Context

- Well formed XML documents may be represented "directed acyclic graphs" – or trees.
- Unlike the Simple API for XML (SAX) which operates on a byte stream XML document, The Document Object Model APIs (DOMs) API operate on the tree.
- The DOM APIs are more complex than SAX because:
  - There are multiple DOMs
  - There are multiple API’s
  - The resulting structure is more complex and much more manipulable

DOM versus the Document

- A Schema defines a document in terms of its allowable content, i.e. elements or text
- The root is the main element of the document
- Each element has children which are either subsumed elements or text
- The leaves of the tree are the text of the document
- The Document Object Model, or DOM, defines a tree of nodes which starts with a "root" node
- The DOM tree is made up of a series of nodes, only some of which are element nodes.
- One of the children of the root node is the root element of the DTD.
- Nodes are of multiple types – e.g. elements, attributes, text, etc.
The Parser and the API

- Underlying the DOM API is a parser:
  - Which scans an XML document for well-formedness and validity – throwing an exception if there is an error
  - That constructs a tree in memory according to the DOM model.
- The DOM API:
  - Is implemented as a series of interfaces bundled as modules
  - Allows for manipulation of the tree
    - Traversal
    - Interrogation, deletion, and insertion of nodes
  - Converts the tree to a serial stream (file, socket, or byte stream)
DOM Level 0

- DOM Level 0 was an effort to provide access to selected components of HTML documents – and the browser.
- The document was parsed and a tree model of critical elements – forms, input elements, anchors, etc. was built.
- The “API” was in essence a series of allowed javascript and vb script calls

Evolution of the DOM

- The DOM has evolved from an informal model developed for Netscape Navigator and Internet Explorer to aid in the manipulation of HTML documents. This DOM is referred to as DOM Level 0
- DOM Level 1 represents a standardization effort for DOM Level 0 and includes some support for XML as well as HTML documents
- DOM Level 2 represents the beginning of a real DOM and notably includes namespace support
- DOM Level 3 provides a variety of utility functions

DOM Level 2 Modules

- There are four primary modules at DOM Level 2:
  - Core defines the node types and exceptions
  - XML defines the subinterfaces for XML
  - HTML defines the HTML specific interfaces
  - Traversal defines the interfaces for walking trees or processing node sets
- There are additional modules for:
  - Events (HTML, mouse, user interface, document mutation, range)
  - Views (Style sheets, CSS1, CSS2)
The State of DOM Compliance

• There are many different parsers, each of which will have its idiosyncrasies.
• The degree of its compliance with the DOM specification, both level and completeness will vary.
• At this point in time:
  • Most parsers are namespace aware
  • Most parsers will validate – if only using DTD
  • Many parsers provide weak or no support for schema.
  • Support for serialization is weak
  • Support for styles is weak
  • Support for various events is weak
**JAXP Features**

- JAXP provides some additional functionality in three areas:
  - The parser to be produced can be configured or checked before it is produced
  - The parse method is overloaded to allow a variety of input sources to be read:
    - `InputSource`
    - `File`
    - `URI`
    - `InputStream`
  - The xml Transformer factory provides a mechanism for serializing DOM objects (It can do much more, but it serves the DOM 3 load and save functions.)

---

**Configuration Options**

- The `DocumentBuilderFactory` has options that allow the parser to be configured:
  - `setCoalescing` determines if text nodes and CDATA sections to be merged (default false)
  - `setExpandEntityReferences` determines if entity references are expanded (default false)
  - `setIgnoreComments` determines if comments are processed (default false)
  - `setIgnoringElementWhitespace` determines if whitespace between elements is ignored (default false)
  - `setNamespaceAware` determines if the parser is namespace aware (default false)
  - `setValidating` determines if the parser will validate the document (default false)

---

**Core Module Interfaces**

- The core module defines a series of interfaces, which we break down into two sets:
  - A collection of utility interfaces
    - `DOMException`
    - `DOMImplementation`
    - `NodeList`
    - `NamedNodeMap`
  - A set of interfaces that inherit from the `Node` interface
    - `Element`
    - `Text`
    - `Document`
    - `CharacterData`
    - `Attr`
    - `Comment`
    - `DocumentFragment`
The “Tree” – Parents and Children

- Not all nodes are created alike in the DOM.
- In accord with DOM 2, all nodes that need to be namespace aware – this means that they have a local name, a namespace prefix, a namespace URI, and a full name.
- Further, nodes, as appropriate have parents, children, and siblings
- Further, nodes, as appropriate, have values

The Node Interface (A first look)

- The Node interface is inherited by all the other interfaces.
- It has four categories of methods:
  - Properties
    - Gets and set the name, value, prefix, etc.
  - Navigation
    - Gets related nodes (parent, children, sibling, etc.)
  - Manipulation
    - Inserts, removes, appends nodes
  - Utilities
    - Cloning (individual node or branch), normalizing, etc.

