XML and JSON

The ***eXtensible Markup Language*** (**XML**) is a powerful way to describe hierarchically structured text information. Using XML means you can have your file formats well defined in a standard way, using ***DTD***s or ***Schema***s, which allows them to be automatically checked for validity. In addition, XML data can be easily transformed into other formats using ***XSLT***. You can create style sheets for your XML file formats, that allow them to be nicely displayed by any web browser. Finally there are standard APIs (libraries) that implement a lot of the low-level details of reading, modifying, searching, and writing XML files (***SAX*** and ***DOM***).

> Java comes with a number of ways to work with XML. The Java API for XML Processing (JAXP) enables applications to parse, transform, validate and query XML documents using an API that is independent of a particular XML processor implementation. It includes a number of APIs including DOM, SAX, StAX, XPath, and XSLT (although the implementation must support these).
>
> To use JAXP you must install some implementation of it (a jar file put in the "`ext`" directory). The reference implementation only comes with Java EE (not SE). But it is just the Apache XALAN-J XML implementation, and you can download that jar file yourself to use with Java SE.

XML technology replaces custom file format parsers and is today used for SOAP, RSS, and many other standard formats. What's so great about this is that to use (say) RSS, you only need an RSS document, which includes a link to its definition and so can automatically (up to a point) be parsed. (We used to call such documents *self-describing*.) Using standardized document definitions makes it much easier to share documents!

An XML document should start with an XML declaration:

```
<?xml version ="1.0"?>
```

You can (and should) specify the document's encoding too:

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

And include an optional (from XML's point of view) DOCTYPE header:

```
<!DOCTYPE person ...>
```

This tells a read the type of the document (and optionally, where the definition for the document's structure can be found).

You can include comments anywhere:

```
<!-- anything not containing a double dash -->
```

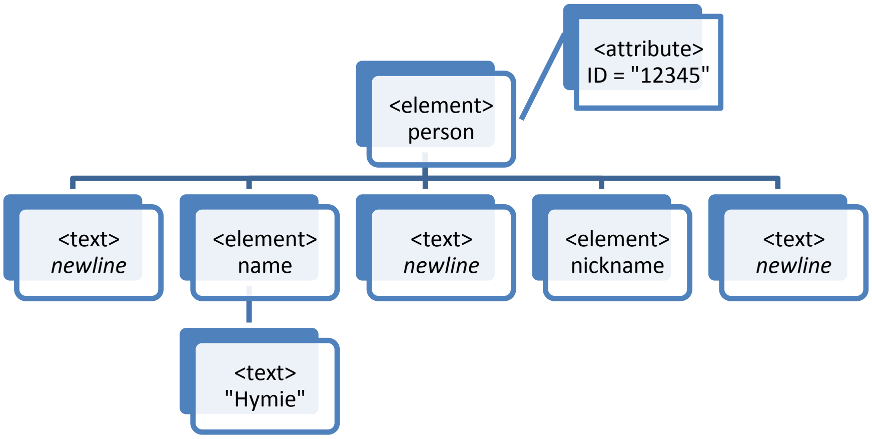
After these headers comes the document itself. An XML document forms a tree, similar to the structure of a filesystem on a disk. The topmost tag is the *root* node of the tree, and it contains other nodes.

An XML document uses *tags* that **always have an open and a close pair**. Tags can have required or optional *attributes* of the form *name="value"* (Note in XML the *value* must always be quoted.) The first tag defines the type of the document (and should match the DOCTYPE declaration if you provided one). In-between an open and close tag are other tag pairs and/or text.

If there is no enclosed data, then you can combine the start and end tags into a single *self-closing* tag. Here's an example:

```
<person id="12345">
  <name>Hymie</name>       tag data is: Hymie
  <nickname></nickname>   OR: <nickname/>
</person>
```

This XML document forms a graph that looks like this:



In this example, the *root* element (or *node*) is "person", which has 5 *children* nodes and one attribute node. The five children of person are *siblings*. One of these ("name") also has a child. Note siblings have an order. So you can refer to the first-child of a node, the next-sibling of a node, or a parent of a node. The tree can also be searched but functions that return a node or a list of nodes (e.g. all "name" nodes, or the "person" node with an "ID" attribute of "12345"). You can also add new nodes, modify nodes, or remove nodes from the tree.

A (starting) tag is generally referred to as an *element*. Both elements and text are *nodes*. The document above has seven nodes but only three of them are elements. The distinction may be important; the second element is "name" but the second node is a text node.

Depending on the document definition, white space between tags may be permitted or not. If permitted they from text nodes; if optional they are called *ignorable* and may not be shown in the tree.

Some characters such as the left or right bracket (or double-quote) would be difficult to include as data. Instead you can use *Unicode character references* (in decimal or hex): `&#x2122;` (hex) or `&#8482;` (decimal) represent the TM mark. A few characters have names (the ***character entity references***): `&lt;`, `&gt;`, `&amp;`, and a few others.

