# Document Object Model

## DOM

DOM is a programming interface that provides a way for the values and structure of an XML document to be accessed and manipulated.

### Tasks that can be performed with DOM

- Navigate an XML document's structure, which is a tree stored in memory.
- Report the information found at the nodes of the XML tree.
- Add, delete, or modify elements in the XML document.

DOM represents each node of the XML tree as an object with properties and behavior for processing the XML.

The root of the tree is a Document object. Its children represent the entire XML document except the xml declaration.

On the next page we consider a small XML document with comments, a processing instruction, a CDATA section, entity references, and a DOCTYPE declaration, in addition to its element tree.

It is valid with respect to a DTD, named *root.dtd*.

```
<!ELEMENT root (child*)>
<!ELEMENT child (name)>
<!ELEMENT name (#PCDATA)>
<!ATTLIST child position NMTOKEN #REQUIRED>
<!ENTITY last1 "Dover">
<!ENTITY last2 "Reckonwith">
```

## Example: root.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE root SYSTEM "root.dtd">
<!-- root.xml -->
<?DomParse usage="java DomParse root.xml"?>
<root>
  <child position="first">
     <name>Eileen &last1;</name>
  </child>
  <child position="second">
     <name><![CDATA[<<<Amanda>>>]]> &last2;</name>
  </child>
<!-- Could be more children later. -->
</root>
```

DOM imagines that this XML information has a *document root* with four children:

1. A DOCTYPE declaration.
2. A comment.
3. A processing instruction, whose *target* is DomParse.
4. The *root* element of the document.

The second comment is a child of the *root* element.

To process this document tree, DOM provides names to describe the different kinds of nodes that may appear in the tree.

Java represents these various kinds of nodes using interfaces and constants. The node interfaces are all subinterfaces of the interface org.w3c.dom.Node.

# The Node Hierarchy

All of these interfaces are found in the package *org.w3c.dom*.

