DAY 19: Reading and Writing RSS Feeds

Today you work with Extensible Markup Language (XML), a formatting standard that enables data to be completely portable.

You'll explore XML in the following ways:

- Representing data as XML
- Discovering why XML is a useful way to store data
- Using XML to publish web content
- Reading and writing XML data

The XML format employed throughout the day is Really Simple Syndication (RSS), a popular way to publish web content and share information on site updates adopted by millions of sites.

Using XML

One of Java's main selling points is that the language produces programs that can run on different operating systems without modification. The portability of software is a big convenience in today's computing environment, where Windows, Linux, Mac OS, and a half dozen other operating systems are in wide use and many people work with multiple systems.

XML, which stands for Extensible Markup Language, is a format for storing and organizing data that is independent of any software program that works with the data.

Data that is compliant with XML is easier to reuse for several reasons.

First, the data is structured in a standard way, making it possible for software programs to read and write the data as long as they support XML. If you create an XML file that represents your company's employee database, there are several dozen XML parsers that can read the file and make sense of its contents.

This is true no matter what kind of information you collect about each employee. If your database contains only the employee's name, ID number, and current salary, XML parsers can read it. If it contains 25 items, including birthday, blood type, and hair color, parsers can read that, too.

Second, the data is self-documenting, making it easier for people to understand the purpose of a file just by looking at it in a text editor. Anyone who opens your XML employee database should be able to figure out the structure and content of each employee record without any assistance from you.

This is evident in Listing 19.1, which contains an RSS file. Because RSS is an XML dialect, it is structured under the rules of XML.

LISTING 19.1 The Full Text of `workbench.rss`

```
<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0">
  <channel>
    <title>Workbench</title>
    <link>http://www.cadenhead.org/workbench/</link>
    <description>Programming, publishing, politics, and popes</description>
    <docs>http://www.rssboard.org/rss-specification</docs>
    <item>
      <title>Toronto Star: Only 100 Blogs Make Money</title>
      <link>http://www.cadenhead.org/workbench/news/3132</link>
      <pubDate>Mon, 26 Feb 2007 11:30:57 -0500</pubDate>
      <guid isPermaLink="false">tag:cadenhead.org,2007:weblog.3132</guid>
      <enclosure length="2498623" type="audio/mpeg"
```

LISTING 19.1 Continued

```
14:           url="http://mp3.cadenhead.org/3132.mp3" />
15:     </item>
16:     <item>
17:       <title>Eliot Spitzer Files UDRP to Take EliotSpitzer.Com</title>
18:       <link>http://www.cadenhead.org/workbench/news/3130</link>
19:       <pubDate>Thu, 22 Feb 2007 18:02:53 -0500</pubDate>
20:       <guid isPermaLink="false">tag:cadenhead.org,2007:weblog.3130</guid>
21:     </item>
22:     <item>
23:       <title>Fuzzy Zoeller Sues Over Libelous Wikipedia Page</title>
24:       <link>http://www.cadenhead.org/workbench/news/3129</link>
25:       <pubDate>Thu, 22 Feb 2007 13:48:45 -0500</pubDate>
26:       <guid isPermaLink="false">tag:cadenhead.org,2007:weblog.3129</guid>
27:     </item>
28:   </channel>
29: </rss>
```

Enter this text using a word processor or text editor and save it as plain text under the name `workbench.rss`. (You can also download a copy of it from the book's website at http://www.java21days.com on the Day 19 page.)

Can you tell what the data represents? Although the `?xml` tag at the top might be indecipherable, the rest is clearly a website database of some kind.

The `?xml` tag in the first line of the file has a `version` attribute with a value of `1.0` and an `encoding` attribute of `"utf-8"`. This establishes that the file follows the rules of XML 1.0 and is encoded with the UTF-8 character set.

Data in XML is surrounded by tag elements that describe the data. Opening tags begin with a "<" character followed by the name of the tag and a ">" character. Closing tags begin with the "</" characters followed by a name and a ">" character. In Listing 19.1, for example, `<item>` on line 8 is an opening tag, and `</item>` on line 15 is a closing tag. Everything within those tags is considered to be the value of that element.

Elements can be nested within other elements, creating a hierarchy of XML data that establishes relationships within that data. In Listing 19.1, everything in lines 9–14 is related; each element defines something about the same website item.

Elements also can include attributes, which are made up of data that supplements the rest of the data associated with the element. Attributes are defined within an opening tag element. The name of an attribute is followed by an equal sign and text within quotation marks.

