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 applictions 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>with this</red> marked as red
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 (")
- <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: <
> encode as: >
& encode as &
" encode as: "
' encode as '

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 text</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

How should each x,y 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** wide agreement about exactly when to use tags vs. attributes.

I prefer the "attribute" way where possible, since it seems simpler. It works best when the number of children is fixed, and where the 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

```
<?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 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.
- These are in the cs108/jars directory -- add them to your project if compiling on a pre-1.4 system. At runtime, use the command line "java -cp jaxp.jar:crimson.jar MyClass" to run with the XML code.

DOM Document

In memory representation of the whole tree
Has a pointer to the root node
Building the DOM tree is a little expensive -- it's the whole XML tree built out of java objects.

Node / Element

The nodes that make up the XML tree -- a node represents each <tag>....</tag> section
Nodes contain other nodes -- "children"
Nodes can have attribute/value bindings
There can be free-form text in between nodes (like HTML), but we're not using that feature.
In JAXP, Element is a subclass of Node. Our code will tend to use Element, since Element responds to get/set attribute, but Node does not.

Root

The root node is the one child of the document object
The root contains all the content

1. Reading

Our technique
- Use the DocumentBuilder.parse() method to read the XML and build the DOM in memory.
- Traverse it and examine the nodes to get the data out

Alternatives
- The SAX interface will show you, one at a time, the nodes of the XML document. It does not build the tree in memory, so it's faster.
- Another technique would be to use the DOM tree as our data model itself, so there is no translation step for reading or writing.

Read DOM Into Memory

```
// Standard imports for XML
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import org.w3c.dom.*;


....


      // The following is the standard incantation to get a Document object
        DocumentBuilderFactory dbf =
            DocumentBuilderFactory.newInstance();

        dbf.setValidating(false);

        DocumentBuilder db = null;
        try {
```

```
            db = dbf.newDocumentBuilder();
        } catch (ParserConfigurationException pce) {
            pce.printStackTrace();
        }

        // Parse the XML to build the whole doc tree
        Document doc  = db.parse(file);
```

Traversal Methods

Once you have the document object in memory, it's easy to look at its nodes...

```
// Get root node of document
Element root = doc.getDocumentElement();

// Get list of children of given tag name
NodeList list = root.getElementsByTagName("tagname");

// Number of children in list
int len = list.getLength();

// Get nth child
Element elem = (Element) list.item(n);

// Get an attribute out of a element
// (returns "" if there is no such attribute)
String s = elem.getAttribute("attribute");
```

2. Writing

Our technique

- Construct the DOM Document tree in memory
- Trick: downcast the Document object to an XmlDocument
- XmlDocument responds to a write() message where it writes itself out in text form (undocumented trick)
- There is another technique that uses XSLT, but it is quite lenghty, so we will use our two line trick. In the spirit of "client oriented design" I expect that a one or two line way of doing this common operation will be added to the library someday.

Alternatives

- A faster technique would be to write the XML out just using println(), but then you need to make sure you're writing valid XM (take care of < for <, etc.)

DOM Writing Code

```
        /*
         TRICK: as of JAXP1.1, there is no documented way to write a Document
         object out. However, by digging around a little, I found this
         unsupported way.
        */

        // 1. Cast the doc down to an XmlDocument
        XmlDocument x = (XmlDocument) doc;

        // 2. XmlDocument knows how to write itself out Woo Hoo!
        x.write(out, "UTF-8");
```

DOM Editing Methods

```
// Create a new node (still needs to be added)
```

```
Element elem = document.createElement("tagname");

// Append a child node to an existing node
node.appendChild(child_node);

// Set an attribute/value binding in a node.
// (the strings should be xml-ready text --
// no embeded " or < or &)
node.setAttribute(attr-string, value-string);
```

Dots Example

Recall the "Dots" example -- a panel with an ArrayList of Point objects. Each Point represented the center of a dot, and the user could add dots and move them around.

The DotPanel example (later in the handout) shows how to save and load the Dots data with XML...

// DotPanel.java

```
// DotPanel.java
/**
 The DotPanel class demonstrates a few things...