```
Node
   |– Attr
   |– Document
   |– DocumentFragment
   |– DocumentType
   |– Entity
   |– Element
   |– EntityReference
   |– Notation
   |– ProcessingInstruction
   |– CharacterData
         |– Comment
         |– Text
               |– CDATASection
```

The interfaces inherit methods from the Node interface. These methods allow us to navigate the document tree and extract information from it.

String getNodeName()

String getNodeValue()

**short** getNodeType()

Node getFirstChild()

Node getLastChild()

Node getNextSibling()

Node getPreviousSibling()

NodeList getChildNodes()

NamedNodeMap getAttributes()

## Interface Names and Their Constants

| Interface | Constant | short |
|---|---|---|
| Element | ELEMENT_NODE | 1 |
| Attr | ATTRIBUTE_NODE | 2 |
| Text | TEXT_NODE | 3 |
| CDATASection | CDATA_SECTION_NODE | 4 |
| EntityReference | ENTITY_REFERENCE_NODE | 5 |
| Entity | ENTITY_NODE | 6 |
| ProcessingInstruction | PROCESSING_INSTRUCTION_NODE | 7 |
| Comment | COMMENT_NODE | 8 |
| Document | DOCUMENT_NODE | 9 |
| DocumentType | DOCUMENT_TYPE_NODE | 10 |
| DocumentFragment | DOCUMENT_FRAGMENT_NODE | 11 |
| Notation | NOTATION_NODE | 12 |

Normally we use the constants Node.ELEMENT_NODE and Node.TEXT_NODE in place of the numeric values 1 and 3.

## Node Interfaces, Their Names, and Their Values

| **Interface** | **nodeName()** | **nodeValue()** |
|---|---|---|
| Attr | Name of attribute | Value of attribute |
| Element | Tag name | **null** |
| Text | Content of text node | **null** |
| CDATASection | #cdata-section | Content of CDATA section |
| EntityReference | Name of entity reference | **null** |
| Entity | Entity name | **null** |
| ProcessingInstruction | Target | Entire content excluding target |
| Comment | #comment | Content of comment |
| Document | #document | **null** |
| DocumentType | Document type name | **null** |
| DocumentFragment | #document-fragment | **null** |
| Notation | Notation name | **null** |

# Methods for Inspecting DOM a Tree

Each of the twelve node interfaces inherits the methods from the Node interface, even if they do not make sense for that particular kind of node. The methods just return **null** in this case.

In addition each interface has additional methods that allow us to inspect the nodes and values in the DOM tree.

Interfaces not mentioned here rely on inherited methods mostly.

### Methods in Element

String getTagName()

NamedNodeMap getAttributes()

String getAttribute(String name)

Attr getAttributeNode(String name)

NodeList getElementsByTagName(String name)

**boolean** hasAttributes()

**boolean** hasAttribute(String name)

### Methods in Attr

String getName()

String getValue()

Element getOwnerElement()

Although Attr is a subinterface of Node, these objects are not actually child nodes of the element they describe and they are not considered part of the DOM tree.

They have no parent or sibling nodes in the tree.

Attributes are thought of as properties of elements rather than nodes themselves.

The value of an attribute may be a default value obtained from a dtd.

### Methods In CharacterData

Inherited by Text, Comment, and CDATASection

**int** getLength()

String getData()

### Method in ProcessingInstruction

String getTarget()

String getData()

### Methods in DocumentType

String getName()     // name of DTD root

NamedNodeMap getEntities()     // only general entities

NamedNodeMap getNotations()

String getSystemId()

String getPublicId()

This node has no children, but methods allow us to fetch entities and notations from the dtd it represents.

### Methods in NodeList

**int** getLength()

Node item(**int** index)

### Methods in NamedNodeMap

**int** getLength()

Node item(**int** index)

Node getNamedItem(String name)

### Method in DocumentBuilderFactory

DocumentBuilder newDocumentBuilder()
**throws** ParserConfigurationException

## Displaying a DOM Tree

In the next program we display the information uncovered in a DOM tree by using the interface methods to perform a depth-first traversal of the tree.

At each node, pertinent information about its values and content is printed. Since white space can be overlooked easily in an XML tree, textual spaces in values and content are converted to plus signs (+). In addition new line characters become "[nl]" and tab characters become "[tab]".

In this first version, the parse is executed with default settings. Afterward we consider changing some of the parser settings.

A parameter to the method *traverse* is used to provide spaces so that the structure of the tree is shown by indenting.

## The Program: DomParse.java

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.ParserConfigurationException;

import org.xml.sax.SAXException;

import org.w3c.dom.Node;
import org.w3c.dom.Attr;
import org.w3c.dom.CDATASection;
import org.w3c.dom.Comment;
import org.w3c.dom.Document;
import org.w3c.dom.DocumentType;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.ProcessingInstruction;
import org.w3c.dom.Text;

import java.io.*;
```

```
public class DomParse
{
   public static void main(String [] args)
   {
      if (args.length != 1)
         System.out.println("Usage: java DomParse file.xml");
      else
      {
         File file = new File(args[0]);
         DocumentBuilderFactory factory =
                  DocumentBuilderFactory.newInstance();
         try
         {
            DocumentBuilder builder =
                        factory.newDocumentBuilder();
            builder.setErrorHandler(new MyErrorHandler());
            Document doc = builder.parse(file);
            traverse(doc, "");
         }
         catch (SAXException e)
         { System.out.println(e); }
         catch (ParserConfigurationException e)
         { System.out.println(e); }
         catch (IOException e)
         { System.out.println(e); }
      }
   }

   static void traverse(Node node, String indent)
   {
      switch (node.getNodeType())
      {
         case Node.DOCUMENT_NODE:
         {
            System.out.println(indent + node.getNodeName());
            break;
         }
```

```
case Node.ELEMENT_NODE:
{
   Element elem = (Element)node;
   System.out.println(indent + "ELEMENT: "
                               + elem.getTagName());
   NamedNodeMap nm = elem.getAttributes();
   for (int k=0; k<nm.getLength(); k++)
      traverse(nm.item(k), indent + "   ");
   break;
}

case Node.ATTRIBUTE_NODE:
{
   Attr at = (Attr)node;
   System.out.println(indent + "ATTRIBUTE:  " +
            at.getName() + "=\"" + at.getValue() + "\"");
   break;
}

case Node.TEXT_NODE:
{
   Text tx = (Text)node;
   System.out.println(indent + "TEXT (length=" +
                      tx.getLength() + ") \"" +
                      replace(tx.getData() + "\""));
   break;
}

case Node.COMMENT_NODE:
{
   Comment cm = (Comment)node;
   System.out.println(indent + "COMMENT (length=" +
                      cm.getLength() + ") \"" +
                      replace(cm.getData() + "\""));
   break;
}
```

```
    case Node.CDATA_SECTION_NODE:
    {
      CDATASection cds = (CDATASection)node;
      System.out.println(indent + "CDATA SECTION (length="
                      + cds.getLength() + ") \"" +
                      replace(cds.getData() + "\""));
      break;
}

    case Node.PROCESSING_INSTRUCTION_NODE:
    {
      ProcessingInstruction pi =
                      (ProcessingInstruction)node;
      System.out.println(indent + "PI: target="
                              +  pi.getTarget());
      System.out.println(indent + replace(pi.getData()));
      break;
    }

    case Node.DOCUMENT_TYPE_NODE:
    {
      DocumentType dt = (DocumentType)node;
      System.out.println(indent + "DOCUMENT_TYPE: "
                              + dt.getName());
      if (dt.getPublicId() != null)
          System.out.println(indent + "Public ID: "
                          + dt.getPublicId());
      if (dt.getSystemId() != null)
          System.out.println(indent + "System ID: "
                          + dt.getSystemId());
      NamedNodeMap entities = dt.getEntities();
      for (int k=0; k<entities.getLength(); k++)
          traverse(entities.item(k), indent + "  ");
      break;
    }
```

```
            case Node.ENTITY_NODE:
            {
                System.out.println(indent + "ENTITY:  "
                                                + node.getNodeName());
                break;
            }

            case Node.ENTITY_REFERENCE_NODE:
            {
                System.out.println(indent + "ENTITY REFERENCE:  "
                                                + node.getNodeName());
                break;
            }
        }

        NodeList list = node.getChildNodes();

        for (int k=0; k<list.getLength(); k++)
            traverse(list.item(k), indent + "   ");
    }

    static String replace(String s)
    {
            if (s != null)
            {
                  s = s.replaceAll(" ", "+");
                  s = s.replaceAll("\n", "[nl]");
                  s = s.replaceAll("\t", "[tab]");
                  return s;
            }
            return "";
    }
}
```

Observe that the ErrorHandler for the DocumentBuilder object has been set in the same way as in DomCheck.java, but at this point we have not requested validation of the XML document.

Next we execute the DOM parser using the XML document *root.xml*.

```
% java DomParse root.xml
```

```
#document
  DOCUMENT_TYPE: root
  System ID: root.dtd
   ENTITY: last1
     TEXT (length=5) "Dover"
   ENTITY: last2
     TEXT (length=10) "Reckonwith"
  COMMENT (length=10) "+root.xml+"
  PI: target=DomParse
  usage="java+DomParser+root.xml"
  ELEMENT:  root
    TEXT (length=4) "[nl]+++"
    ELEMENT:  child
      ATTRIBUTE:   position="first"
        TEXT (length=5) "first"
      TEXT (length=8) "[nl]+++++++"
      ELEMENT: name
        TEXT (length=12) "Eileen+ Dover"
      TEXT (length=4) "[nl]+++"
    TEXT (length=4) "[nl]+++"
    ELEMENT: child
      ATTRIBUTE: position="second"
        TEXT (length=6) "second"
      TEXT (length=8) "[nl]+++++++"
      ELEMENT: name
        CDATA SECTION  (length=12) "<<<Amanda>>>"
        TEXT (length=11) "+Reckonwith"
      TEXT (length=4) "[nl]+++"
    TEXT (length=1) "[nl]"
    COMMENT (length=31) "+Could+be+more+children+later.+"
    TEXT (length=1) "[nl]"
```

## Entity References

Observe that no entity references appear in the output obtained from parsing the XML document *root.xml*, which clearly contains two entity references, *&last1;* and *&last2;*.

The DocumentBuilderFactory object that we created recognizes a number of options that can be used to configure the parser.