The DOMImplementation Interface

- The DOMImplementation is accessed through the builder obtained from the factory
- The DOMImplementation Interface provides three methods:
  - createDocumentType – which is used to associate a Schema or DTD with a type
  - createDocument – which is used to create a document of a given type
  - hasFeature
- The features are queried using defined strings and version numbers
Checking Parser Features


DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
try{
    DocumentBuilder builder = factory.newDocumentBuilder();
    DOMImplementation implementation = builder.getDOMImplementation();
    for (int j=1;j<=3;j++){
        Integer Version = new Integer(j);
        for (int i = 0; i < features.length; i++) {
            if (implementation.hasFeature(features[i],Version.toString()+".0")) {
                System.out.println("Version " +Version.toString()+" supports " + features[i]);
            } else {
                System.out.println("Version " + Version.toString()+ " does not support " + features[i]);
            }
        } //for i
    } //for j
} //try
catch (ParserConfigurationException pce) {
pce.printStackTrace();}

Parsing a Document

DOMImplementation implementation;
DocumentBuilder builder;
Document dom;
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setValidating(true);
try{
    builder = factory.newDocumentBuilder();
    implementation = builder.getDOMImplementation();
    dom = builder.newDocument();
    dom = builder.parse("othello.xml");
    System.out.println("Parse successful");
} catch (ParserConfigurationException pce) {
pce.printStackTrace(); } catch (SAXException se) {
    se.printStackTrace(); } catch (IOException ioe) {
    ioe.printStackTrace(); }

Node Type Names And Values

<table>
<thead>
<tr>
<th>Interface</th>
<th>nodeType</th>
<th>nodeValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>node</td>
<td>node.name</td>
</tr>
<tr>
<td>Node/Attribute</td>
<td>attribute</td>
<td>attribute.value</td>
</tr>
<tr>
<td>Text</td>
<td>text</td>
<td>text.value</td>
</tr>
<tr>
<td>DocumentFragment</td>
<td>fragment</td>
<td>fragment.text</td>
</tr>
<tr>
<td>DocumentType</td>
<td>type</td>
<td>type.name</td>
</tr>
<tr>
<td>Entity</td>
<td>entity</td>
<td>entity.value</td>
</tr>
<tr>
<td>EntityReference</td>
<td>reference</td>
<td>reference.name</td>
</tr>
<tr>
<td>EntityReference</td>
<td>entity</td>
<td>entity.value</td>
</tr>
<tr>
<td>ProcessingInstruction</td>
<td>instruction</td>
<td>instruction.target</td>
</tr>
<tr>
<td>Text</td>
<td>text</td>
<td>text.value</td>
</tr>
</tbody>
</table>

April 5, 2007 XML Processing
Processing a DOM Object

```java
builder = factory.newDocumentBuilder();
implementation = builder.getDOMImplementation();
dom = builder.parse("othello.xml");
NodeIterator it = implementation.createNodeIterator(dom, NodeFilter.SHOW_TEXT, null, true);
Pattern p = Pattern.compile(".*the.*");
while ((tn = it.nextNode()) != null) {
    if (!tn.getParentNode().getNodeName().equals("LINE")) continue;
    String cont = tn.getNodeValue();
    Matcher m = p.matcher(cont);
    if (m.matches()) {
        int i = 0;
        NodeList nl = tn.getParentNode().getChildren();
        while (nl.item(i).getNodeName().equals("#text")) i++;
        System.out.println(nl.item(i).getNodeValue());
    } // if matches
} // while tn
```

Invoking DOM

- To build a document, minus the catch try blocks and the imports, the code would be:
  ```java
  DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
  dbf.setValidating(true); // in general configure parser
  DocumentBuilder db = dbf.newDocumentBuilder();
  Document doc = db.newDocument();
  ```

- To read a document, leaving for a second the nature of the error handler:
  ```java
  DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
  dbf.setValidating(true); // in general configure parser
  DocumentBuilder db = dbf.newDocumentBuilder();
  Db.setErrorHandler(new MyErrorHandler());
  db.parse(new File("xyz.xml"));
  ```

The DOM Parser Error Handler

- If the document builder sets its error handler to "null", the underlying default implementation will be used.
- The user may write their own error handler by extending the class ErrorHandler and providing for three methods:
  - fatalError() – errors that violate XML1.0 and halt processing
  - error() – errors that violate validity constraints but do not stop processing
  - warning() – neither of the above, and do not stop processing
Selected Interfaces

- Under DOM3 the document interface, which provides access to the root element of the document will provide an extended set of functions. For now, most of what the document interface provides is also provided by the element interface.
- Several other subinterfaces of node provide specialized functions:
  - CharacterData
  - Text
  - Comment
  - CDATASection
  - Attr
  - ProcessingInstruction
- The functionality of these interfaces is briefly described on the following slides.