In line 12 of Listing 19.1, the `guid` element includes an `isPermaLink` attribute with a value of `"false"`. This indicates that the element's value, "`tag:cadenhead.org,2007:weblog.3132`", is not a *permalink*, the URL at which the item can be loaded in a browser.

XML also supports elements defined by a single tag rather than a pair of tags. The tag begins with a "<" character followed by the name of the tag and ends with the "/>" characters. The RSS file includes an `enclosure` element in lines 13–14 that describes an MP3 audio file associated with the item.

XML encourages the creation of data that's understandable and usable even if the user doesn't have the program that created it and cannot find any documentation that describes it.

The purpose of the RSS file in Listing 19.1 can be understood, for the most part, simply by looking at it. Each item represents a web page that has been updated recently.

TIP

Publishing new site content over RSS and a similar format, Atom, has become one of the best ways to build readership on the Web. Thousands of people subscribe to RSS files, which are called feeds, using reader software such as Google Reader, Bloglines, and My Yahoo.

Rogers Cadenhead, the lead author of this book, is the current chairman of the RSS Advisory Board, the group that publishes the RSS 2.0 specification. For more information on the format, visit the board's site at http://www.rssboard.org or subscribe to its RSS feed at http://www.rssboard.org/rss-feed.

Data that follows XML's formatting rules is said to be *well-formed*. Any software that can work with XML reads and writes well-formed XML data.

NOTE

By insisting on well-formed markup, XML simplifies the task of writing programs that work with the data. RSS makes website updates available in a form that's easily processed by software. The RSS feed for Workbench at http://www.cadenhead.org/workbench/rss, published by one of this book's authors, has two distinct audiences: humans reading the blog through their preferred RSS reader and computers that do something with this data, such as Technorati, which offers a searchable database of site updates, links between different blogs, and categorization. To see how Technorati uses that RSS feed, visit `http://technorati.com/blogs/cadenhead.org/workbench`.

Designing an XML Dialect

Although XML is described as a language and is compared with Hypertext Markup Language (HTML), it's actually much larger in scope than that. XML is a markup language that defines how to define a markup language.

That's an odd distinction to make, and it sounds like the kind of thing you'd encounter in a philosophy textbook. This concept is important to understand, though, because it explains how XML can be used to define data as varied as health-care claims, genealogical records, newspaper articles, and molecules.

The "X" in XML stands for Extensible, and it refers to organizing data for your own purposes. Data that's organized using the rules of XML can represent anything you want:

- A programmer at a telemarketing company can use XML to store data on each outgoing call, saving the time of the call, the number, the operator who made the call, and the result.
- A lobbyist can use XML to keep track of the annoying telemarketing calls she receives, noting the time of the call, the company, and the product being peddled.
- A programmer at a government agency can use XML to track complaints about telemarketers, saving the name of the marketing firm and the number of complaints.

Each of these examples uses XML to define a new language that suits a specific purpose. Although you could call them XML languages, they're more commonly described as *XML dialects* or *XML document types*.

An XML dialect can be designed using a Document Type Definition (DTD) that indicates the potential elements and attributes that it covers.

A special `!DOCTYPE` declaration can be placed in XML data, right after the initial `?xml` tag, to identify its DTD. Here's an example:

```
<!DOCTYPE Library SYSTEM "librml.dtd">
```

The `!DOCTYPE` declaration is used to identify the DTD that applies to the data. When a DTD is present, many XML tools can read XML created for that DTD and determine whether the data follows all the rules correctly. If it doesn't, it is rejected with a reference to the line that caused the error. This process is called *validating the XML*.

One thing you'll run into as you work with XML is data that has been structured as XML but wasn't defined using a DTD. Most versions of RSS files do not require a DTD. This data can be parsed (presuming it's well-formed), so you can read it into a program and do something with it, but you can't check its validity to make sure that it's organized correctly according to the rules of its dialect.

TIP

To get an idea of what kind of XML dialects have been created, Cover Pages offers a list at http://xml.coverpages.org/xmlApplications.html.

Processing XML with Java

Java supports XML through the Java API for XML Processing, a set of packages for reading, writing, and manipulating XML data.

The `javax.xml.parsers` package is the entry point to the other packages. These classes can be used to parse and validate XML data using two techniques: the Simple API for XML (SAX) and the Document Object Model (DOM). However, they can be difficult to implement, which has inspired other groups to offer their own class libraries to work with XML.

You'll spend the remainder of the day working with one of these alternatives: the XML Object Model (XOM) library, an open source Java class library that makes it extremely easy to read, write, and transform XML data.

