XML and DOM
Historical Perspective

• 1989 - Tim Berners-Lee proposed a hypertext system for CERN including HTML and HTTP
• 1993 - Marc Andreessen unleashed the alpha version of Mosaic
• 1993 - (Sept) WWW traffic is 1% of the NFS backbone.
• 1994 - more than 200,000 web servers
• 2002 - more than 30,000,000 web servers
Pre-XML

• Factors leading to the creation of XML
• Problems with HTML
  – primarily presentation
  – hard to derive meaning from the markup
  – fixed tag set
  – static
• Web browsers were being viewed as potential application platforms.
Pre-XML (2)

- SGML - Standard Generalized Markup Language
  - Working standards draft 1980
  - Allow text editing, formatting, and information retrieval systems to share documents.

- Did not become widely used
  - general consensus: too heavy-weight
XML

- Goal: combine the power of SGML (extensibility) with the simplicity of HTML
- 1998: XML 1.0 standard published
- Focus of the language quickly became its value as a data interchange language
- XML is a meta-language
  - Also need to publish the description of the tags that an application will use (DTD)
Simple Format

- **Element**: basic unit
  - noun defining what the content (data) is
  - opening and closing tags with content in between
    `<tag>content</tag>`
- A document begins with an *XML declaration*
- **Root element** immediately follows and encloses the entire content of the document
  ```xml
  <?xml version="1.0"?>
  <book>
    [everything else]
  </book>
  ```
Attributes

• Information about the data (adjectives)
• Stored in the tag as name=value pairs.

<tag attribute="value">
  content
</tag>
Rules for well-formed XML

1) Elements that contain data must have start tags and ending tags.

2) Empty tags must be closed.
   • `<br />` or `<br> </br>

3) Elements should not overlap.
   • Bad Nesting
     `<trunk> <branch> </trunk> </branch>`

4) All attribute values must be wrapped in quotation marks: `<a href="newpage.html">
More Rules

5) A declaration must appear at the top of an XML document to signify what it is:
   - XML Declaration, e.g. `<?xml version="1.0"?>`
   - DocType Declaration e.g.
     `<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
     "http://www.w3.org/TR/html4/loose.dtd">`

6) XML is case sensitive, so `<TAG>` and `<Tag>` will be treated differently. (Use lower case.)
Document Object Model (DOM)

• Cross-language API for representing XML documents as trees
  – Easier to manipulate than strings or streams
  – But may require a lot of memory for large documents

• Several implementations in Java
  – This course uses org.jdom
Tree Structure

• The document

```html
<html>
<body>
<h1>Title</h1>
<p>A <em>word</em></p>
</body>
</html>
```
DOM Rules

• Every document's root is an object of type Document
  
• This has a single child of type Element
  – The root element of the document

• Its children may be:
  – Other elements
  – Text objects
  – Other things that we won't worry about

• Note: white space is preserved
  – Like the new lines in the previous slide

• But comments are not
public static void main(String[] args) {
    for (int i=0; i<args.length; ++i) {
        try {
            // Build document tree
            SAXBuilder builder = new SAXBuilder();
            Document doc = builder.build(args[i]);
            // Show top-level elements (next slide)
        } 
        catch (Exception e) {
            System.err.println(e);
        }
    }
}
Iterate over children

// Show top-level elements
Element root = doc.getRootElement();
Iterator ic = root.getChildren().iterator();
while (ic.hasNext()) {
    Element elt = (Element)ic.next();
    System.out.println(elt.getName());
}
Input and output

• Input
```xml
<?xml version="1.0" ?>
<doc>
<h1>First heading</h1>
<p>First</p>
<em>paragraph.</em></p>
<p><em>Second paragraph.</em></p>
</doc>
```

• Output
```
<doc>
<h1>First heading</h1>
<p>First</p>
<em>paragraph.</em></p>
<p><em>Second paragraph.</em></p>
</doc>
```
public static void descend(Element current, int depth) {
    for (int i=0; i<depth; ++i) {
        System.out.print(" ");
    }
    Element elt = (Element)current;
    System.out.println(elt.getName());
    Iterator ic = elt.getChildren().iterator();
    while (ic.hasNext()) {
        descend((Element)ic.next(), depth+1);
    }
}
The Visitor Pattern