These options are specified by six instance methods understood by a DocumentBuilderFactory object.

| **Instance Method** | **Default** |
|---|---|
| setIgnoreElementContentWhitespace | false |
| setValidating | false |
| setCoalescing | false |
| setExpandEntityReferences | true |
| setIgnoringComments | false |
| setNamespaceAware | false |

To leave the entity references unresolved, add the following code immediately after the factory is created.

```
factory.setExpandEntityReferences(false);
```

The tree starting with the first *child* element is show below. Now the entity references appear. The value of the entity is shown as a text node child of the entity reference.

```
ELEMENT: child
  ATTRIBUTE:  position="first"
    TEXT (length=5) "first"
  TEXT (length=8) "[nl]+++++++"
```

```
 ELEMENT: name
    TEXT (length=7) "Eileen+"
    ENTITY REFERENCE:  last1
      TEXT (length=5) "Dover"
  TEXT (length=4) "[nl]+++"
TEXT (length=4) "[nl]+++"
ELEMENT: child
  ATTRIBUTE: position="second"
    TEXT (length=6) "second"
  TEXT (length=8) "[nl]+++++++"
  ELEMENT: name
    CDATA SECTION (length=12) "<<<Amanda>>>"
    TEXT (length=1) "+"
    ENTITY REFERENCE: last2
      TEXT (length=10) "Reckonwith"
  TEXT (length=4) "[nl]+++"
    TEXT (length=1) "[nl]"
    COMMENT (length=31) "+Could+be+more+children+later.+"
    TEXT (length=1) "[nl]"
```

## Ignorable Whitespace

The white space characters that occur between different element tags is known as ignorable whitespace.

Consider this fragment from *root.xml*.

```
<root>
  <child position="first">
    <name>Eileen &last1;</name>
  </child>
```

The characters between <root> and <child position="first"> play no significant role in the meaning of the document.

But the DOM parser preserves these characters by default.

```
ELEMENT: root
  TEXT (length=4) "[nl]+++"
  ELEMENT: child
```

To remove ignorable white space from the DOM tree, several steps are required.

1. The XML document needs a DOCTYPE declaration that specifies a dtd.
2. The factory needs to know that the parser to be created will be a validating parser.

   ```
   factory.setValidating(true);
   ```

3. The XML document must be valid with respect to the dtd.
4. The factory needs to know that the parser should ignore these white space characters.

   ```
   factory.setIgnoringElementContentWhitespace(true);
   ```

Now the DOM tree displayed by the parser shows none of this extraneous white space.

```
#document
  DOCUMENT_TYPE: root
  System ID: root.dtd
   ENTITY: last1
     TEXT (length=5) "Dover"
   ENTITY: last2
     TEXT (length=10) "Reckonwith"
  COMMENT (length=10) "+root.xml+"
  PI: target=DomParse
  usage="java+DomParser+root.xml"
  ELEMENT: root
    ELEMENT: child
      ATTRIBUTE: position="first"
        TEXT (length=5) "first"
      ELEMENT: name
        TEXT (length=12) "Eileen+Dover"
```

```
    ELEMENT:  child
      ATTRIBUTE:  position="second"
        TEXT (length=6) "second"
      ELEMENT:  name
        CDATA SECTION (length=12) "<<<Amanda>>>"
        TEXT (length=11) "+Reckonwith"
    COMMENT (length=31) "+Could+be+more+children+later.+"
```

## Coalescing CDATA and Removing Comments

The last options we consider deal with the CDATA sections and the comments in the XML document.

If we ask the factory object to build a parser that coalesces the CDATA sections, then these sections will be merged with adjacent text, either from text nodes or other CDATA sections.

```
          factory.setCoalescing(true);
```

Finally, we can ask the parser to remove all comments in the XML document.

```
          factory.setIgnoringComments(true);
```

### The Result

```
#document
  DOCUMENT_TYPE:  root
  System ID: root.dtd
   ENTITY:  last1
     TEXT (length=5) "Dover"
   ENTITY:  last2
     TEXT (length=10) "Reckonwith"
  PI: target=DomParse
  usage="java+DomParser+root.xml"
```

```
ELEMENT:  root
  ELEMENT:  child
     ATTRIBUTE:  position="first"
       TEXT (length=5) "first"
    ELEMENT:  name
       TEXT (length=12) "Eileen+Dover"
  ELEMENT:  child
     ATTRIBUTE: position="second"
       TEXT (length=6) "second"
    ELEMENT:  name
       TEXT (length=23) "<<<Amanda>>>+Reckonwith"
```

## Extracting Information from an XML Document

Now we look at a series of programs that parse an XML document and build Java objects with the information found.

The document is a variation of the phone listing we considered earlier. One element can have an attribute and the *city* element is optional in addition to the *middle* element.

### DTD: phoneA.dtd

```
<!ELEMENT phoneNumbers (title, entries)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT entries (entry*)>
<!ELEMENT entry (name, phone, city?)>
<!ELEMENT name (first, middle?, last)>
<!ATTLIST name gender (female | male) #IMPLIED>
<!ELEMENT first (#PCDATA)>
<!ELEMENT middle (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT city (#PCDATA)>
```

## XML Document: phoneA.xml

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE phoneNumbers SYSTEM "phoneA.dtd">
<phoneNumbers>
  <title>Phone Numbers</title>
  <entries>
   <entry>
     <name>
       <first>Rusty</first>
       <last>Nail</last>
     </name>
     <phone>335-0055</phone>
     <city>Iowa City</city>
   </entry>
   <entry>
     <name gender="male">
       <first>Justin</first>
       <last>Case</last>
     </name>
     <phone>354-9876</phone>
     <city>Coralville</city>
   </entry>
   <entry>
     <name gender="female">
       <first>Pearl</first>
       <middle>E.</middle>
       <last>Gates</last>
     </name>
     <phone>335-4582</phone>
     <city>North Liberty</city>
   </entry>
   <entry>
     <name gender="female">
       <first>Helen</first>
       <last>Back</last>
     </name>
     <phone>337-5967</phone>
   </entry>
  </entries>
</phoneNumbers>
```

## Java Classes

To build appropriate objects we need two classes to describe the data in the XML document.

