The Joy of Coding

XML, the Extensible Markup Language, is a text-based language for representing arbitrary data. The fact that it is text-based, well-formed, and hierarchical makes it especially useful for data storage and exchange. We will discuss XML and learn about the JAXP API for manipulating XML data. Despite being old (and, at times, a little clunky), these APIs provide good examples of some important patterns and abstractions in object-oriented programming.

The Extensible Markup Language

- Introduction to XML
- The Document Type Definition
- The Document Object Model

The Extensible Markup Language

XML is a text-based language that, like HTML, uses *tags* to represent data:

```
<?xml version='1.0' encoding='us-ascii'?>
<person shoeSize="10.5">
  <!-- Every person has a name -->
  <name>Dave</name>
</person>
```

Each XML document begins a *prolog* that states the version of XML being used (`<?xml ... ?>`)

Tags are said to denote *elements* of an XML file

- Elements can be nested as shown above
- Elements can have no subelements (e.g. `<person/>`)

Unlike HTML, XML documents are well-formed

- Each opening tag must have a closing tag
- Tags must be properly nested

Elements may also have name/value pairs associated with them called *attributes* (e.g. `shoeSize`)

The Extensible Markup Language

What makes XML interesting?

- Plain text can be easily read by humans and a bevy of tools
- Tells you what kind of data you have, not just how to display it
  - Easier to extract the information you want
- Well-formatted means that it can be processed safer and easier
- Hierarchical in nature
  - Can be searched more efficiently
  - Natural fit with objects!

Working with XML

Industry-wide acceptance and standardization

- SAX API: Serial access to XML documents (fast)
- DOM API: Modeling XML documents as a tree of objects (convenient)
- Numerous XML-based standards for sharing data (multimedia, documents, converting XML to HTML, etc.)
- *Web Services* use XML and HTTP to communicate over the internet
  - Helps homogenize the internet
  - Windows, UNIX, cell phones, etc. can all speak web services
- Many organizations working to improve, extend, and use XML

Model data with an element or an attribute?

Use an element when:

- The data contains substructures
- The data contains multiple lines (e.g. long lines of text)
- The data changes frequently or varies greatly
- The data is meant to be displayed to the user
- The data is the “contents” of some “container”

Use an attribute when:

- The data is small and rarely changes
- There are only a small, fixed number of choices for the data’s value
- The data is used for processing the document (not seen by the user)
- The data is a “characteristic” of some “container”

Text Data in XML

Because XML data, itself, is represented as text, describing text in an XML document is little screwy:

Character Data (`CDATA`) is taken as a literal string

- Special characters like < will be ignored
- Attributes that have textual values are of type `CDATA`
- CDATA is quoted or double-quoted

Parsed Character Data (`PCDATA`) is parsed by the parser

- Special characters like < must be escaped with character sequences like `<`
- Elements that contain text are of type `#PCDATA`
- PCDATA is free standing in the XML document

Example:

```
<expression short="x>5">
  <display>x &gt; 5</display>
</expression>
```


The Document Type Definition

Many XML documents reference a Document Type Definition

- Specifies the kinds of elements (tags) that may appear in an XML document
- When the XML document is parsed, its DTD is consulted to ensure that it is structurally correct
- Not overly easy to use

A DTD for a person:

```
<?xml version='1.0' encoding='us-ascii'?>
<!ELEMENT person (name)>
<!ATTLIST person
          shoeSize CDATA #IMPLIED
>

<!ELEMENT name #PCDATA>
```


Parts of a DTD

A DTD looks a little like an XML document

- Begins with a declaration
- Denote XML elements with the `ELEMENT` tag that states the name of the element followed by its valid contents
- The following qualifiers can be used when describing elements

?	Optional (zero or one)
*	Zero or more
+	One or more
|	“or”
,	“followed by”

- Elements that contain text are denoted as `#PCDATA` for “parsed character data”
- Elements with empty bodies are denoted with `EMPTY`
- An element name can only be defined once

Parts of a DTD

The attributes of an element are specified by the `ATTLIST` tag

Each attribute is described by three values: name, type, and specification

- type may be a list of choices such as `(red | blue | green)` or un-parsed character data (`CDATA`)
- The specification states if the attribute is required

`#REQUIRED`	Attribute must be specified
`#IMPLIED`	Not required, application must have default value
value	Some default value for the attribute
`#FIXED` value	If value is present it must have this value

There are also DTD tags called “entities” that are essentially macros that are expanded in the XML document.

