XML

XML -- Hype and Reality

XML stands for eXtensible Markup Language

What fundamental CS problem Is XML supposed to solve?

Suppose you have a bunch of dots (x,y pairs) you need to represent in a program for processing.

In-memory representation is easy; use a Dot object to store the (x, y) coordinate and then a Dots class that wraps an ArrayList of Dot objects work well.

Now you need to save these dots in a file or convert to a byte stream to be sent over a network connection (The problem of **Serialization**)

We could use Java's default serialization mechanism, as we did for project 2 (Have Dot Implement Serializable and use ObjectOutputStream; but we have a number of problems:

- If we change the data structure (e.g., add a Z value to a dot), cannot load old file any more
- Cannot publish a spec for the file format so that other people can code a reader
- Difficult to code a parser in a different language (such as C/C++) that can read this file
- Collectively the problems above are known as the **Data Interchange Format** problem

Instead of using Java Serialization, we could come up with our own plain-text format, similar to what we did for project 1. The simple design of placing the values for X and Y for each dot on a single line, separated by a space, solve the problems above:

```
2  3
3  5
```

But we still have Issues:

- The format does not tell the structure of data; the same file could also be storing a 2x2 matrix, for example.
- Difficult to generalize to more complex data structures; what If some dots have more attributes than others? What If some dots could be associated with an unlimited number of data items?

How about:

```
Dot   X:  2  Y:  3  EndDot
Dot   X:  3  Y:  5  EndDot
```

Now we are getting closer, but it's hard to design a general specification that could result in a good data-format for any type of data, and have everyone like it enough to use it. Finding a good **Data Formatting Standard** is the problem that XML is designed to solve.

How does XML solve the problem?

- By using tags to separate and annotate data Items, it could portray (but not necessarily define, since that is application specific) a relationship between a data Item, its attributes, and any associated data Items.
- Defines a uniform representation of syntax such that "off-the-shelf" syntactic parsers can be produced

XML Is not the only solution, however It has become the most popular one. The reasons being:

- XML looks and feels like HTML, so people feel comfortable with it.
- Applications in almost any language can find "off-the-shelf" parsers, transformation libraries, and more
- DTDs and other schema languages can describe a particular XML schema for parsers to check against
- Applications only need to agree on the meaning, or the "semantics" of the data. They only have to do semantic parsing (In theory!)

Bottom Line:
By being a widely used standard, it makes many boring incompatibilities go away -- creating real value.

Where do I find out more Info?

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; e.g.

```
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, i.e. <tag></tag>, can be written as <tag />

- **Every opening tag must have matching closing tag to be considered "well-formed". Just <tag> without </tag> would not parse. (Different from HTML)**
- Closing tag must match the opening tag at the same nesting level; unlike HTML, which allows <bold><Italics>…</bold></Italics>, XML does not allow <e-mail><address>…</e-mail></address>. It must be <e-mail><address></address></e-mail>
- Tag attributes stores a binding inside of a tag; may use single quote (') or double quote (") for attribute values. example:

```
<dot x="72" y="13" />
<node foo="bar" pi='3.14'/>
```

Comments

Comments can be included in an XML document:

```
<!— This is a comment that would be ignored by an XML parser -->
```

Special Characters

A few characters have meta-meaning in XML and must always be encoded. Note that the encodings end with a semi-colon (;)

< encode as: <
> encode as: >
& encode as &
" encode as: "
' encode as '

Encoding special or reserved characters Is one of the biggest challenge In developing a data-formatting language. XML does this well, but not significantly better than other Ideas. In my opinion, the success of HTML has made people tolerate the above special characters and use the encoding.

The XML Tree

Each XML document contains tag that defines a root "element." This tag must enclose the entire document. I.e., the following XML file would be illegal:

```
<?xml version='1.0' encoding='utf-8'?>
<Person>
…
</Person>
<Person>
…
</Person>
```

If you cannot decide on what to call the root element, just use <xml> </xml>

The root element makes the XML data to closely resemble a tree structure. If we write XML without free text except between the Innermost (leaf) tags, the resulting structure is like a tree:

```
<CS193J-Staff>
  <Person job="Lecturer">
    <name>Nick Parlante</name>
    <e-mail>nick@stanford.edu</e-mail>
  </Person>
  <Person job="TA">
    <name>Yan Liu</name>
    <e-mail>yl314@stanford.edu</e-mail>
  </Person>
  <Person job="TA">
    <name>Henry Hsieh</name>
    <e-mail>henri@stanford.edu</e-mail>
  </Person>
</CS193J-Staff>
```

In my experience, this is the most common use of XML in programming projects.

Tags vs. Attributes

How do we use XML to represent a single 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 no wide agreement about exactly when to use tags instead of attributes and vice versa.

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

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

But suppose If a dot also has an id and also list of ids of dots that it's supposed to be connected with (i.e. we want to represent a graph), then using attribute for that list would look like:

```
<dot Id='1' x='6' y='13' neighbor='0' neighbor='2'….>
```

It works, but compare to:

```
<dot Id='1' x='6' y='13'>
      <neighbor Id='0'/>
      <neighbor Id='2'/>
      …
</dot>
```

I think this looks better, and is generally easier to parse.

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

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

I also think that you should use tags If the relationship between the parent tag and the would-be-child attribute Is not one-to-one. Also, the tag method is better 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>
```

There are other rules-of-thumb that you can use, such as using a tag for all data that users would see while using an attribute for data that the user would not see. It's something to get a feel for once you are in an environment where XML is used everyday.