```
class Entry
{
    private Name name;
    private String gender, phone, city;

    Entry(Name n, String g, String p, String c)
    {
        name = n;       gender = g;
        phone = p;      city = c;
    }

    Entry(Name n, String p, String c)
    {  this(n, null, p, c);  }

    Entry(Name n, String p)
    {  this(n, null, p, null);  }

    Name getName()
    {  return name;  }

    String getGender()
    {  return gender;  }

    String getPhone()
    {  return phone;  }

    String getCity()
    {  return city;  }
    public String toString()
    {
        if (gender != null && gender.length() > 0)
            gender = "gender = " + gender + "\n";
        else gender = "";
        if (city != null && city.length()>0)
            city = city + "\n";
        else city= "";
        return name + "\n" + gender + phone + "\n" + city;
    }
}

class Name
{
    private String first, middle, last;

    Name(String f, String m, String lt)
    {
        first = f;  middle = m; l ast = lt;
    }

    Name(String f, String lt)
    {
        this(f, null, lt);
    }

    String getFirst()
    {  return first;  }

    String getMiddle()
    {  return middle;  }

    String getLast()
    {  return last;  }

    public String toString()
    {
        if (middle == null || middle.equals(""))
            return first + " " + last;
        else
            return first + " " + middle + " " + last;
    }
}
```

## Document

In these programs we use several instance methods belonging to the Document interface. They are list below with a couple of other useful methods from this interface.

```
Element getDocumentElement()

NodeList getElementsByTagName(String tag)

DocumentType getDocType()

DOMImplementation getImplementation()
```

The method *getElementsByTagName* returns all elements in the entire document with that tag name.

## Version 1: PhoneParser.java

In this version we use a method *getEntries* to search the grandchildren of the root element, looking for the *entry* elements. Remember that this set of grandchildren includes a number of nodes that consist of ignorable white space.

When we find an *entry* element, we call a method *getEntry* that searches its children for *name*, *phone*, and *city* elements. This method builds an Entry object with the data found.

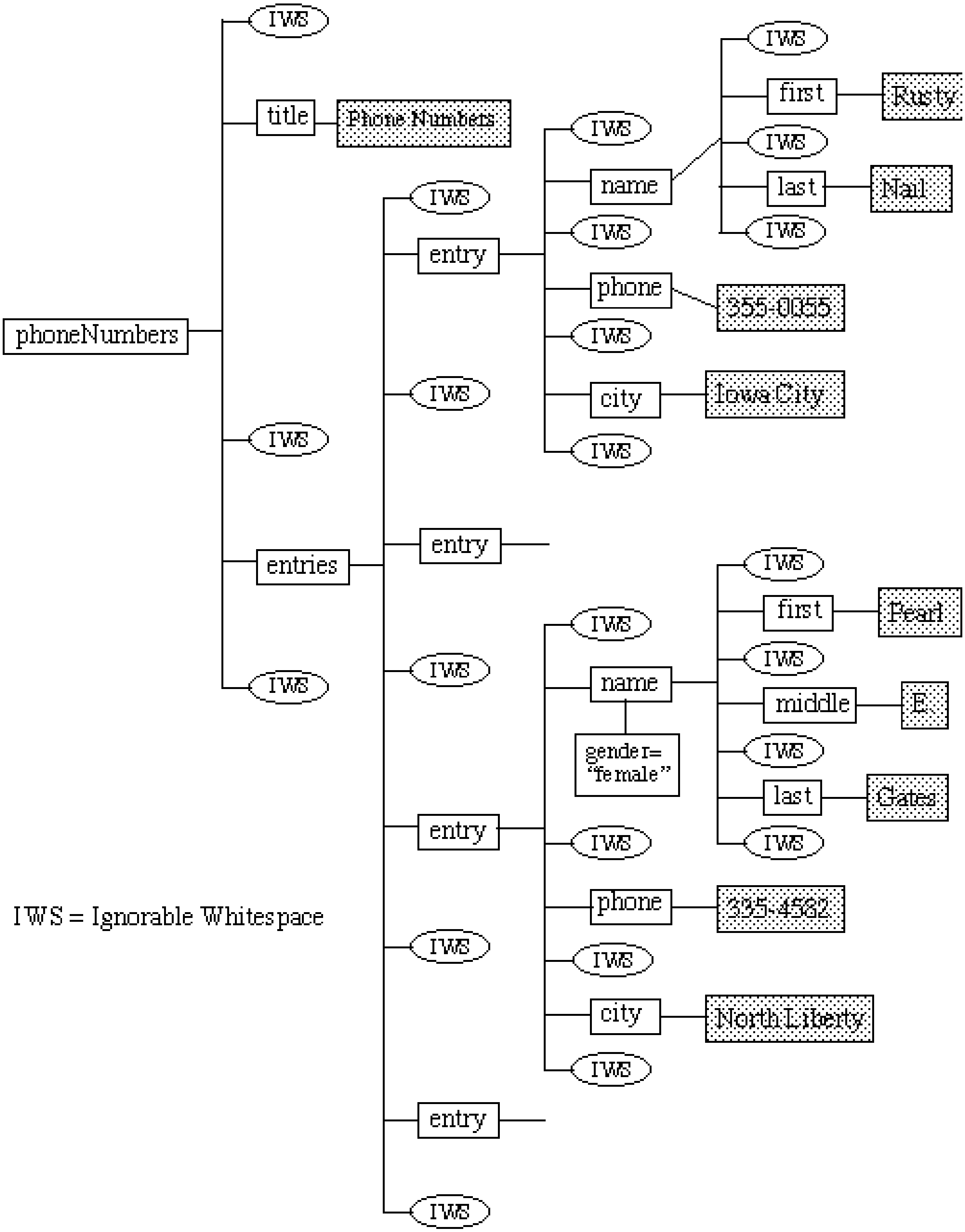
When we find a *name* element, we call a method *getName* that searches its children for *first*, *middle*, and *last* element and looks for a possible *gender* attribute. This method builds a Name object with the string values found.

The *getEntries* method builds an ArrayList of Entry objects. Using the generics feature of Java 1.5, the ArrayList is declared to hold only Entry objects using the syntax:

```
ArrayList<Entry>
```

When items are retrieved from this ArrayList, they are already Entry objects and need no downcasting.

# Part of phoneA.xml



IWS = Ignorable Whitespace

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Node;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Text;
import org.w3c.dom.Attr;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;
import java.util.ArrayList;
import java.io.*;

public class PhoneParser
{
    private DocumentBuilder builder;

    PhoneParser() throws ParserConfigurationException
    {
        DocumentBuilderFactory factory
                = DocumentBuilderFactory.newInstance();
        builder = factory.newDocumentBuilder();
    }

    ArrayList<Entry> parse(String fileName)
                        throws SAXException, IOException
    {
        File f = new File(fileName);
        Document doc = builder.parse(f);
        Element root = doc.getDocumentElement();
        return getEntries(root);
    }
```

```
private static ArrayList<Entry> getEntries(Element e)
{
   ArrayList<Entry> entries = new ArrayList<Entry>();
   NodeList children = e.getChildNodes();
   for (int k = 0; k < children.getLength(); k++)
   {
      Node childNode = children.item(k);
      NodeList grandChildren =
                                 childNode.getChildNodes();
      for (int m = 0; m < grandChildren.getLength(); m++)
      {
         Node grandChildNode = grandChildren.item(m);
         if (grandChildNode instanceof Element)
         {
            Element element = (Element)grandChildNode;
            if  (element.getTagName().equals("entry"))
            {
               Entry c = getEntry(element);
               entries.add(c);
            }
         }
      }
   }
   return entries;
}

private static Entry getEntry(Element e)
{
   Name name = null;
   String gender = null, phone = null, city = null;
   NodeList children = e.getChildNodes();
   for (int k = 0; k < children.getLength(); k++)
   {
      Node childNode = children.item(k);
      if (childNode instanceof Element)
      {
         Element childElement = (Element)childNode;
         String tagName = childElement.getTagName();
         if (tagName.equals("name"))
         {
            name = getName(childElement);
            gender = childElement.getAttribute("gender");
         }
         else if (tagName.equals("phone"))
         {
            Text textNode =
                           (Text)childElement.getFirstChild();
            phone = textNode.getData();
         }
         else if (tagName.equals("city"))
         {
            Text textNode =
                           (Text)childElement.getFirstChild();
            city = textNode.getData();
         }
      }
   }
   return new Entry(name, gender, phone, city);
}

private static Name getName(Element e)
{
   String first = null, middle = null, last = null;
   NodeList children = e.getChildNodes();
   for (int k = 0; k < children.getLength(); k++)
   {
      Node childNode = children.item(k);
      if (childNode instanceof Element)
      {
         Element childElement = (Element)childNode;
         String tagName = childElement.getTagName();
         Text textNode =
                           (Text)childElement.getFirstChild();
         String data = textNode.getData();
```

```
                if (tagName.equals("first"))
                    first = data;
                else if (tagName.equals("middle"))
                    middle = data;
                else if (tagName.equals("last"))
                    last = data;
            }
        }
        return new Name(first, middle, last);
    }

    public static void main(String [] args) throws Exception
    {
        PhoneParser parser = new PhoneParser();
        ArrayList<Entry> entries = parser.parse("phoneA.xml");
        for (int k = 0; k < entries.size(); k++)
        {
            Entry anEntry = entries.get(k);
            System.out.println(anEntry);
        }
    }
}
```

## Alternative for Command (Java 1.5)

In addition to the generic collection types, Java 1.5 has a new version of the **for** command. Replace the for-loop in the main method with the code:

```
for (Entry anEntry : entries)
    System.out.println(anEntry);
```

Restriction: The loop variable, *anEntry* in this case, is read-only (an R-value, not an L-value).

## Output

The Entry and Name objects each have a *toString* method for the purpose of output.

```
% java PhoneParser
Rusty Nail
335-0055
Iowa City

Justin Case
gender = male
354-9876
Coralville

Pearl E. Gates
gender = female
335-4582
North Liberty

Helen Back
gender = female
337-5967
```

## Version 2: TagPhoneParser.java

In this version we use the method *getElementsByTagName* to combine the *parse* and the *getEntries* methods into one method.

We no longer have to worry about children and grandchildren because we can select all of the *entry* nodes from the tree in one step.

## New parse Method (incorporating getEntries)

```
ArrayList<Entry> parse(String fileName)
                      throws SAXException, IOException
{
    File f = new File(fileName);
    Document doc = builder.parse(f);
    Element root = doc.getDocumentElement();
    ArrayList<Entry> entries = new ArrayList<Entry>();

    NodeList entryNodes =
                  root.getElementsByTagName("entry");
    for (int k = 0; k < entryNodes.getLength(); k++)
    {
        Element element = (Element)entryNodes.item(k);
        Entry c = getEntry(element);
        entries.add(c);
    }
    return entries;
}
```

The output from this version is identical to that from version 1.

## Version 3: VPhoneParser.java

In this version we eliminate the ignorable white space and comments so that the only descendants of the *entries* node are element nodes.

Then we know the second child of *phoneNumbers* is *entries*, the only children of *entries* are *entry* nodes, the children of each *entry* are *name*, *phone*, and *city*, in that order, and the children of *name* are *first*, *middle*, and *last*.

These assumptions reduce the complexity of the code significantly.