Designing a DTD

To motivate our discussion of XML and Document Type Definitions, we model a phone book containing entries for residents and businesses

Each resident has a first and last name, an optional middle initial, an address, and a telephone number. Residents may be marked as “unlisted”.

Each business has a name, an address, and a telephone number.

Each address consists of multiple lines of text giving the street address, an optional apartment number, and a city, state, and zip code

Each telephone number has a three-digit area code and a seven digit number

A DTD for our phonebook

```
<?xml version='1.0' encoding='us-ascii'?>
<!ELEMENT phonebook (resident | business)*>

<!ELEMENT resident (first-name, initial?, last-name,
                    address, phone)>
<!ATTLIST resident
            unlisted (true | false) #IMPLIED
>

<!ELEMENT business (name, address, phone)>

<!ELEMENT name (#PCDATA)>
<!ELEMENT first-name (name)>
<!ELEMENT last-name (name)>
<!ELEMENT initial (#PCDATA)>

<!ELEMENT address (street+, apt?, city, state, zip)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT apt (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT zip (#PCDATA)>

<!ELEMENT phone EMPTY>
<!ATTLIST phone
           areacode   CDATA  #REQUIRED
           number     CDATA  #REQUIRED
>
```


Specifying the DTD for an XML document

The `<!DOCTYPE>` tag is used to specify the root element and the DTD that describes the format of an XML document.

The DTD is an *external entity* from the XML document and can be specified by a “system id” or a “public id”

A system id specifies a URL (or relative file name) for the DTD

```
<!DOCTYPE phonebook SYSTEM "phonebook.dtd">

<!DOCTYPE family-tree SYSTEM
   "http://www.--.edu/~whitlock/dtds/familytree.dtd">
```

A public id specifies a URN (Universal Resource Name) for the DTD

```
<!DOCTYPE family-tree PUBLIC
   "-//Portland St Univ//DTD CS399J Family Tree//EN"
   "http://www.--.edu/~whitlock/dtds/familytree.dtd">
```

When an XML processor sees a public id, it can use a local copy of the DTD (e.g. in a jar file) instead of visiting the URL.

An XML document for our phonebook

```
<?xml version='1.0' encoding='us-ascii'?>
<!DOCTYPE phonebook SYSTEM 'phonebook.dtd'>

<phonebook>
  <resident>
    <first-name><name>David</name></first-name>
    <initial>M</initial>
    <last-name><name>Whitlock</name></last-name>
    <address>
      <street>PSU CS Department</street>
      <street>P.O. Box 751</street>
      <city>Portland</city>
      <state>OR</state>
      <zip>97201</zip>
    </address>
    <phone areacode="503" number="725-4039"/>
  </resident>
  <business>
    <name>Powell's Technical Bookstore</name>
    <address>
      <street>33 NW Park</street>
      <city>Portland</city>
      <state>OR</state>
      <zip>97209</zip>
    </address>
    <phone areacode="503" number="228-3906"/>
  </business>
</phonebook>
```


Java APIs for Working with XML

The World Wide Web Consortium (W3C) is the international standards body responsible for XML

- The W3C has published several APIs consiting of interfaces and abstract classes that model and manipulate XML
  - `org.xml.sax`: The Simple API for XML Provides event-driven, serial access to XML data
  - `org.w3c.dom`: The Document Object Model models XML data as an object graph
- A vendor (Sun, IBM, Apache) implements these interfaces

People from across the Java ecosystem came together to develop a standard set of APIs for parsing XML (JAXP)