NOTE

To find out more about the Java API for XML Processing, visit the company's Java website at http://java.sun.com/xml.

Processing XML with XOM

One of the most important skills you can develop as a Java programmer is the ability to find suitable packages and classes that can be employed in your own projects. For obvious reasons, making use of a well-designed class library is much easier than developing one on your own.

Although Sun's Java class library contains thousands of well-designed classes that cover a comprehensive range of development needs, the company isn't the only supplier of packages that may prove useful to your efforts.

Dozens of Java packages are offered by other companies, groups, and individuals under a variety of commercial and open source licenses. Some of the most notable come from Apache Jakarta, a Java development project of the Apache Software Foundation that has produced the web application framework Struts, the Log4J logging class library, and many other popular libraries.

Another terrific open source Java class library is the XOM library, a tree-based package for XML processing that strives to be simple to learn, simple to use, and uncompromising in its adherence to well-formed XML.

The library was developed by the programmer and author Elliotte Rusty Harold based on his experience with Sun's XML processing packages and other efforts to handle XML in Java.

The project was originally envisioned as a fork of JDOM, a popular tree-based model for representing an XML document. Harold has contributed code to that open source project and participated in its development.

Instead of forking the JDOM code, Harold decided to start from scratch and adopt some of its core design principles in XOM.

The library embodies the following principles:

- XML documents are modeled as a tree with Java classes representing nodes on the tree such as elements, comments, processing instructions, and document type definitions. A programmer can add and remove nodes to manipulate the document in memory, a simple approach that can be implemented gracefully in Java.
- All XML data produced by XOM is well-formed and has a well-formed namespace.
- Each element of an XML document is represented as a class with constructor methods.
- Object serialization is not supported. Instead, programmers are encouraged to use XML as the format for serialized data, enabling it to be readily exchanged with any software that reads XML regardless of the programming language in which it was developed.
- The library relies on another XML parser to read XML documents and fill trees instead of doing this low-level work directly. XOM uses a SAX parser that must be downloaded and installed separately. Right now, the preferred parser is Apache Xerces 2.7.1.

XOM is available for download from the web address http://www.cafeconleche.org/XOM. The most current version at this writing is 1.1, which includes Xerces 2.7.1 in its distribution.

CAUTION

XOM is released according to the terms of the open source GNU Lesser General Public License (LGPL). The license grants permission to distribute the library without modification with Java programs that use it.

You also can make changes to the XOM class library as long as you offer them under the LGPL. The full license is published online at http://www.cafeconleche.org/XOM/license.xhtml.

After you have downloaded XOM and added its packages to your system's `Classpath`, you're ready to begin using XOM.

The full installation instructions are available from the XOM and Xerces websites. The classes are distributed as JAR archive files—`xom-1.1.jar`, `xercesImpl.jar`, and `xml-apis.jar`. These files should be added to your system's `Classpath` environment variable so that your Java programs can use XOM classes.

Creating an XML Document

The first application you will create, `RssWriter`, creates an XML document that contains the start of an RSS feed. The document is shown in Listing 19.2.

LISTING 19.2 The Full Text of feed.rss

```
<?xml version="1.0"?>
<rss version="2.0">
  <channel>
    <title>Workbench</title>
    <link>http://www.cadenhead.org/workbench/</link>
  </channel>
</rss>
```

The base `nu.xom` package contains classes for a complete XML document (`Document`) and the nodes a document can contain (`Attribute`, `Comment`, `DocType`, `Element`, `ProcessingInstruction`, and `Text`).

The `RssStarter` application uses several of these classes. First, an `Element` object is created by specifying the element's name as an argument:

```
Element rss = new Element("rss");
```

This statement creates an object for the root element of the document, `rss`. `Element`'s one-argument constructor can be used because the document does not employ a feature of XML called namespaces; if it did, a second argument would be necessary: the namespace uniform resource identifier (URI) of the element. The other classes in the XOM library support namespaces in a similar manner.

In the XML document in Listing 19.2, the `rss` element includes an attribute named `version` with the value "2.0". An attribute can be created by specifying its name and value in consecutive arguments:

```
Attribute version = new Attribute("version", "2.0");
```

Attributes are added to an element by calling its `addAttribute()` method with the attribute as the only argument:

```
rss.addAttribute(version);
```

The text contained within an element is represented by the `Text` class, which is constructed by specifying the text as a `String` argument:

```
Text titleText = new Text("Workbench");
```

When composing an XML document, all of its elements end up inside a root element that is used to create a `Document` object—a `Document` constructor is called with the root element as an argument. In the `RssStarter` application, this element is called `rss`. Any `Element` object can be the root of a document:

```
Document doc = new Document(rss);
```

In XOM's tree structure, the classes representing an XML document and its constituent parts are organized into a hierarchy below the generic superclass `nu.xom.Node`. This class has three subclasses in the same package: `Attribute`, `LeafNode`, and `ParentNode`.