We show only those methods that change.

```
VPhoneParser() throws ParserConfigurationException
{
   DocumentBuilderFactory factory
              = DocumentBuilderFactory.newInstance();
   factory.setIgnoringElementContentWhitespace(true);
   factory.setValidating(true);
   factory.setIgnoringComments(true);

   builder = factory.newDocumentBuilder();
}

ArrayList<Entry> parse(String fileName)
                    throws SAXException, IOException
{
   File f = new File(fileName);
   Document doc = builder.parse(f);
   Element root = doc.getDocumentElement();
   return getEntries(root);
}
```

```
private static ArrayList<Entry> getEntries(Element e)
{
    ArrayList<Entry> entries = new ArrayList<Entry>();
    NodeList children = e.getChildNodes();
    Node entriesNode = children.item(1);
    NodeList entryChildren = entriesNode.getChildNodes();
    for (int m = 0; m < entryChildren.getLength(); m++)
    {
        Node entryNode = entryChildren.item(m);
        Element element = (Element)entryNode;
        Entry c = getEntry(element);
        entries.add(c);
    }
    return entries;
}

private static Entry getEntry(Element e)
{
    Name name = null;
    String gender = null, phone = null, city = null;
    NodeList children = e.getChildNodes();
    Element child1 = (Element)children.item(0);
    name = getName(child1);
    gender = child1.getAttribute("gender");
    Element child2 = (Element)children.item(1);
    Text textNode = (Text)child2.getFirstChild();
    phone = textNode.getData();
    Element child3 = (Element)children.item(2);
    if (child3 != null)
    {
        textNode = (Text)child3.getFirstChild();
        city = textNode.getData();
    }
    return new Entry(name, gender, phone, city);
 }
```

```
    private static Name getName(Element e)
    {
        String first = null, middle = null, last = null;
        NodeList children = e.getChildNodes();
        Element child = (Element)e.getFirstChild();
        first = ((Text)child.getFirstChild()).getData();
        if (children.getLength() == 3)
        {
            child = (Element)child.getNextSibling();
            middle = ((Text)child.getFirstChild()).getData();
        }
        child = (Element)child.getNextSibling();
        last = ((Text)child.getFirstChild()).getData();
        return new Name(first, middle, last);
    }
```

Again the output remains the same.

## Version 4: XPhoneParser.java

The last version of the phone parser program uses an addition to Java that appears in version 1.5. This new feature allows a Java program to evaluate any XPath expression.

To extract information from the DOM tree we can define XPath expressions that specify exactly which nodes we want from the tree.

But first we need to look at some of the basics of the XPath language. Many of the details of XPath will be postponed to later, but it is a language that we need to master because it is used heavily in XSLT and XQuery

# An Introduction to XPath

XPath is an expression language.

It has no commands (statements) and no declarations.

### XPath expression result types

- Node set
- String
- Number
- Boolean

### Components of XPath expressions

- A series of location steps to specify a node or a set of nodes in a DOM tree.
- A collection of functions that manipulate node sets, strings, numbers, and boolean values.

## Location Steps

- Primary unit in an XPath.
- A series of location steps defines node or a set of nodes.
- Syntax:   axis :: nodeTest [predicate]

### axis

One of thirteen directions from the current node.

At this point we need only the *child* axis and the *attribute* axis.

### node test

A specification of a node of some sort using the name of the node, a function that specifies a kind of node, or an abbreviation.

### predicate

A filter that removes unwanted nodes by means of a boolean expression. Only those nodes for which the predicate is true are retained.

## Location Definitions

A node or a set of nodes can be specified in the DOM tree by a sequence of location steps separated by / symbols.

The path can be defined from the document root of the tree, denoted by starting with / (an absolute path) or can be specified relative to some node that we are currently visiting.

We consider only absolute paths starting at the document root specified by starting the path expression with a slash (/).

Location paths are read from left to right.

## Sample XPath Expressions

These expressions are to be evaluated against the XML document *phoneA.xml*.

Some of the expression will use abbreviations:

No axis means the child axis.

@ means the attribute axis.

A numeric predicate, such as [3] means [position()=3].

Positions start counting at 1.

| **Expression** | **Value** |
|---|---|
| / | Entire document tree |
| /child::phoneNumbers/child::entries/child::entry | Set of 4 entry nodes |
| /phoneNumbers/entries/entry[1] | First entry node |
| /phoneNumbers/entries/entry[3]/name/attribute::gender | Value of gender attribute for third entry node |
| /phoneNumbers/entries/entry[4]/name/@gender | Value of gender attribute for last entry node |
| count(/phoneNumbers/entries/entry) | Number of entry nodes |

## Displaying XPath Expressions

When an XPath expression is evaluated to be used in a context that expects a string, its value is converted into string data.

- Numbers become the corresponding strings of digits, signs, and special symbols.
- Boolean values become the strings "true" or "false".
- Node sets become the concatenation of the values of each of the nodes in the set.
- An element node becomes the concatenation of all of the text in the content of the node, including descendants.
- An attribute node becomes the value of the attribute.

For illustration here are the string results from some of the XPath expressions listed above. For the purpose of this example, all spaces have been replaced by "+" and all new line characters by "[nl]".

XPath = /phoneNumbers/entries/entry[1]/name

value = "[nl]+++++++++++Rusty[nl]+++++++++++Nail[nl]++++++++"

XPath = count(/phoneNumbers/entries/entry)

value = "4"

XPath = /phoneNumbers/entries/entry[2]/name/@gender

value = "male"

Xpath = /phoneNumbers/entries/entry[1]/name/@gender

value = ""

```
XPath = /phoneNumbers/entries/entry[5]/name
value = ""
```

```
XPath = /phoneNumbers/entries/entry[1]
value = "[nl]++++++++[nl]+++++++++++Rusty[nl]+++++++++++Nail[nl]
    ++++++++[nl]++++++++335-0055[nl]++++++++Iowa+City[nl]+++++"
```

```
XPath = count(/phoneNumbers/entries/entry) > 0
value = "true"
```

These examples were executed in Java using a DOM parser.

If we alter the DOM factory so that it creates a validating parser that removes ignorable white space and comments, some of the values are quite different.

```
XPath = /phoneNumbers/entries/entry[1]/name
value = "RustyNail"
```

```
XPath = /phoneNumbers/entries/entry[1]
value = "RustyNail335-0055Iowa+City"
```

```
XPath = /

value = "Phone+NumbersRustyNail335-0055Iowa+City
         JustinCase354-9876CoralvillePearlE.Gates
         335-4582North+LibertyHelenBack337-5967"
```

# XPath in Java

Java 1.5 has added a package *javax.xml.xpath* that provides classes and interfaces for creating XPath expressions and evaluating them.

First we use the class XPathFactory to create an instance of itself and with that instance create an XPath object.

Below we list the methods to be used.

## Methods in XPathFactory

XPathFactory newInstance()

XPath newPath()

## Methods in XPath

String evaluate(String exp, Object domNode)
**throws** XPathExpressionException

Object evaluate(String exp, Object domNode, QName rType)
**throws** XPathExpressionException

XPathExpression compile(String exp)
**throws** XPathExpressionException

## Method in XPathExpression

String evaluate(Object domNode)
**throws** XPathExpressionException

Object evaluate(Object domNode, QName rType)
**throws** XPathExpressionException

The QName parameter *rType* specifies the type of the Object returned by the *evaluate* method.