> HTML documents are processed in exactly the same way, forming a tree of nodes known as the HTML DOM. JavaScript can be used to search this tree and modify it.

XML Namespaces

A single XML document may have some parts defined by one schema, and other parts by another. What if each part had a tag that used the same name as another? This would be a tag name conflict.

To avoid this, tag names can be given a prefix. You pick the prefix and associated it with some schema (DTD). Often a single letter prefix is used. For example:

```
<p:person id="12345">
  <p:name>Hymie</p:name>
  <p:nickname></p:nickname>
</p:person>
```

The namespace declaration has the following syntax: `xmlns:prefix="URI"`. It can be used with most tags, but is often used in the *root* (outermost) tag. For example:

```
<people xmlns:h="http://www.w3.org/TR/html4/"
        xmlns:p="http://wpollock.com/people.dtd">
  <h:h1>List of People</h:h1>
  <p:person id="12345">
    <p:name>Hymie</p:name>
    <p:nickname></p:nickname>
  </p:person>
</people>
```

The URI isn’t actually used for anything except to provide a unique name. It is confusing since you expect to be able to type that URI in a web browser and see something. Sometimes organizations will put something there, a DTD or some web page, but this isn’t required.

If one namespace is used more often than the rest, you can make it the *default namespace*. This is done by adding this attribute (again, often to the root tag): `xmlns="namespaceURI"`. (Note this is the same as before, except no prefix.)

XML namespaces are commonly used for JSP (and JSF) documents, XSLT stylesheets, and xHTML.

XML Validation and Processing

XML documents that obey the above rules are said to be ***well-formed***. This is a simple check for a corrupted document. On the other hand you can supply a definition for some type of document, with a URL in the DOCTYPE header, or by including the definition within special tags. Either way a *parser* can read the document and its definition, and verify the document. Such a document is called ***valid***. The definition can be either a ***DTD*** (*document type definition*) or a ***Schema***. Neither is suitable for all documents, so some will have a DTD, some a Schema, and some will have both. This is why the language is *extensible*.

The definition can include new character entity definitions, legal tags and attributes, the type of data that can be used for an attribute or tag data, required and optional tags and attributes, the order of tags, etc.

To process an XML you need to *parse* it. You can use any of several methods, including:

- **SAX** (simple API for XML), which reads the document once from the beginning to the end, invoking *call-backs* to your code when tags are noted. Use for very large documents that won't fit into RAM, or if you are only interested in a few bits of the document (imagine a bank's daily transaction log, and you are only looking for a particular customer's records). No tree is built; each bit of the document is read, call-back methods are called, then the next bit is read.
- **DOM** (document object model), which reads the whole document into RAM as a tree, that can be walked, searched, or modified and then saved back as a file.

There are other APIs as well, notably StAX. Read the JAXP tutorials (for SAX, DOM, StAX, etc.) from Oracle.com for details.

There are a number of packages used for XML processing. See **`javax.xml`**, **`javax.xml.parsers`**, and other `javax.xml.*` packages. In addition the SAX processing classes you need are in **`org.xml.*`** packages while the DOM processing classes are in **`org.w3c.dom`** package (and a few others).

To use SAX, an instance of the `SAXParserFactory` class is used to generate an instance of an XML SAX parser. The parser wraps a `SAXReader` object. When the parser's `parse()` method is invoked, the reader invokes one of several callback methods implemented in the application. Those methods are defined by interfaces such as `ContentHandler`. `ContentHandler` has methods such as `startDocument`, `endDocument`, `startElement`, and `endElement`, which are invoked when an XML tag is recognized. This interface also defines methods such as `characters()`, which is invoked when the parser encounters the text in an XML element.

To build a DOM document in RAM (`org.w3c.dom.`**`Document`**) you use a **`DocumentBuilder`**. Class `Document` inherits from class `Node` which provides most of the interesting methods you want to use (show).

To allow for the future use of different types of DOM documents, Sun makes you get a `DocumentBuilder` object from a **`DocumentBuilderFactory`** class:

```
DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse(file);
                 // or inputStream or URL
Element root = doc.getDocumentElement();
String name = root.getTagName();
```

You use **`NodeList elements = root.getChildNodes`** to get the remaining elements (tags and data). (This was invented before Java Collections.) See on-line Demos for examples.

[*Adapted from a JavaWorld article, Pitfalls #4, by Michael Daconta*] A common problem with DOM is when a search for an element fails. All well-formed XML files have a tree structure. For example `myaddresses.xml` can be represented by a tree with two ADDRESS nodes:

```
<?xml version="1.0"?>
<!DOCTYPE ADDRESS_BOOK SYSTEM "abml.dtd">
<ADDRESS_BOOK>
  <ADDRESS>
    <NAME>Joe Jones </NAME>
    <STREET>4332 Sunny Hill Road</STREET>
    <CITY>Fairfax</CITY>
    <STATE>VA</STATE>
    <ZIP>21220</ZIP>
  </ADDRESS>
  <ADDRESS>
    <NAME>Sterling Software</NAME>
    <STREET> 7900 Sudley Road</STREET>
    <STREET> Suite 500</STREET>
    <CITY>Manassas</CITY>
    <STATE>VA </STATE>
    <ZIP>20109</ZIP>
  </ADDRESS>
</ADDRESS_BOOK>
```

Often, XML beginners wrongly assume that a DOM tree will look exactly like their mental image of the corresponding XML document. Let’s say you must find the first

NAME element of the first ADDRESS. By looking at the XML, you might think that the DOM's third node is the one you want. Unfortunately, the DOM tree includes text nodes for ignorable white space. Ignorable white space is white space that falls between tags, such as a newline or leading spaces or tabs.

