Handout #17
Nick Parlante

# Advanced

## XML Niche

- Structured
- Standard Parsing
- Text + tags
- Tree structure
- e.g. pref file

## XML DTD

- Define meta info
- Define format -- e.g. MS Word 6
- Used by editor tools
- Used by verifiers

## Does not solve everything

XML defines the parsing basis so that two programs may share a tree of text.

It does not define the interpretation or semantics of that information.

e.g. it is now possible that you can write a program that can read in the MS Word 2002 doc format, but there's still the matter of interpreting that document in the same way as Microsoft. XML solves the first half of that problem -- reading it in.

## XML Docs

http://developer.java.sun.com/developer/products/xml/docs/api/
http://www.w3c.org/xml
http://www.xml.com/

## Code

Need xml.jar
Perl etc. all are adding XML support

# 1. SAX Parser

## Serial Access

## L-R notification

# 2. XmlDocument / DOM

## Parse -> Memory Tree

In memory rep of the whole tree
Has a pointer to the root node

## Build

Use to build an XML tree in memory for writing out

## Costly

The whole XmlDocument approach is more costly than SAX (reading) or just printlns (writing).

# Node Class

The nodes that make up the XML tree
Nodes contain other nodes -- "children"
Nodes can have attribute / value bindings

# Root Node

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

# 1. Writing

Construct the right XmlDocument tree in memory
Write it out

# 2. Reading

The XmlDocument reads itself into memory
Traverse it and examine the nodes get the data out

# DotExample

## Root -- "dots"

## Root children -- "dot"

## Each dot has "x" and "y" attributes

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

```
<dots>
  <dot x="72" y="101" />
  <dot x="72" y="82" />
  <dot x="81" y="65" />
  <dot x="194" y="146" />
  <dot x="191" y="133" />
  <dot x="164" y="84" />
  <dot x="119" y="89" />
</dots>
```

## Creation Methods

1. doc.createElement(tag-string)
2. node.appendChild(node)
3. node.setAttribute(attr-string, value-string)

## Traversal Methods

doc = XmlDocument.createXmlDocument(in, false);
root = doc.getDocumentElement
nodeList = root.getElementsByTagName(tag-string)
nodeList.getLength() -- number of children
nodeList.item(i) -- get that node
node.getAttribute(attr-string)

## Dot Code

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

```
public final String Y = "y";

/*
Here's our XML format...

 -there's a single doc object
 -there's a single "root" node tagged DOTS
 -there's a DOT node for each dot
 -each DOT node has X and Y attributes
 -the DOTs are appended to the root node


<?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>

*/


/*
 Create the XML node for a single dot.
 We use X and Y attributes to store x and y.
*/
public ElementNode createDotNode(XmlDocument doc, int x, int y) {
   ElementNode dotNode = (ElementNode) doc.createElement(DOT);

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

   return(dotNode);
}


/*
 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 XmlDocument createXML() {
   XmlDocument doc = new XmlDocument();

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


   // Go through all the dots and append them to the root
   // ("dots" is a collection of Points)
   Iterator it = dots.iterator();
   while (it.hasNext()) {
```

```
      Point point = (Point)it.next();
      ElementNode dotNode = createDotNode(doc, point.x, point.y);
      root.appendChild(dotNode);
   }

   return(doc);
}


/*
 Create an XML document for out state, and ask it to write itself out.
*/
public void saveXML(File file)  {
   try {
      Writer out = new OutputStreamWriter (new FileOutputStream(file));
      XmlDocument doc = createXML();

      doc.write(out, "UTF-8");   // XMLDoc knows how to write itself

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



/*
 Inverse of saveXML.
 Build the thing in memory, and itereat through the DOT nodes.
*/
private void loadXML(File file)  {

   try {
      InputStream in = new FileInputStream(file);

      // This parses the XML file and builds the XML doc in memory
      XmlDocument doc = XmlDocument.createXmlDocument(in, false);

      // 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);
      in.close();
```

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

# Java Library Areas:

## Collections

Built in collections better than Vector -- list, hashMap, Iterator, etc. etc.

All operations to Vector are synchronized which makes it too slow -- that was in retrospect a design mistake which has been fixed with the new collections (one has to wonder if the performance-not-important theme was applied a little too much in Java's design tradeoffs).

### Compile-Time Types

Collections are being revised so that the compile-time type is correct -- no more casting back result of elementAt(int)

The VM will still check the real type at runtime, you just don't need to put the cast in your source code

## JNI

Java Native Interface. Set up Java to be able to call native code. Typically used to connect Java to some legacy C / C++ code or to call some platform specific feature.

## Security

Java has all sorts of built in classes for signing and encryption

## JDBC

JDBC classes provide an API for speaking to an SQL database.

Ideally, a database will have a "JDBC driver"

(Slower) All databases have a (Microsoft) ODBC driver. There's a JDBC-ODBC bridge that Java programs can go through.

## Java 2d

A floating point imaging standard -- 2d imaging standards all look more similar than different, since they were all influenced by Postscript.

Support for color gradients, anti-aliasing, beziers, and floating point transforms like shearing and rotating.

# Java 3d

More embryonic than 2d -- uses some native code. Not the highest performance, but nice portability. Sun just announced some sort of strategic linkage with SGI on this.

# Servlet / JSP

A nice, portable, relatively fast standard for server side code -- see the CS193i handouts on servlets (www.stanford.edu/class/cs193i)

JSPs are like ASPs and PHPs, but they are in Java

# Jini

Java for networked toasters -- things can discover and cooperate dynamically by flinging Java around.

(rumor) Supposedly a lot of the excitement within Sun for Java revolves around such "embedded world of the future" applications. Java's portability could be crucial in such an environment (if the entities need to exchange code and objects, as opposed to just data).

I guess the idea is that Microsoft won the desktop, so it's easier to imagine success another domain. (IMHO) It's more important to make Java work well on the desktop.

# New I/O

## Old: One thread / socket

This has problems trying to support a very large number of sockets. Also, it was too difficult to implement interruption correctly.

## New: non-blocking variation

Support very large numbers of simultaneous "in flight" sockets efficiently.

# J2ME - Micro Edition

Runs on Windows CE

# KVM - smaller than J2ME

Runs on Palm pilot

# 1. Server Side

Servlets + hot spot + JSP + no GUI
popular now

# 2. Applets

## Used somewhat -- dynamic content

### Problem: AWT portability

profit center error

### Problem: sandbox

### Problem: performance (maybe)

## 3. Applications / Swing

### Yes: Custom

For custom applications works great
e.g. The verification tool used against our custom internal database

### No: Shrinkwrap 3rd party

### Installation problems

Does not pass the "mom" test. Sun needs to push the "jar" format better so it is possible to send someone something, they double click it, and it just runs.

### Performance problems maybe

### AWT = bad reputation

Swing has not yet overcome the bad reputation that AWT created

## 4. Palm pilot / toaster / java ring

Sun is very excited about the future of this sort of market.

### Pro

Java's programmer efficiency and portability are great here

### Con

Java's using a lot of memory is a real problem here

## Java Dynasty?

### Age of interpreted languages

### Java Network effects

Good tools, everyone learns it in school, good books .... for any new project, Java has a lot of built-in network effect advantages

### C/C++

### Perl, VB