- `javax.xml.parsers`: APIs for parsing XML Provides an interface to vendors' XML parsers
- `javax.xml.transform`: APIs for transforming XML into text, HTML, etc. using the Extensible Stylesheet Language for Transformations (XSLT)

The Simple API for XML (SAX)

SAX provides serial access to XML data

- The programmer provides an implementation of callback* methods that are invoked as the XML data is parsed
- Fast and memory-efficient, but not very natural
- Cannot look backwards in the document

`javax.xml.parsers.SAXParserFactory` and `javax.xml.parsers.SAXParser` are used to parse XML data using SAX

Interfaces in the `org.xml.sax` package such as `ContentHandler`, `ErrorHandler`, and `InputSource` are inputs to SAX parsing

`org.xml.sax` also contains exceptions throwing during parsing

- `SAXException`, `SAXParseException`, etc.

*A callback is a piece of functionality that a user provides to a third party piece of software such as a parser or a GUI event manager.

XML Parsers and the Factory Pattern

The W3C and standard Java specify the interfaces for parsing XML, but somebody else actually implements it

- The consumer of these APIs shouldn't know about the implementation classes
- "Program to the Interface"

XML Parsers use the "Factory" design pattern to encapsulate their configuration and creation

- A "factory" object knows how to create a parser
- Hides details of creation, constructors are not invoked directly
- Implementation-specific features can be set on the factory as key/value `String`s

The choice of parser factory implementation is specified by a system property or in a JRE configuration file

Obtaining a SAX parser

`javax.xml.parsers` contains classes for parsing XML

A `SAXParserFactory` is an object that creates a `SAXParser`

- The factory's configuration determines what kind of parser is created
- Features may be set by name (a `String`)
- The `setValidating` method is used to enable validation of XML againsts its DTD
- Once the factory has been configured, its `newSAXParser` method returns the desired `SAXParser`
- If the configuration isn't correct a `ParserConfigurationException` is thrown

Note that `SAXParser`'s constructor is protected!

Parsing XML with SAX

`SAXParser`'s `parse` methods parse XML data using SAX

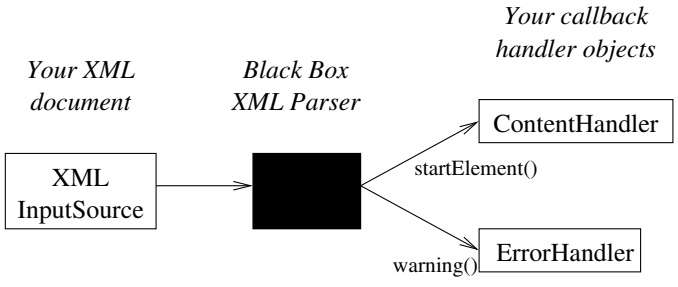
The XML data can come from several sources

- A `java.io.File`
- A `java.io.InputStream`
- A `String` Uniform Resource Identifier (URI)
- An `org.xml.sax.InputSource` (can be wrapped around an `InputStream`, `Reader`, etc.)
  - Should call `setSystemId` with URI of source, so it can find relative entities (like DTDs)

Parsing XML with SAX

During parsing, the methods of an `org.xml.sax.helpers.DefaultHandler` are invoked

- A `DefaultHandler` is a convience class that provides no-op implementations of: `EntityResolver`, `DTDHandler`, `ContentHandler`, `ErrorHandler`



While parsing, an `IOException` or `SAXException` may be thrown

`org.xml.sax.ContentHandler`

`ContentHandler` contains callback methods that are invoked as the XML document is being parsed

- `startDocument/endDocument`
- `startElement/endElement`
  - Has an `Attributes` with a `type` and `value`

`org.xml.sax.EntityResolver`

An `EntityResolver`'s `resolveEntity` method is invoked when the SAX parser is looking for an external identity with a given public ID and system ID

- Good for intercepting requests for external documents
  - For example, if the dtd is located in a jar file or database

`org.xml.sax.DTDHandler`

A `DTDHandler` provides callback methods invoked while parsing a DTD ("events") – Not widely used

org.xml.sax.ErrorHandler

An `ErrorHandler` contains methods that are invoked when problems are encountered during parsing

- Allows applications to handle errors in different ways
- For example, log error message to a file
- The default behavior is to not report errors, so you really need an `ErrorHandler`!