One way to deal with this problem is to walk the DOM tree by fetching elements (the opening tags) rather than nodes (which are either elements or data):

```
Element root = doc.getDocumentElement();
NodeList = root.getElementsByTagName( "*" );
```

Another way is first to *normalize* your DOM by removing the ignorable while space nodes from the DOM tree. (Note there is another meaning to *normalize*; see Document.normalizeDocument().) You must be careful to not remove all white space nodes, just the ignorable ones! Then you can process as normal. (Show sample code on-line that shows both ways.)

In Java 5, you can validate an XML document from a schema (or from a DTD, the only method supported in older Java versions). Here's how:

```
final String s = XMLConstants.W3C_XML_SCHEMA_NS_URI;
SchemaFactory sf = SchemaFactory.newInstance(s);
StreamSource ss = new StreamSource("SomeSchema.xsd");
Schema schema = sf.newSchema(ss);

Validator v = schema.newValidator();
v.validate( new StreamSource(xml) );
```

A faster method of validation when using SAX is to replace the last two lines above with this:

```
SAXParserFactory spf = SAXParserFactory.newInstance();
spf.setSchema( schema );
SAXParser parser = spf.newSAXParser();
parser.parse( XML_DOCUMENT );
```

Java 5 and Java 6 include many enhancements to Java's XML capabilities and performance. Java 6 includes the *Java XML Digital Signature* API. This API allows you to generate and validate XML digital signatures. XML signatures are a standard for digital signatures in the XML data format, and they allow you to authenticate and protect the integrity of data in XML and web service transactions. (See java.sun.com/developer/technicalArticles/xml/dig_signature_api.)

JSON — JavaScript Object Notation

JSON is a syntax for storing and exchanging text information, much like XML. But JSON files are smaller than XML ones, and they can be faster to create or parse (read) than XML.

While JSON uses JavaScript syntax for describing data objects, it is language and platform independent. JSON parsers and JSON libraries exists for many different programming languages, making it a good choice for file or message formats. The details can be found at JSON.org, and in RFC-4627.

JSON is one of two things: a collection of name-value pairs, or a list of values. The values can be anything, including other collections or lists. A value can be a string in double quotes, or a number, or `true` or `false` or `null`, or an object or an array. Collections are enclosed in curly braces, using a colon between the name and value, lists in square braces, and items in a collection or list are separated with commas. You can have arbitrary white-space between any tokens, to make the results human-readable. That's it! Here's an example (from Adobe Labs):

```
{ "id": "0001", "type": "donut", "name": "Cake",
 "ppu": 0.55, "batters": {
   "batter":
   [{ "id": "1001", "type": "Regular" },
    { "id": "1002", "type": "Chocolate" },
    { "id": "1003", "type": "Blueberry" },
    { "id": "1004", "type": "Devil's Food" }
   ] },
 "topping":[
   { "id": "5001", "type": "None" },
   { "id": "5002", "type": "Glazed" },
   { "id": "5005", "type": "Sugar" },
   { "id": "5007", "type": "Powdered Sugar" },
   { "id": "5006",
     "type": "Chocolate with Sprinkles" },
   { "id": "5003", "type": "Chocolate" },
   { "id": "5004", "type": "Maple" }
  ]
}
```

(Note this example includes inconsistant whate-space, and also no numbers, just strings.)

Currently, only Java EE has built-in support for JSON (See JAXB). However, it was voted to add JSON to Java SE in 1/2012 (JSR 353); look for it in Java 8.

In the meantime, you need to download one of the many free JSON libraries available to use JSON easily now. One of the most popular is called **JSON-lib**, available at json-lib.sourceforge.net. To use, download the jar file for this and add it to your JRE's extentions directory. However, you also need several additional jar files, available from Apache.org.

A simpler, stand-alone library is **`org.json`**, available as a zip of source files from github.com/douglascrockford/JSON-java. I have compiled these and added them to a jar file; just download it into your ext directory. The Java docs for `org.json` are available at json.org/java.

Using org.json

To construct new JSON objects, use `JSONArray` to construct JSON lists, and `JSONObject` to construct JSON collections. The JSON arrays and objects have useful `toString` methods, making it easy to write JSON to files. These classes can also read in ("parse") JSON and build `JSONObjects`, which can then be easily changed or converted to regular Java objects. Finally, this library supports converting JSON to and from XML. (***Show JSONdemo.java***)