 -Mouse tracking -- clicking makes a new point, clicking
 on an existing point moves it. The data model is the collection
 of points where there is a dot on screen.

 -Smart repaint -- only repaints the needed rectangle when a dot moves

 -File Open/Save -- uses the KDeskFrame/KinnerFrame
 code to provide a document/window interface.

 -Serialization -- has code to save and load the data model
 using Java serialization. See saveSerial() and loadSerial().

 -XML -- has code that uses the Java XML package (JAXP-1.1)
 to save and load the data model to XML text.
 See saveXML() and loadXML().
 http://java.sun.com/xml/
 The Jaxp libraries are in jaxp.jar and crimson.jar for Java 1.3 and earlier.
*/

import java.awt.*;
import javax.swing.*;
import java.util.*;
import java.awt.event.*;
import java.beans.*;
import java.io.*;


// Standard imports for XML
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import org.w3c.dom.*;

class DotPanel extends JPanel implements DocPanel  {
   private ArrayList dots; // represent each dot by its center point
```

```
public final int SIZE = 20;   // diameter of one dot

// remember the last point for mouse tracking
private int lastX, lastY;
private Point lastPoint;

public boolean smartRepaint = true;

 // dirty = changed from disk version
private boolean dirty;



/**
 Utility test-main creates a DotPanel in a window.
*/
public static void main(String[] args) {
    final boolean testing = false;

    // testing -> create a little panel window
    if (testing) {

      JFrame frame = new JFrame("Dot Panel");

      JComponent container = (JComponent) frame.getContentPane();

      DotPanel dotPanel = new DotPanel(300, 300);

      container.add(dotPanel);

         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.pack();
      frame.setVisible(true);
   }
   // otherwise -> use the real kdesk/kinner
   // We pass the "DotPanel" classname, and KDeskFrame
   // uses introspection to make objects of that class.
   else {
       KDeskFrame desk = new KDeskFrame("DotPanel", args);
   }
}


/**
 Create an empty DotPanel. Load the contents of the
 given File if it is non-null.
*/
public DotPanel(int width, int height) {
   super();
   setPreferredSize(new Dimension(width, height));
   setOpaque(true);
   setBackground(Color.white);

   dirty = false;
   dots = new ArrayList();


   /*
    Mouse Strategy:
    -if the click is not on an existing dot, then make a dot
    -note where the first click is into lastX, lastY
    -then in MouseMotion: compute the delta of this position
    vs. the last
    -Use the delta to change things (not the abs coordinates)
```

```
   */

   addMouseListener( new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
         //System.out.println("press:" + e.getX() + " " + e.getY());

         Point point = findDot(e.getX(), e.getY());
         if (point == null) { // make a dot if nothing there
            point = addDot(e.getX(), e.getY());
         }

         // Note the starting setup to compute deltas later
         lastPoint = point;
         lastX = e.getX();
         lastY = e.getY();

      }
   });


   addMouseMotionListener( new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
         //System.out.println("drag:" + e.getX() + " " + e.getY());

         if (lastPoint != null) {
            // compute delta from last point
            int dx = e.getX()-lastX;
            int dy = e.getY()-lastY;
            lastX = e.getX();
            lastY = e.getY();

            // apply the delta to that point
            moveDot(lastPoint, dx, dy);
         }
      }
   });

}

 // Default ctor, uses a default size
 public DotPanel() {
     this(300, 300);
 }


/**
 Generates a repaint for the rect around one dot
 smart: repaint the rect just around the dot
 standard: repaint the whole panel
*/
public void repaintDot(Point point) {
   if (smartRepaint) {
      repaint(point.x-SIZE/2, point.y-SIZE/2, SIZE+1, SIZE+1);
   }
   else {
      repaint();
   }
}