To add a child to a parent node, call the parent's `appendChild()` method with the node to add as the only argument. The following code creates three elements—a parent called `domain` and two of its children, `name` and `dns`:

```
Element channel = new Element("channel");
Element link = new Element("link");
Text linkText = new Text("http://www.cadenhead.org/workbench/");
link.appendChild(linkText);
channel.appendChild(link);
```

The `appendChild()` method appends a new child below all other children of that parent. The preceding statements produce this XML fragment:

```
<channel>
   <link>http://www.cadenhead.org/workbench/</link>
</channel>
```

The `appendChild()` method also can be called with a `String` argument instead of a node. A `Text` object representing the string is created and added to the element:

```
link.appendChild("http://www.cadenhead.org/workbench/");
```

After a tree has been created and filled with nodes, it can be displayed by calling the `Document` method `toXML()`, which returns the complete and well-formed XML document as a `String`.

Listing 19.3 shows the complete application.

LISTING 19.3 The Full text of `RssStarter.java`

```
import nu.xom.*;

public class RssStarter {
    public static void main(String[] arguments) {
        // create an <rss> element to serve as the document's root
        Element rss = new Element("rss");

        // add a version attribute to the element
        Attribute version = new Attribute("version", "2.0");
        rss.addAttribute(version);
        // create a <channel> element and make it a child of <rss>
        Element channel = new Element("channel");
        rss.appendChild(channel);
        // create the channel's <title>
        Element title = new Element("title");
        Text titleText = new Text("Workbench");
        title.appendChild(titleText);
        channel.appendChild(title);
        // create the channel's <link>
        Element link = new Element("link");
        Text linkText = new Text("http://www.cadenhead.org/workbench/");
        link.appendChild(linkText);
        channel.appendChild(link);

        // create a new document with <rss> as the root element
        Document doc = new Document(rss);

        // Display the XML document
```

LISTING 19.3 Continued

```
        System.out.println(doc.toXML());
    }
}
```

The `RssStarter` application displays the XML document it creates on standard output. The following command runs the application and redirects its output to a file called `feed.rss`:

```
java RssStarter > feed.rss
```

XOM automatically precedes a document with an XML declaration.

The XML produced by this application contains no indentation; elements are stacked on the same line.

XOM only preserves significant whitespace when representing XML data—the spaces between elements in the RSS feed contained in Listing 19.2 are strictly for presentation purposes and are not produced automatically when XOM creates an XML document. A subsequent example demonstrates how to control indentation.

Modifying an XML Document

The next project, the `DomainEditor` application, makes several changes to the XML document that was just produced by the `RssStarter` application, `feed.rss`. The text enclosed by the `link` element is changed to “http://www.cadenhead.org/”, and a new `item` element is added:

```
<item>
  <title>Fuzzy Zoeller Sues Over Libelous Wikipedia Page</title>
</item>
```

Using the `nu.xom` package, XML documents can be loaded into a tree from several sources: a `File`, `InputStream`, `Reader`, or a URL (which is specified as a `String` instead of a `java.net.URL` object).

The `Builder` class represents a SAX parser that can load an XML document into a `Document` object. Constructor methods can be used to specify a particular parser or let XOM use the first available parser from this list: Xerces 2, Crimson, Piccolo, GNU Aelfred, Oracle, XP, Saxon Aelfred, or Dom4J Aelfred. If none of these is found, the parser specified by the system property `org.xml.sax.driver` is used. Constructors also determine whether the parser is validating or nonvalidating.

The `Builder()` and `Builder(true)` constructors both use the default parser—most likely a version of Xerces. The presence of the Boolean argument `true` in the second constructor configures the parser to be validating. It would be nonvalidating otherwise. A validating parser throws a `nu.xom.ValidityException` if the XML document doesn't validate according to the rules of its document type definition.

The `Builder` object's `build()` method loads an XML document from a source and returns a `Document` object:

```
Builder builder = new Builder();
File xmlFile = new File("feed.rss");
Document doc = builder.build(xmlFile);
```

These statements load an XML document from the file `feed.rss` barring one of two problems: A `nu.xom.ParseException` is thrown if the file does not contain well-formed XML, and a `java.io.IOException` is thrown if the input operation fails.