Each of the callback methods is invoked with a `SAXParserException` containing the line and column number being parsed when the problem was encountered

- `warning`: An element is not declared in the DTD, a DTD contains multiple declarations of the element, etc.
- `error` (recoverable): Portion of data does not match encoding, etc.
- `fatalError` (not recoverable): The XML is not well-formed, cannot process the encoding, etc.

Most of the time, you'll want to just rethrow the `SAXParserException` or wrap it in another exception

Example SAX Parser

```
package edu.pdx.cs399J.xml;

import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;

public class PrintPhoneNumbers
  extends DefaultHandler {

  private static PrintStream out = System.out;
  private static PrintStream err = System.err;

  public void startElement(String namespaceURI,
                           String localName,
                           String qName,
                           Attributes attrs)
    throws SAXException {

    if (qName.equals("phone")) {
      String area = attrs.getValue("areacode");
      String number = attrs.getValue("number");
      out.println("(" + area + ") " + number);
    }
  }
```


Example SAX Parser

```
  public void warning(SAXParseException ex) {
    err.println("WARNING: " + ex);
  }

  public void error(SAXParseException ex) {
    err.println("ERROR: " + ex);
  }

  public void fatalError(SAXParseException ex) {
    err.println("FATAL: " + ex);
  }
```


Example SAX Parser

```
  public static void main(String[] args) {
    SAXParserFactory factory =
      SAXParserFactory.newInstance();
    factory.setValidating(true);

    SAXParser parser = null;
    try {
      parser = factory.newSAXParser();

    } catch (ParserConfigurationException ex) {
      // ...
    } catch (SAXException ex) {
      // ...
    }

    DefaultHandler handler = new PrintPhoneNumbers();
    try {
      File file = new File(args[0]);
      InputSource source =
        new InputSource(new FileReader(file));
      source.setSystemId(file.toURL().toString());
      parser.parse(source, handler);

    } catch (SAXException ex) {
      // ...
    } catch (IOException ex) {
      // ...
    }
```

Example SAX Parser

```
$ java edu.---.PrintPhoneNumbers phonebook.xml
(503) 725-4039
(503) 228-3906
```

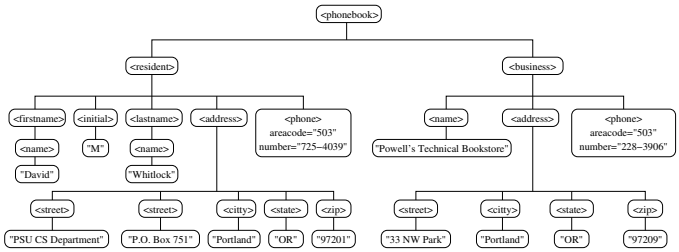
When the parser parsed an element, it called `startElement`

SAX parsing is good when you only want a little information from XML data or the data is fairly simple

The Document Object Model

The Document Object Model is a programming API for documents put forward by the World Wide Web Consortium (W3C)

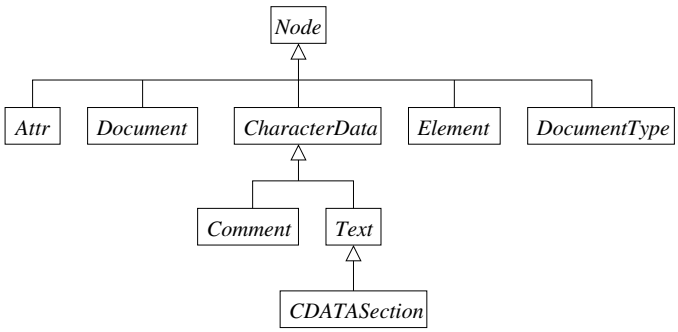
DOM views structured hierarchical documents as trees of objects