Element Interface Methods

- The most “important” element method is:
  - getElementsByTagName(String name) which returns a NodeList of elements
- There are also a series of methods for getting and setting the attributes of an element
  - The Node interface defines a method called getAttributes() which returns a NamedNodeMap. (The method returns null for all but element nodes.) The NamedNodeMap interface provides accessor and setter methods.

CharacterData Interface Methods

- Void appendData(String arg) appends string to the end of the character data of the node.
- Void deleteData(int offset, int count) removes a range of 16-bit units from the node.
- String getData() The character data of the node that implements this interface.
- Int getLength() The number of 16-bit units that are available through data and the substringData method below.
- Void insertData(int offset, String arg) Insert a string at the specified 16-bit unit offset.
- Void replaceData(int offset, int count, String arg) Replace the characters starting at the specified 16-bit unit offset with the specified string.
- Void setData(String data) The character data of the node that implements this interface.
- String substringData(int offset, int count) Extracts a range of data from the node.
Attr Interface Methods

• The Attr interface helps with a couple particular problems:
  • It allows the element to which the attribute belongs to be identified – getParent won’t work and therefore the method getOwnerElement() can be used.
  • The getSpecified method allows us to determine whether the given attribute was specified in the document or provided by the parser.

NodeList and NamedNodeMap

• The NodeList interface is very simple
  • int getLength() provides the number of nodes in the list.
  • Node item(int index) returns the indexth item in the collection.
• The NamedNodeMap interface is a little more complex (This is only a partial list)
  • int getLength() provides the number of nodes in this map.
  • Node getNamedItem(String name) Retrieves a node specified by name.
  • Node item(int index) Returns the indexth item in the map.
  • Node removeNamedItem(String name) Removes a node specified by name.

DOM Traversal Module

• Most DOM parsers implement the traversal module which provides access to two basic types of traversal:
  • The NodeIterator provides access to a linear list of nodes
  • The TreeWalker provides access to the tree
• The NodeIterator and TreeWalker are created via the documentTraversal interface invoked by casting the DOM document DocumentTraversal
• Both of these interfaces provide for control of the node set returned and processed through the use of NodeFilters
Constructors

- Assuming the Document has been cast to a DocumentTraversal type document:
  - NodeIterator createNodeIterator(Node root, int whatToShow, NodeFilter filter, boolean entityReferenceExpansion) creates a new NodeIterator over the subtree rooted at the specified node.
  - TreeWalker createTreeWalker(Node root, int whatToShow, NodeFilter filter, boolean entityReferenceExpansion) creates a new TreeWalker over the subtree rooted at the specified node.

- Two levels of control are provided
  - WhatToShow is a constant which identifies node type
  - NodeFilter is a subclass that allows more refined control

NodeFilter

- The NodeFilter interface has one method and a series of constants. The method is acceptNode();

```java
import org.w3c.dom.traversal.NodeFilter;
import org.w3c.dom.*;
public class MyFilter implements NodeFilter {
    public short acceptNode(Node node) {
        Element candidate = (Element) node;
        String type = candidate.getTagName();
        if (type.equals("A")) return FILTER_ACCEPT;
        return FILTER_SKIP;
    }
}
```

Using DOM To Build a Document

```java
Document d = builder.newDocument();
Element root = d.createElement("root");
d.appendChild(root);
Comment c = d.createComment("This is a comment");
root.appendChild(c);
Element p = d.createElement("person");
Element n = d.createElement("name");
n.appendChild(d.createTextNode("John Doe"));
s.appendChildTextNode("123-45-6789");
p.appendChild(n);
p.appendChild(s);
root.appendChild(p);
```
**Selected TreeWalker Methods**

- **Node firstChild()**: Moves to the first visible child of the current node, and returns the new node.
- **Node getCurrentNode()**: The current node.
- **Node getRoot()**: Moves to the root node as specified at creation.
- **Node nextNode()**: Moves the next visible node in document order relative to the current node.
- **Node nextSibling()**: Moves to the next sibling of the current node, and returns the new node.
- **Node parentNode()**: Moves to and returns the closest visible ancestor node of the current node.
- **Node previousNode()**: Moves to the previous visible node in document order relative to the current node, and returns the new node.
- **Node previousSibling()**: Moves to the previous sibling of the current node, and returns the new node.