It must take one of the predefined constants

XPathConstants.NUMBER
XPathConstants.STRING
XPathConstants.BOOLEAN
XPathConstants.NODE
XPathConstants.NODESET

The first three constants correspond to the Java classes Double, String, and Boolean, respectively.

## Version 4 Continued: XPhoneParser.java

In this version we collect information from the DOM tree by addressing nodes directly using XPath expressions.

Since we do not have to navigate through the trees, the Java code is much simpler.

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.xml.sax.SAXException;

import javax.xml.xpath.XPathFactory;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpressionException;

import java.util.ArrayList;
import java.io.*;
```

```
public class XPhoneParser
{
    private DocumentBuilder builder;
    private XPath path;

    XPhoneParser() throws ParserConfigurationException
    {
        DocumentBuilderFactory factory
                 = DocumentBuilderFactory.newInstance();
        builder = factory.newDocumentBuilder();

        XPathFactory xpf = XPathFactory.newInstance();
        path = xpf.newXPath();
    }

   ArrayList<Entry> parse(String fileName)
              throws SAXException, IOException,
                             XPathExpressionException
   {
      File f = new File(fileName);
      Document doc = builder.parse(f);
      ArrayList<Entry> entries = new ArrayList<Entry>();

      Double d = (Double)path.evaluate(
                    "count(/phoneNumbers/entries/entry)",
                    doc, XPathConstants.NUMBER));

      int entryCount = (int)d.doubleValue();

      for (int k=1; k<=entryCount; k++)
      {
         String first = path.evaluate(
   "/phoneNumbers/entries/entry["+k+"]/name/first", doc);

         String middle = path.evaluate(
   "/phoneNumbers/entries/entry["+k+"]/name/middle", doc);
```

```
         String last = path.evaluate(
      "/phoneNumbers/entries/entry["+k+"]/name/last", doc);

            Name name = new Name(first, middle, last);

            String gender = path.evaluate(
   "/phoneNumbers/entries/entry["+k+"]/name/@gender", doc);

            String phone = path.evaluate(
      "/phoneNumbers/entries/entry["+k+"]/phone", doc);

            String city = path.evaluate(
      "/phoneNumbers/entries/entry["+k+"]/city", doc);

            Entry ent = new Entry(name, gender, phone, city);
            entries.add(ent);
         }
         return entries;
      }

      public static void main(String [] args) throws Exception
      {
         XPhoneParser parser = new XPhoneParser();
         ArrayList<Entry> entries = parser.parse("phoneA.xml");

         for (Entry anEntry : entries)
            System.out.println(anEntry);
      }
   }
```

## Creating XML Documents Dynamically

Since XML documents consist of text only, a program can create XML by simply printing strings.

In the next example, we generate Fibonacci numbers and write their values marked up with appropriate XML. The program even writes an internal DTD to output.

```
import java.math.BigInteger;

public class ValidFib
{
   public static void main(String [] args)
   {
      System.out.println("<?xml version=\"1.0\"?>");
      System.out.println("<!DOCTYPE fibonacciNumbers [");
      System.out.println(
         "  <!ELEMENT fibonacciNumbers (fibonacci)*>");
      System.out.println(
         "  <!ELEMENT fibonacci (#PCDATA)>");
      System.out.println(
         "  <!ATTLIST fibonacci index NMTOKEN #REQUIRED>");
      System.out.println("]>");

      System.out.println("<fibonacciNumbers>");
      BigInteger low  = BigInteger.ONE;
      BigInteger high = BigInteger.ONE;

      for (int k=0; k<10; k++)
      {
         System.out.print("  <fibonacci index=\"" + k + "\">");
         System.out.print(low);
         System.out.println("</fibonacci>");
         BigInteger temp = high;
         high = high.add(low);
         low = temp;
      }
      System.out.println("</fibonacciNumbers>");
   }
}
```

## Output

```
<?xml version="1.0"?>
<!DOCTYPE fibonacciNumbers [
 <!ELEMENT fibonacciNumbers (fibonacci)*>
 <!ELEMENT fibonacci (#PCDATA)>
 <!ATTLIST fibonacci index NMTOKEN #REQUIRED>
]>
<fibonacciNumbers>
 <fibonacci index="0">1</fibonacci>
 <fibonacci index="1">1</fibonacci>
 <fibonacci index="2">2</fibonacci>
 <fibonacci index="3">3</fibonacci>
 <fibonacci index="4">5</fibonacci>
 <fibonacci index="5">8</fibonacci>
 <fibonacci index="6">13</fibonacci>
 <fibonacci index="7">21</fibonacci>
 <fibonacci index="8">34</fibonacci>
 <fibonacci index="9">55</fibonacci>
</fibonacciNumbers>
```

## Problems with Using Text Output to Build XML

The responsibility for creating well-formed XML lies entirely on the programmer.

The only guarantee of matching tags is very careful programming.

This approach may be adequate for small and simple XML documents, but it will be very difficult to produce a complex document using hand-coded output.

Alternative: Use methods in the DOM classes and interfaces to build a DOM tree. Then we are guaranteed to get at least a well-formed XML document.

## Methods for Creating a DOM Tree

These methods are designed to work down from the root of a newly created empty DOM tree of type Document.

Observe that we have ways to create the nodes we need and to add them to the document tree as we build the XML document it represents.

### Method in DocumentBuilder