DOM gives you an object view of an XML file and provides an API by which the objects can be examined and modified

The DOM API: The `org.w3c.dom` package

`org.w3c.dom` contains interfaces that specify which objects a DOM tree may contain



`org.w3c.dom.Node`

`Node` represents a node in a DOM tree

- `appendChild`: Adds a child `Node`
- `getNodeName`: Returns the name of the node (e.g. element name)
- `getAttributes`: Returns a `Node`'s attributes as a `NamedNodeMap` (only `Element` nodes have attributes)
- `getNodeValue`: Returns the "value" of a `Node` (e.g. the text of a `Text` node)
- `getOwnerDocument`: Returns the `Document` in which a `Node` resides
- `getParentNode`: Returns a `Node`'s parent in the DOM tree

org.w3c.dom.Document

A `Document` represents an XML document and contains factory methods for creating `Node`s:

- `createElement`
- `createTextNode`
- `createAttribute`
- `createComment`

Also contains methods to examine the XML document

- `getDoctype`: Returns a `DocumentType` object that models the DTD for an XML document
- `getDocumentElement`: Returns the root `Element` of the document
- `getElementsByTagName`: Returns all of the `Element`s in a document with the given name

org.w3c.dom.Element

An `Element` represents an element in an XML document.

- `getAttribute`/`getAttributeNode`: Returns the value (`String`/`Attr`) of a given named attribute
- `getTagName`: Returns the name of an `Element` (you can also use `getNodeName`)
- `removeAttribute`/`removeAttributeNode`
- `setAttribute`/`setAttributeNode`

org.w3c.dom.Attr

`Attr` represents an attribute of an `Element`

- `getName`: Returns the name of an `Attr`
- `getValue` / `setValue`

org.w3c.dom.NamedNodeMap

The attributes of an `Element` are described by a `NamedNodeMap` (a "collection" that maps `String`s to `Node`s)

- `getLength`: Returns the number of items in a `NamedNodeMap`
- `getNamedItem`: Returns the `Node` (e.g. `Attr`) with a given name
- `item`: Returns the `Node` at a given index in the map
- `setNamedItem`: Adds a `Node` to the mapping
- `removeNamedItem`

org.w3c.dom.CharacterData

`CharacterData` represents (duh) character data in a document

- `getData`: Returns a `String` representing the character data
- `setData` / `appendData` / `insertData`

org.w3c.dom.Text

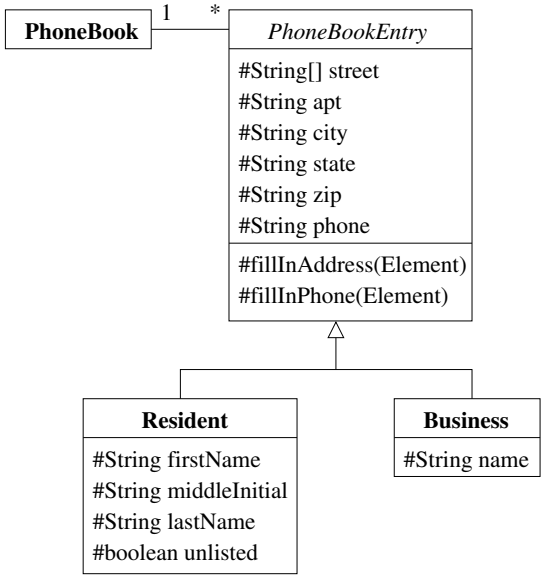
`Text` represents PCDATA

org.w3c.dom.Comment

`Comment` represent a comment in a document

Turning DOM trees into Objects

Now, we're going to use the data stored in a DOM tree to construct Java objects



The Phone Book Classes

`PhoneBook`s, `Resident`s and `Business`es are constructed from pieces (`Element`s) of a DOM tree*

- Examine each of the `Element`'s children in the DOM tree and extracting information from them
- Common code for extracting address and phone information was *refactored* into the `PhoneBookEntry` class
- The `toString` method of `Resident` creates a `String` for the resident's name and then invokes `super.toString()` to handle the address and phone portion
- Code reuse!
- Note the use of `protected` fields and methods

*In this example code, the domain classes are highly dependent on the XML APIs. In general, it's better to separate domain objects from code that converts them to other data formats.