/**
 Moves a dot from one place to another.
 Trick: needs to repaint both the old and new locations
 Moving components get this right automatically --
```

```
 see component.setBounds().
*/
public void moveDot(Point point, int dx, int dy) {
   repaintDot(point);   // repaint its old rectangle
   point.x += dx;
   point.y += dy;
   repaintDot(point);   // repaint its new rectangle

   setDirty(true);
}


/**
 Private utility -- adds a dot to the data model.
*/
private Point addDot(int x, int y) {
   Point point = new Point(x, y);
   dots.add(point);
   repaintDot(point);

   setDirty(true);

   return(point);
}


/**
 Finds a dot in the data model that contains
 the given point, or return null.
*/
public Point findDot(int x, int y) {
   Iterator it = dots.iterator();
   while (it.hasNext()) {
      Point point = (Point)it.next();
      int left = point.x-SIZE/2;
      int top = point.y-SIZE/2;
      if (left<=x && x<left+SIZE &&
         top<=y && y<top+SIZE) {
         return(point);
      }
   }
   return(null);
}


/**
 Standard override -- draws all the dots.
*/
public void paintComponent(Graphics g) {
   // As a JPanel subclass we need call super.paintComponent()
   // so JPanel will draw the background for us.
   super.paintComponent(g);

   Iterator it = dots.iterator();


   while (it.hasNext()) {
      Point point = (Point)it.next();

      g.fillOval(point.x - SIZE/2, point.y - SIZE/2, SIZE, SIZE);
   }


}
```

```
/**
 --File Saving Stuff--
 from here down.
*/


/**
 Accessors for the dirty bit.
*/
public boolean getDirty() {
   return(dirty);
}
public void setDirty(boolean dirty) {
   this.dirty = dirty;
}


/**
 Save the DotPanel state using either
 the SERIAL or XML code.
 (required by the DocPanel interface)
*/
public void save(File file) {
   if (isXML(file)) saveXML(file);
   else saveSerial(file);
}


/**
 Given a file, write the data model to it with Java serialization.
 Makes an Point[] array of points and writes it
 which avoids the bother of iteration.
 We use an array instead of the ArrayList to keep our
 archived format as version-independent as possible
 (not really necessary).
*/
public void saveSerial(File file)  {
   try {
      ObjectOutputStream out = new ObjectOutputStream(
         new FileOutputStream(file));

      /*
       toArray() is a collection method that copies to
       an array. The "new Point[0]" shows the type
       of array you would like.
      */
      Point[] points = (Point[]) dots.toArray(new Point[0]);

      out.writeObject(points);   //  serialization!

      out.close();   // polite to close on the way out
      setDirty(false);
   }
   catch (Exception e) {
      e.printStackTrace();
   }
}


/**
 Inverse of saveSerial.
 Reads an Point[] array of Points, and adds
```

```
 them to our data model.
*/
private void loadSerial(File file)  {
   try {
      ObjectInputStream in = new ObjectInputStream(new FileInputStream(file));

      // Read in the object -- the CT type should be exactly as it was written
      // -- Point[] in this case.
      // Transient fields would be null.
      Point[] points =  (Point[])in.readObject();

      for (int i=0; i<points.length; i++) {
         dots.add(points[i]);
      }

      in.close(); // polite to close on the way out
      setDirty(false);
   }

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


/**
 Load the state from the given file using
 either the SERIAL or XML methods.
*/
public void open(File file) {
   if (isXML(file)) loadXML(file);
   else loadSerial(file);
}


// XML tag strings
public final String DOTS = "dots";
public final String DOT = "dot";
public final String X = "x";
public final String Y = "y";


/**
 XML strategy:

 -there's a single DOTS node
 -inside, there's a DOT node for each dot
 -each DOT node has 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>

 We use the Jaxp library to read and write XML.
*/
```

```
/**
 Decide if a file is XML or serialized format.
 Cheezy -- just uses the name.
 Later we could actually look at the file in some way,
 or try one method and fall back to the other.
*/
private boolean isXML(File file) {
   return(file.getName().toLowerCase().endsWith(".xml"));
}