```
    Document newDocument()
```

### Methods in Document

```
    Element createElement(String tagName)
    Text createTextNode(String data)
    Attr createAttribute(String name)
    Comment createComment(String data)
    CDATASection createCDATASection(String data)
    ProcessingInstruction
        createProcessingInstruction(String target, String data)
```

## Methods in Node

Node appendChild(Node newChild)

Node insertBefore(Node newChild, Node refChild)

Node removeChild(Node oldChild)

Node replaceChild(Node newChild, Node oldChild)

## Methods in Element

**void** setAttribute(String name, String value)

Attr setAttributeNode(Attr newAttr)

**void** removeAttribute(String name)

Attr removeAttributeNode(Attr oldAttr)

## Method in Attr

**void** setValue(String value)

## Methods In CharacterData

Inherited by Text, Comment, and CDATASection

**void** appendData(String data)

**void** deleteData(**int** offset, **int** count)

**void** insertData(**int** offset, String data)

**void** replaceData(**int** offset, **int** count, String data)

## Method in ProcessingInstruction

**void** setData(String data)

# Example: XMLBuilder.java

In this example we create an array list containing four Entry objects built using hard-coded information (in the main method).

### Methods in the Program

Document build(ArrayList<Entry> entries)

Creates a Document object (the DOM tree) using the array list of Entry objects.

**private** Element createEntries(ArrayList<Entry> entries)

Create an element node representing the tree rooted at the element *entries*.

**private** Element createEntry(Entry anEntry)

Create an element node representing a tree rooted at the element *entry*.

**private** Element createName(Name n, String gender)

Create an element node representing a tree rooted at the element *name*.

**private** Element createTextElement(String nm, String txt)

This utility operation creates an element with a given name *nm* and specified text content *txt*.

Printing the tree created will require some new features of Java 1.5 that we will describe after the program listing.

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Text;
import java.util.ArrayList;
import java.util.Iterator;
import org.w3c.dom.DOMImplementation;
import org.w3c.dom.ls.DOMImplementationLS;
import org.w3c.dom.ls.LSSerializer;

public class XMLBuilder
{
   private DocumentBuilder builder;
   private Document doc;

   Builder() throws ParserConfigurationException
   {
      DocumentBuilderFactory factory =
                 DocumentBuilderFactory.newInstance();
      builder = factory.newDocumentBuilder();
   }

   Document build(ArrayList<Entry> entries)
   {
      doc = builder.newDocument();
      Element pn = doc.createElement("phoneNumbers");
      doc.appendChild(pn);
      pn.appendChild(
            createTextElement("title", "Phone Numbers"));
      pn.appendChild(createEntries(entries));
      return doc;
   }
```

```
private Element createEntries(ArrayList<Entry> entries)
{
   Element e = doc.createElement("entries");
   for (Entry anEntry : entries)
         e.appendChild(createEntry(anEntry));
   return e;
}

private Element createEntry(Entry ent)
{
   Element e = doc.createElement("entry");
   e.appendChild(createName(ent.getName(),
                            ent.getGender()));
   e.appendChild(
         createTextElement("phone", ent.getPhone()));
   String city = ent.getCity();
   if (city != null && !city.equals(""))
      e.appendChild(createTextElement("city", city));
   return e;
}

private Element createName(Name n, String gender)
{
   Element e = doc.createElement("name");
   if (gender != null && !gender.equals(""))
      e.setAttribute("gender", gender);
   e.appendChild(createTextElement("first", n.getFirst()));
   String middle = n.getMiddle();
   if (middle != null && !middle.equals(""))
      e.appendChild(createTextElement("middle", middle));
   e.appendChild(createTextElement("last", n.getLast()));
   return e;
}
```

```
   private Element createTextElement(String nm, String txt)
   {
      Element e = doc.createElement(nm);
      Text t = doc.createTextNode(txt);
      e.appendChild(t);
      return e;
   }

   public static void main(String [] args) throws Exception
   {
      ArrayList<Entry> entries = new ArrayList<Entry>();

      entries.add(new Entry(new Name("Robin", "Banks"),
                            "354-4455"));

      entries.add(new Entry(new Name("Forrest", "Murmers"),
                            "male", "341-6152", "Solon"));

      entries.add(new Entry(new Name("Barb", "A", "Wire"),
                            "337-8182", "Hills"));

      entries.add(new Entry(new Name("Isabel", "Ringing"),
                            "female", "335-5985", null));

      XMLBuilder builder = new XMLBuilder();
      Document doc = builder.build(entries);

      DOMImplementation imp = doc.getImplementation();
      DOMImplementationLS impLS =
         (DOMImplementationLS)imp.getFeature("LS", "3.0");
      LSSerializer ser = impLS.createLSSerializer();
      String out = ser.writeToString(doc);
      System.out.println(out);
   }
}
```

## Displaying the Tree

One way to get a string representation of the DOM tree (the XML document) uses an object of type LSSerializer (LS means Load and Save).

We obtain this LSSerializer using the following mysterious recipe.

1. Using the Document object, get the DOMImplementation object that handles this document.

   ```
   DOMImplementation imp = doc.getImplementation();
   ```

2. Get a DOMImplementationLS object using the *getFeature* method of the DOMImplementation with two parameters, feature = "LS" and version = "3.0".

   ```
   DOMImplementationLS implLS =
       (DOMImplementationLS)imp.getFeature("LS", "3.0");
   ```

3. Using the instance method *createLSSerializer* for the DOMImplementationLS object, we get an LSSerializer object.

   ```
   LSSerializer ser = impLS.createLSSerializer();
   ```

4. The LSSerializer object knows how to write the tree. We use the method that creates a string.

   ```
   String out = ser.writeToString(doc);
   ```

Note that the string produced by *ser.writeToString* has no ignorable white space, which mean there are no new lines and no indenting.

To see the XML document with new lines and indenting, we can run it through the linux utility *xmllint* with the *--format* option.

If we want to look at the file in a text editor, we must change the encoding to "UTF-8" before formatting with *xmllint*.

# Formatted Output

```
<?xml version="1.0" encoding="UTF-16"?>
<phoneNumbers>
  <title>Phone Numbers</title>
  <entries>
    <entry>
      <name>
        <first>Robin</first>
        <last>Banks</last>
      </name>
       <phone>354-4455</phone>
    </entry>
    <entry>
      <name gender="male">
        <first>Forrest</first>
        <last>Murmers</last>
      </name>
       <phone>341-6152</phone>
      <city>Solon</city>
    </entry>
    <entry>
      <name>
        <first>Barb</first>
        <middle>A</middle>
        <last>Wire</last>
      </name>
       <phone>337-8182</phone>
      <city>Hills</city>
    </entry>
    <entry>
      <name>
        <first>Isabel</first>
        <last>Ringing</last>
      </name>
       <phone>335-5985</phone>
    </entry>
  </entries>
</phoneNumbers>
```