Constructing a `PhoneBook`

```
package edu.pdx.cs399J.xml;
import java.io.*;
import java.util.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;

public class PhoneBook {
  private Collection entries = new ArrayList<>();

  public PhoneBook(Element root) {
    NodeList entries = root.getChildNodes();
    for (int i = 0; i < entries.getLength(); i++) {
      Node node = entries.item(i);
      if (!(node instanceof Element)) {
        // Ignore other stuff
        continue;
      }
      Element entry = (Element) node;
      switch (entry.getNodeName()) {
        case "resident":
          this.entries.add(new Resident(entry));
          break;
        case "business":
          this.entries.add(new Business(entry));
          break;
      }
    }
```


Highlights from `PhoneBookEntry`

```
package edu.pdx.cs399J.xml;
import java.util.*;
import org.w3c.dom.*;

public abstract class PhoneBookEntry {
  protected List streetLines = new ArrayList();
  protected String apt;     // snip...

  protected void fillInAddress(Element root) {
    NodeList elements = root.getChildNodes();
    for (int i = 0; i < elements.getLength(); i++) {
      Node node = elements.item(i);
      if (!(node instanceof Element)) {
        continue;
      }
      Element element = (Element) node;
      switch (element.getNodeName()) {
        case "street": {
          Node text = element.getFirstChild();
          this.streetLines.add(text.getNodeValue());
          break;
        }
        case "apt": {
          Node text = element.getFirstChild();
          this.apt = text.getNodeValue();
          break;
        }
        // Snip...
```

Extracting attributes from elements

From `PhoneBookEntry`:

```
protected void fillInPhone(Element phone) {
  String areacode = null;
  String number = null;

  // Examine the phone's attributes
  NamedNodeMap attrs = phone.getAttributes();
  for (int i = 0; i < attrs.getLength(); i++) {
    Node attr = attrs.item(i);
    switch (attr.getNodeName()) {
      case "areacode":
        areacode = attr.getNodeValue();
        continue;
      case "number":
        number = attr.getNodeValue();
        continue;
    }
  }

  this.phone = areacode + "-" + number;
}
```

We could have also used:

```
  areacode = phone.getAttribute("areacode");
  number = phone.getAttribute("number");
```


Parsing XML data as DOM

`javax.xml.parsers` contains classes for parsing XML data as DOM

- Follows the "factory" pattern
- A `DocumentBuilderFactory` creates a `DocumentBuilder`
- A `DocumentBuilderFactory` may be configured to create validating parsers
- `DocumentBuilder`'s `parse` methods read XML data from a source (`File`, `InputSource`, etc.) and from it create a `Document`
- DOM parsers use SAX parsers to do the heavy lifting
- `DocumentBuilder` has `setEntityResolver` and `setErrorHandler` methods

Parsing XML data as DOM

From `PhoneBook.java`

```
public static void main(String[] args) {
  // Parse the XML file to create a DOM tree
  Document doc = null;
  try {
    DocumentBuilderFactory factory =
      DocumentBuilderFactory.newInstance();
    factory.setValidating(true);

    DocumentBuilder builder =
      factory.newDocumentBuilder();
    doc = builder.parse(new File(args[0]));

  } catch (ParserConfigurationException ex) {
    // ...
  } catch (SAXException ex) {
    // ...
  } catch (IOException ex) {
    // ...
  }

  Element root =
    (Element) doc.getChildNodes().item(1);
  PhoneBook phonebook = new PhoneBook(root);
  System.out.println(phonebook);
}
```