/**
 Create the XML element for a single dot.
 We use X and Y attributes to store x and y.
*/
private Element createDotElement(Document doc, int x, int y) {
   Element dot =  doc.createElement(DOT);

   dot.setAttribute(X, Integer.toString(x));
   dot.setAttribute(Y, Integer.toString(y));

   return(dot);
}


/**
 Create the whole XML doc object in memory representing the current
 dots state.
 Creat the root node and append all the dot children to it.
*/
public Document createXML() {
   // The following is the standard incantation to get a Document object
   // (i.e. I copied this from the API docs)
     DocumentBuilderFactory dbf =
         DocumentBuilderFactory.newInstance();

     dbf.setValidating(false);

     DocumentBuilder db = null;
     try {
         db = dbf.newDocumentBuilder();
     } catch (ParserConfigurationException pce) {
      pce.printStackTrace();
     }

   Document doc  = db.newDocument();


   // Create the root node and add to the document
   Element  root = doc.createElement(DOTS);
   doc.appendChild(root);


   // Go through all the dots and append them to the DOTS node
   Iterator it = dots.iterator();
   while (it.hasNext()) {
      Point point = (Point)it.next();
      Element dot = createDotElement(doc, point.x, point.y);
      root.appendChild(dot);
   }

   return(doc);
```

```
}


/**
 Create an XML document for our state, and ask it to write itself out.

 <p>
 Note: creating the whole document object in memory like this is
 one strategy for writing XML. Another strategy would be to just
 println() the XML text out straight from our data model.
*/
public void saveXML(File file)  {
   try {
      Writer out = new OutputStreamWriter (new FileOutputStream(file));
      Document doc = createXML();

      /*
       TRICK: as of JAXP1.1, there is no simple, documented way to write a doc
       object out. However, by digging around a little, I found this
       unsupported way. There is a supported way via the XSLT lib, but it
       is far more clumsy than this 2 line trick.
      */

      // 1. Cast the doc down to an XmlDocument
      // (the long class name is required since I did not use an import
      // at the top of the file)
      org.apache.crimson.tree.XmlDocument x =
         (org.apache.crimson.tree.XmlDocument) doc;

      // 2. XmlDocument knows how to write itself out Woo Hoo!
      x.write(out, "UTF-8");

      out.close();
      setDirty(false);
   }
   catch (Exception e) {
      System.err.println("Save XML err:" + e);
   }
}



/**
 Inverse of saveXML.
 Build the XML node tree in memory and iterate through the DOT nodes.

 <p>
 Note: constructing the whole DOM tree in memory is one
 strategy. Another strategy would be to look at the XML
 nodes one by one, but without building the whole tree.
 The SAX parsers do that, and they are faster.
*/
```

```
   private void loadXML(File file)  {

      try {
         // The following is the standard incantation to get a Document object
           DocumentBuilderFactory dbf =
               DocumentBuilderFactory.newInstance();

           dbf.setValidating(false);

           DocumentBuilder db = null;
           try {
               db = dbf.newDocumentBuilder();
           } catch (ParserConfigurationException pce) {
            pce.printStackTrace();
           }

         // Parse the XML to build the whole doc tree
         Document doc  = db.parse(file);

         // Get the root
         Element root = doc.getDocumentElement();

         // Get all the DOT children
         NodeList dots = root.getElementsByTagName(DOT);

         // Iterate through them
         for (int i = 0; i<dots.getLength(); i++) {
            Element dot = (Element) dots.item(i);

            // Get the X and Y attrs out of the dot node
            addDot(Integer.parseInt(dot.getAttribute(X)),
               Integer.parseInt(dot.getAttribute(Y)));
         }

         setDirty(false);
      }
      catch (SAXException e) {
         System.err.println("XML parse err:" + e.getMessage());
      }
      catch (IOException e) {
         System.err.println("IO err:" + e.getMessage());
      }
   }