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>
```

Can we enforce minimum of one <dot> in <dots>, without writing programming code?
Yes, with a DTD; see http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/sax/5a_dtd.html for details.

Java XML

JAXP project
- Overview http://java.sun.com/xml/
- Good tutorial http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/

SAX
- "Simple" parser; uses a event-based callback model that Invokes functions that you define whenever the parser encounters a tag or attribute.
- Very fast & has almost no memory overhead
- Does not require the entire XML document to be read before semantic processing can start (can stream XML data; e.g. If an XML data stream contains a Iist of transactions to perform, a parser using SAX can perform the transactions in parallel with parsing)
- Places more work on the programmer
- Have to access the XML data in a sequential form
- Not covered in this Lecture, but the example code do have SAX parsing code as well as DOM parsing code; if you are interested follow the tutorial at http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/sax/2a_echo.html

DOM
- Parses entire file into memory first represented by an Internal tree of nodes
  - Can optionally check XML syntax
- Can iterate over the tree to look at the nodes
- Can edit the tree: add/remove/change nodes
- If you know how to traverse a tree, you can use DOM
- High memory overhead
- Must have entire XML document to process semantically; cannot Invoke meaningful action until the entire document Is parsed syntactically.

Using the library files
- JAXP 1.1: xalan,jar, jaxp.jar, and crimson.jar -- these are in the cs108/jars directory
- JAXP 1.2: xalan.jar and xerces.jar – download the Java for XML Pack Winter 01 from http://java.sun.com/xml/downloads/javaxmlpack.html
- On unix, add these jar files to your classpath
- For CodeWarrior, add these jar files to your project

DOM Document

From an XML document builds an in-memory representation of the whole tree with a pointer to the root node. It Is costly, but easy to use.

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 it 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 Into our own run-time data structure (I.e., our ArrayList of Point objects)

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, but 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

```
// 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);
```

Essentially, we first create an instance of DocumentBuilderFactory, and then use that object to create an instance of DocumentBuilder (this is called the “Factory” design pattern, and used throughout JAXP), and finally use the document builder object to parse the document into memory, building the DOM tree in the process.

Traversal Methods

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

```
// Get root node of document; the root node is the outer most block
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

Creating and Editing Elements in the DOM Tree

```
// 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)
```

DOM Writing Code

First technique (JAXP Standard)

- Construct the DOM Document tree in memory
- Use an XSLT Transform to format the DOM tree for output (in our example we use the identity transform)

Second technique (An undocumented Trick in JAXP 1.1, no longer works in JAXP 1.2)

- Construct the DOM Document tree
- Downcast the Document object to an XmlDocument
- XmlDocument responds to a write() message where it writes itself out in text form (undocumented and does not work for JAXP 1.2)

Third technique

- Form the XML directly using print/println or string manipulation methods. You would need to make sure you're writing valid XML (take care of < for < etc.)

The following example uses the JAXP standard which works with both 1.1 and 1.2; the example code includes both this technique and the technique of forming XML directly.

```
 /**
 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.
*/
private Document createXMLDoc() {
  // 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(TAG_DOTS);
      doc.appendChild(root);

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

      return(doc);
}

public void writeToStream(InputStream is) {
     // Do it the more official way, build a DOM tree model
     // and then use an XSLT transform (in this example we
     // just use the 'identity' transform) to write out the result
     Document doc = createXMLDoc();  // build DOM tree

     try {
        // Use a Transformer for output
        TransformerFactory tFactory =
            TransformerFactory.newInstance();
        Transformer transformer = tFactory.newTransformer();

        DOMSource source = new DOMSource(doc);
        StreamResult result = new StreamResult(os);
        transformer.transform(source, result);
     } catch (TransformerConfigurationException tce) {
         tce.printStackTrace();
     } catch (TransformerException te) {
```

```
            te.printStackTrace();
        }
}
```

Essentially we first follow the “factory” design pattern to make a transformer, then wrap the XML document we generated with a DOMSource object, then wrap the output stream with a StreamResult object, and finally invoke transform with the doc as the source and the stream as the result.

Completed Dots Example

Paper copy of code handed out in class, and soft copy accessible on the web from http://www.stanford.edu/class/cs193j/assignments/XMLExample/
Or from Leland from /usr/class/cs193j/assignments/XMLExample
The setup script setup the classpath for you
The directory contains following classes:

Dot - stores a dot (x, y)
Dots - abstract; stores a collection of dots; contains abstract methods writeDotsToStream and readDotsFromStream
DotsSerialize - Subclass of Dots; achieves stream I/O through java Serialization
DotsXML - Subclass of Dots; achieves stream I/O through XML parsing and writing

In running DotsXML, experiment with the different XML reading/writing methods:

To read XML file with DOM:

```
java DotsXML readdom file.xml
```

To read XML file with SAX:

```
java DotsXML readsax file.xml
```

To write XML file with DOM:

```
java DotsXML writedom file.xml
```

To write XML file with direct-string formulation:

```
java DotsXML writedirect file.xml
```

XML Analysis

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. We have already used the identity transform.
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 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 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 other’s 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.
However, creates problems when trying to delete nodes, or If the schema has more dependency between nodes other than the default tree structure

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. Minidraw 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; backward/forward compatibility could be a big advantage
- "It's just kewl!"

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 slow (retrieving values of data may require look up on the attribute or element name) executing dot.getX() Is always faster than getElement("dot").getAttribute("X")

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. It has the ACID properties which XML does not; a hard disk crash could wipe out your XML files but Is often "shrugged off" by a good DBMS.

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.

DBMS is generally good for "native" and/or "permanent" storage. 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. Good XML application In the real world: 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.