Creating a `PhoneBook`

```
$ java edu.---.PhoneBook phonebook.xml
Phone Book

David M Whitlock
PSU CS Department
P.O. Box 751
Portland, OR 97201
503-725-4039

Powell's Technical Bookstore
33 NW Park
Portland, OR 97209
503-228-3906
```

Creating an empty DOM tree

The `org.w3c.dom.DOMImplementation` interface provides methods for creating DOM trees

- `createDocumentType` creates a `org.w3c.dom.DocumentType` that represents a DTD
  - Name of root element (e.g. `phonebook`)
  - DTD's public identifier (text description of DTD)
  - DTD's system identifier (e.g. DTD's URL)
- `createDocument` is used to create an empty XML `Document`
  - "Namespace URI" of the document (`null` for our purposes)
  - Name of the root element of the document
  - The `DocumentType` for the document

`DocumentBuilder`'s `getDOMImplementation` method returns a `DOMImplementation`

If an error occurs while creating a DOM tree, a `org.w3c.dom.DOMException` will be thrown

Converting XML

XML by itself is nice, but often we want to convert it into something else

- Text for viewing by humans
- HTML for display in a web browser
- A different XML format

The Extensible Stylesheet Language (XSL) is used for transforming XML

- XSLT: A language for specifying XML transformations
- XPath: A language for "addressing" portions of XML data
  - Lets XSLT distinguish between `<person><address>` and `<order><address>`
- XSL-FO: "Flow objects" that desribe font sizes, layouts and how information flows from one page to another

XSLT in Java

`javax.xml.transform` contains classes and interfaces for transforming XML

- A `TransformerFactory` creates a `Transformer`
- A `Transformer` transforms a `Source` to a `Result`
  - Accomplished with the `transform` method
- An `ErrorListener` has callback methods invoked when a `TransformerException` is thrown

`javax.xml.transform.dom` contains classes for using DOM as an input or output to/from an XML transformation

- `DOMSource`s and `DOMResult`s are created from an `org.w3c.dom.Node`

Similarly for `javax.xml.transform.sax`

`javax.xml.transform.stream` contains classes for using I/O streams for input/output to/from a transformation

- `StreamSource`: Reads from an `InputStream`, `Reader`, etc.
- `StreamResult`: Writes to an `OutputStream`, `Writer`

Formatting the Result of a Transformation

By default, a `Transformer` creates raw, unformatted XML

`Transformer`'s `setOutputProperty` method is used to set formatting properties

The constants defined in the `OutputKeys` class are used to specify a property

- `DOCTYPE_SYSTEM`: The system id of the DTD
- `DOCTYPE_PUBLIC`: The public id of the DTD
- `OMIT_XML_DECLARATION`: Should there be a `<?xml` declaration? (`yes` or `no`)
- `INDENT`: Should the output be indented? (`yes` or `no`)
- `METHOD`: How should the result outputted? (`text`, `xml`, `html`, or other)

Building a DOM tree from scratch

```
package edu.pdx.cs399J.xml;
import java.io.*;
import java.net.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*; // DOMSource
import javax.xml.transform.stream.*;  // StreamResult
import org.w3c.dom.*;

public class BuildPhonebook {
  private static PrintStream err = System.err;

  public static void main(String[] args) {
    String publicID = null;  // Who cares?
    String systemID = null;

    try {
      File dtd = new File("phonebook.dtd");
      systemID = dtd.toURL().toString();
    } catch (MalformedURLException ex) {
      err.println("** Bad URL: " + ex);
      System.exit(1);
    }

    // continued...
```


Building a DOM tree from scratch

```
    // Create an empty Document
    Document doc = null;
    try {
      DocumentBuilderFactory factory =
        DocumentBuilderFactory.newInstance();
      factory.setValidating(true);
      DocumentBuilder builder =
        factory.newDocumentBuilder();
      DOMImplementation dom =
        builder.getDOMImplementation();
      DocumentType docType =
        dom.createDocumentType("phonebook", publicID,
                               systemID);
      doc = dom.createDocument(null, "phonebook",
                               docType);

    } catch (ParserConfigurationException ex) {
      // ...
    } catch (DOMException ex) {
      // ...
    }
```