}
```

Standard Format

Like the plain text file, XML is a good, default way to store data in a way that will be easy for other programs to parse.
e.g. your application's data file
e.g. your application's prefs file -- why make up yet another text format?
e.g. data exchange format -- a format to export your data so that some other tool can read it
Deals with nesting, naming, quoting, ... in a standard way

XSL / XSLT

A movement to keep "presentation" out of XML
XSL is like a style sheet for XML
XML just stores the data, and then XSLT translates the XML into some other format, like HTML
XSLT defines an XML transform that can be used to produce other formats.
XSLT can be used for many arbitrary XML translations. The theory is that it will be easier to express simple translations and transforms in XSLT, instead of writing Java or Perl code to do the translation.

Big and Slow -- ok?

Data encoded in XML tends to take more space than other methods.
However it creates compatibility and saves programmer time -- historically that tradeoff has fared well, especially as hardware gets faster.
DOM parsers are slow; SAX parsers are somewhat slow
In theory, XML can be compressed like a ZIP file

Backward/Forward Compatibility

The tag names declare what each piece of data is
This makes it easier to have optional bits of data in the format, which makes backward/forward compatibility easier

Backward

Backward compatible: A new version of an app will be able to read the documents of the old version -- just don't get confused if certain new nodes are not there

Forward

Forward compatible: An old version of an app may be able to read the new docs -- just ignore nodes you don't understand.

Round-Trip Compatibility

Round-trip (this is somewhere between hard and impossible), the new and old versions of an application 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 in the document tree and write them back out again after editing.

XML - "strict" lesson

Bad standards: TIFF, RTF, HTML -- different vendors implement it different ways -- which makes the "space" of valid TIFF, HTML documents ill defined and randomly incompatible.

Given a TIFF file that works in one program, it's hard to know if it will work in another. This undermines the **network effect** advantage.

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.

As a result, the boundary between valid and invalid XML is sharply drawn -- an XML document should work the same against different XML parser implementations.

XML Scenarios..

1. App doc format

e.g. Draw documents

e.g. Apache prefs file

Rather than invent a custom file format, just use XML

Advantages

- use standard code libs for read/write
- Use standard conventions, for quoting, naming, etc. rather than making up your own format is easier for you and other programmers
- gain XML's features
- buzzword checkoff item

Disadvantages:

- XML is not space efficient
- may introduce more complexity than it is worth

2. Use DOM Tree Throughout

So far, we have read in the DOM tree and translated back and forth to our internal representation.

Could use the DOM tree itself as our representation -- store things in the DOM nodes, arrange the DOM nodes. Then no translation is required.

Also, this helps with the round-trip case

Disadvantage: it's more awkward to use your data model if it's in the DOM form (with current technology anyway)

It's unknown if the DOM strategy is worthwhile

3. XML vs. Database

Your Internet application could store its data as XML rather than text files

As the data gets larger and more complex, a database is probably a better choice than XML

Could still use XML as the "export" format from your database -- an intermediate format, than can be used as part of an SQL -> XML -> HTML or SQL -> XML -> PDF path.

SQL is generally a good "native" storage format. Can use XML as an export format, or an intermediate format as part of a multi-step translation: SQL -> XML -> HTML, although perhaps just SQL -> HTML is simpler.

XML is worthwhile if it really buys something in the translation. For example, if there are multiple output formats that XML can drive easily: SQL -> XML, and then XML ->A, XML -> B, XML -> C. Don't translate to XML without some benefit.

4. EZPrints.com

EZPrints.com offers a service where another website can send an "order" to EZPrints containing image and a shipping address. EZPrints reads the order, makes a print, and postal mails it to the given address.

XML is the perfect format for this problem -- EZPrints can publish a spec for what tags are used in the order. Various web sites can read that spec and write code to produce the XML. The process is simplified because all the parties have a common understanding of XML.

Even if both parties are using databases internally, XML makes a great common language of exchange.