**Using JAXP/DOM To Read and Process a Document**

```java
InputSource source = new InputSource(new FileInputStream("mymessage.xml"));
Document doc = builder.parse(source);
// assuming the document looks as follows:
// <message to = "jon@pitt.edu" from = "pat@cmu.edu" >
//A message for jon from pat
//</message>
Element root = doc.getDocumentElement();
if ( !root.getTagName().equals( "message" ) )
{ // some error handling routine; return;}
String from = root.getAttribute( "from" );
String to = root.getAttribute( "to" );
String text = root.getFirstChild().getNodeValue();
// send message to corresponding user
processmail(to, from, text);
```

**Using JAXP/DOM To Write a Document**

```java
try{
    Document message = builder.newDocument();
    Element root = filelock.createElement("lockdocument");
    root.setAttribute("locktype", "Locked");
    root.setAttribute("validate", "true");
    root.setAttribute("editor", "somename");
    root.appendChild(filelock.createTextNode("some text"));
    message.appendChild(root);
    OutputStream out = clientSocket.getOutputStream();
    TransformerFactory TF = TransformerFactory.newInstance();
    Transformer serial = TF.newTransformer();
    serial.transform( new DOMSource( message ), new StreamResult( output ) );
    output.flush();
}catch ( Exception e ) {
    e.printStackTrace();
}
```
Some Code Snippets

- The following slides provide a conceptual overview and a few pieces of code from a client server application for collaborative authoring.
- The code is written in Java, uses threads, and uses:
  - Dave Meggison’s crimson classes
  - SUN’s jaxp
  - W3C xerces parsers
  - SUN xlan parsers
The Server

- The server simply logs and keeps track of the activity of the clients
- A separate set of threads handles communications among the various clients

Message Construction

- For the application as a whole
  ```java
  try { // obtain the default parser
      factory = DocumentBuilderFactory.newInstance();
      // get DocumentBuilder
      builder = factory.newDocumentBuilder();
  } catch (ParserConfigurationException pce) { pce.printStackTrace(); }
  ```
- To construct a simple document to be sent
  ```java
  Document login = builder.newDocument();
  Element root = login.createElement("user");
  login.appendChild(root);
  lp.set_tf_name(user);
  root.appendChild(login.createTextNode(user));
  send(login);
  ```

Sending a message

```java
public void send( Document message )
{ byte end[]={0,0}; byte mt[]={1,1};
  try {
      // write to output stream
      output.write(mt); //1 indicates a text message
      TransformerFactory transformerFactory = TransformerFactory.newInstance();
      Transformer serializer = transformerFactory.newTransformer();
      serializer.transform( new DOMSource( message ),
                          new StreamResult( output ) );
      output.write(end);
      output.flush();
  } catch (Exception e) { e.printStackTrace(); }
}```
Message Routing

Element root = message.getDocumentElement();
if ( root.getTagName().equals( "user" ) )
    server.checkNewUser( this , sept, message);
else if ( root.getTagName().equals( "message" ) )
    server.sendMessage( message );
else if ( root.getTagName().equals( "updateusers" ) )
    server.updateUsers();
else if ( root.getTagName().equals( "docStatus" ) )
    sept.send(server.docStatusRequest());
else if ( root.getTagName().equals( "lockdocument" ) )
    server.docLockRequest(message);

Processing a DOM to create a JTree

public JTree displayroot() {
    JTree doctree = new JTree;
    doctree.setShowsRootHandles(true);
    doctree.setVisible(true);
    ldp.add(doctree, BorderLayout.CENTER);
    return doctree;
}

private void insertchildren(NodeList nlist, String base, int parent){
    for (int i =0; i<nlist.getLength();i++){
        int cn=nn;
        Node localn = nlist.item(i);
        if (localn.getNodeType() == Node.ELEMENT_NODE){
            insertchildren(localn.getChildNodes(), setbase(base, i),cn);
        }else if (localn.getNodeType()==Node.TEXT_NODE){
            }else if (localn.getNodeType()==Node.COMMENT){
            }else if (localn.getNodeType()==Node.PROCESSING_INSTRUCTION){
            }else if (localn.getNodeType()==Node.DOCUMENTTYPE){
            }else if (localn.getNodeType()==Node.DOCUMENT_FRAGMENT){
            }else if (localn.getNodeType()==Node.ENTITY_REFERENCE){
            }else if (localn.getNodeType()==Node.ENTITY_DECLARATION){
            }else if (localn.getNodeType()==Node.NOTATION_DECLARATION){
            }
    }