Building a DOM tree from scratch

```
    // Construct the DOM tree
    try {
      Element root = doc.getDocumentElement();

      Element biz = doc.createElement("business");
      root.appendChild(biz);

      Element name = doc.createElement("name");
      biz.appendChild(name);
      String br = "Tripwire, Inc.";
      name.appendChild(doc.createTextNode(br));

      Element address = doc.createElement("address");
      biz.appendChild(address);
      // continued...
```


Building a DOM tree from scratch

```
      Element street1 = doc.createElement("street");
      address.appendChild(street1);
      String st1 = "805 SW Broadway";
      street1.appendChild(doc.createTextNode(st1));

      Element city = doc.createElement("city");
      address.appendChild(city);
      city.appendChild(doc.createTextNode("Portland"));

      Element state = doc.createElement("state");
      address.appendChild(state);
      state.appendChild(doc.createTextNode("OR"));

      Element zip = doc.createElement("zip");
      address.appendChild(zip);
      zip.appendChild(doc.createTextNode("97205"));

      Element phone = doc.createElement("phone");
      biz.appendChild(phone);
      phone.setAttribute("areacode", "503");
      phone.setAttribute("number", "973-5200");

    } catch (DOMException ex) {
      err.println("** DOMException: " + ex);
      System.exit(1);
    }

    // continued...
```

Building a DOM tree from scratch

```
    // Write the XML document to the console
    try {
      Source src = new DOMSource(doc);
      Result res = new StreamResult(System.out);

      TransformerFactory xFactory =
        TransformerFactory.newInstance();
      Transformer xform = xFactory.newTransformer();
      xform.setOutputProperty(OutputKeys.INDENT,
                              "yes");
      xform.setOutputProperty(
             OutputKeys.DOCTYPE_SYSTEM, systemID);
      xform.transform(src, res);

    } catch (TransformerException ex) {
      ex.printStackTrace(System.err);
      System.exit(1);
    }
  }
}
```

Note that when we built the DOM tree we used `Elements`, `Documents`, and `DocumentTypes`

JAXP has its own classes that implement the W3C interfaces, but we don't care: Program to the interface!

Building a DOM tree from scratch

Running our program...

```
$ java edu.---.BuildPhonebook
<?xml version=''1.0'' encoding=''UTF-8''?>
<!DOCTYPE phonebook SYSTEM
   "file:/u/whitlock/public_html/src/edu/
    pdx/cs399J/xml/phonebook.dtd">
<phonebook>
<business>
<name>Tripwire, Inc.</name>
<address>
<street>308 SW 2nd Ave</street>
<street>Suite 400</street>
<city>Portland</city>
<state>OR</state>
<zip>97205</zip>
</address>
<phone areacode="503" number="276-7500"/>
</business>
</phonebook>
```


So What?

We've just seen how we can construct an XML document, model an XML document using DOM, and construct Java object from a DOM tree.

We could also generate a DOM tree, and thus an XML file, from our Java objects.

By using XML with a DTD we have agreed upon a standard representation of our data.

If we follow all use the same DTD, we can share our data without worries.

This is the promise of XML!

But, wait. There's more!

The XML Schema Definition (XSD) improves on DTD for defining a structure ("schema") for XML content

- XSD is expressed as XML
- Support for data types like numbers and dates, restrictions on data values, validation with regular expressions, etc.

The Java API for XML Binding (JAXB) streamlines the amount of Java code necessary to work with XML

- Generate Java classes directly from an XSD
  - Command line tools, Maven plugin, etc.
- Java annotations that guide how objects are mapped to XML
- Custom transformations for complex objects and JDK objects/enums

Summary

XML is a text-based markup language used for representing data

The format of an XML document is specified by a DTD

The Document Object Model is used to view hierarchical documents as a tree of objects

The JAXP API provides a standard interface for parsing XML data and working with DOM trees that represent them.

XML allows people to share data regardless of how their programs represent or manipulate it internally.