can be added if the client wants to deal with namespaces.

"Simple things should be simple, hard things should be possible."

# XML Dot Reader

```
// XMLDotReader.java
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/

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>

*/
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 qName and atts to see the node state
    -process that node if appropriate
    -or, update our state to affect future calls to startElem()
    or characters()

```

```

In an unfortunate bit of design, the localName and qName (qualified name)
may be the empty string depending the use of the "namespace"
feature. Without namespaces, just use the qName.
*/
public void startElement (String namespaceURI, String localName,
                          String qName, Attributes atts)
throws SAXException
{
    //System.out.println("start element:" + qName);
    if (qName.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 (qName.equals("flip")) {
        flip = true;
    }
}

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

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

// Called for characters between nodes.
// May be called multiple times for the text within tag --
// once per line for example.
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);
    //}
}
}

```