• Often want to operate on a tree recursively
  – Count elements, search for text that matches a pattern, etc.
• Mechanics of traversing is the same every time
• So build a generic visitor that knows how to traverse the tree
  – Give it do-nothing methods that are invoked at specific times during traversal
  – Users derive from this class and override the methods they're interested in
A DOM Visitor

public abstract class DomVisitor {
    public DomVisitor() {}  
    public void visit(Element root) { 
        fDepth = 0; 
        preRoot(root); 
        atElement(root); 
        recurse(root); 
        postRoot(root);  
    }  
    protected void preRoot(Element root) {}  
    protected void postRoot(Element root) {}  
    protected void atElement(Element elt) {}  
    protected void atText(Text text) {}  
    ...  
}
protected void recurse(Element elt) {
    fDepth += 1;
    Iterator ic = elt.getContent().iterator();
    while (ic.hasNext()) {
        Object node = ic.next();
        if (node instanceof Element) {
            Element child = (Element)node;
            atElement(child);
            recurse(child);
        } else if (node instanceof Text) {
            atText(((Text)node));
        }
    }
    fDepth -= 1;
}
Building an Attribute Inventory

• Want to find out which attributes can appear with which elements
• Create a DOM visitor that inspects each element's attributes
• Result is a map in which
  – Keys are element names (e.g. "h1")
  – Values are sets of attribute names (e.g. "align")
• Do not record the attribute values
  – Exercise: extend this visitor to inventory them as well
public class Inventory extends DomVisitor {
    public Inventory() {
        fSeen = new HashMap();
    }
    protected void preRoot(Element root) {
        fSeen.clear();
    }
    protected void atElement(Element elt) {}

    protected Map fSeen;
}
protected void atElement(Element elt) {
    String eltName = elt.getName();
    Set seen = (Set)fSeen.get(eltName);
    if (seen == null) {
        seen = new HashSet();
        fSeen.put(eltName, seen);
    }
    Iterator ia = elt.getAttributes().iterator();
    while (ia.hasNext()) {
        String attrName =
            ((Attribute)ia.next()).getName();
        seen.add(attrName);
    }
}
Input and Output

<doc>
  <p align="left" role="lead">First.</p>
  <p align="center">Second.</p>
  <p align="right" font="em">Third.</p>
</doc>
Trimming the Tree

- Can add or remove nodes in DOM tree
  - Be careful about deleting items in a list while iterating over that list
  - Like cutting the branch you are standing on
- Pattern: delete or move on
  - When an item is deleted, items above it bump down
  - So either delete or increment loop index
protected void atElement(Element elt) {
    List content = elt.getContent();
    int i = 0;
    while (i < content.size()) {
        Object node = content.get(i);
        boolean keep = true;
        if (node instanceof Text) {
            Text text = (Text)node;
            if (text.getText().trim().length() == 0) {
                keep = false;
            }
        }
        if (keep) {
            i += 1;
        } else {
            content.remove(i);
        }
    }
}
Python

• Like JDOM, Python's DOM library is derived from the W3C standard
  – Uses idiomatic Python instead of trying to be 100% compatible with standard

• In fact, Python has two DOM libraries
  – xml.minidom doesn't have everything
  – But it's fast
import sys, xml.dom.minidom

def showTree(node, indent=0):
    print ' ' * indent + node.nodeName
    for child in node.childNodes:
        if child.nodeType == child.ELEMENT_NODE:
            showTree(child, indent+1)

for filename in sys.argv[1:]:
    doc = xml.dom.minidom.parse(filename)
    root = doc.documentElement
    showTree(root)
SAX

- The Simple (or Stream) API for XML
- Instead of creating a tree in memory, it calls methods each time the parser finds something interesting
  - Start of element, Block of text
  - End of element, Errors
- Neither better nor worse than DOM
- Needs less memory, since only a fraction of the document is stored at a time
- Users have to keep track of context themselves