Elements are retrieved from the tree by calling a method of their parent node.

A `Document` object's `getRootElement()` method returns the root element of the document:

```
Element root = doc.getRootElement();
```

In the XML document `feed.rss`, the root element is `domains`.

Elements with names can be retrieved by calling their parent node's `getFirstChildElement()` method with the name as a `String` argument:

```
Element channel = root.getFirstChildElement("channel");
```

This statement retrieves the `channel` element contained in the `rss` element (or `null` if that element could not be found). Like other examples, this is simplified by the lack of a namespace in the document; there are also methods where a name and namespace are arguments.

When several elements within a parent have the same name, the parent node's `getChildElements()` method can be used instead:

```
Elements children = channel.getChildElements()
```

The `getChildElements()` method returns an `Elements` object containing each of the elements. This object is a read-only list and does not change automatically if the parent node's contents change after `getChildElements()` is called.

`Elements` has a `size()` method containing an integer count of the elements it holds. This can be used in a loop to cycle through each element in turn beginning with the one at position 0. There's a `get()` method to retrieve each element; call it with the integer position of the element to be retrieved:

```
for (int i = 0; i < children.size(); i++) {    Element link = children.get(i);
}
```

This `for` loop cycles through each `child` element of the `channel` element.

Elements without names can be retrieved by calling their parent node's `getChild()` method with one argument: an integer indicating the element's position within the parent node:

```
Text linkText = (Text) link.getChild(0);
```

This statement creates the `Text` object for the text "http://www.cadenhead.org/workbench/" found within the `link` element. `Text` elements always will be at position 0 within their enclosing parent.

To work with this text as a string, call the `Text` object's `getValue()` method, as in this statement:

```
if (linkText.getValue().equals("http://www.cadenhead.org/workbench/"))
    // ...
}
```

The `DomainEditor` application only modifies a `link` element enclosing the text "http://www.cadenhead.org/workbench/". The application makes the following changes: The text of the `link` element is deleted, the new text "http://www.cadenhead.org/" is added in its place, and then a new `item` element is added.

A parent node has two `removeChild()` methods to delete a child node from the document. Calling the method with an integer deletes the child at that position:

```
Element channel = domain.getFirstChildElement("channel");
Element link = dns.getFirstChildElement("link");
link.removeChild(0);
```

These statements would delete the `Text` object contained within the channel's first `link` element.

Calling the `removeChild()` method with a node as an argument deletes that particular node. Extending the previous example, the `link` element could be deleted with this statement:

```
channel.removeChild(link);
```

Listing 19.4 shows the source code of the `DomainEditor` application.

LISTING 19.4 The Full Text of `DomainEditor.java`

```java
import java.io.*;
import nu.xom.*;

public class DomainEditor {
    public static void main(String[] arguments) throws IOException {
        try {
            // create a tree from the XML document feed.rss
            Builder builder = new Builder();
            File xmlFile = new File("feed.rss");
            Document doc = builder.build(xmlFile);

            // get the root element <rss>
            Element root = doc.getRootElement();

            // get its <channel> element
            Element channel = root.getFirstChildElement("channel");

            // get its <link> elements
            Elements children = channel.getChildElements();
            for (int i = 0; i < children.size(); i++) {

                // get a <link> element
                Element link = children.get(i);

                // get its text
                Text linkText = (Text) link.getChild(0);

                // update any link matching a URL
                if (linkText.getValue().equals(
                    "http://www.cadenhead.org/workbench/")) {

                    // update the link's text
                    link.removeChild(0);
                    link.appendChild("http://www.cadenhead.org/");
                }
            }

            // create new elements and attributes to add
            Element item = new Element("item");
            Element itemTitle = new Element("title");

            // add them to the <channel> element
            itemTitle.appendChild(
                "Fuzzy Zoeller Sues Over Libelous Wikipedia Page"
            );
            item.appendChild(itemTitle);
```

LISTING 19.4 Continued

```
            channel.appendChild(item);

            // display the XML document
            System.out.println(doc.toXML());
        } catch (ParsingException pe) {
            System.out.println("Error parsing document: " + pe.getMessage());
            pe.printStackTrace();
            System.exit(-1);
        }
    }
}
```

The `DomainEditor` application displays the modified XML document to standard output, so it can be run with the following command to produce a file named `feeds2.rss`:

```
java DomainEditor > feed2.rss
```

Formatting an XML Document

As described earlier, XOM does not retain insignificant whitespace when representing XML documents. This is in keeping with one of XOM's design goals—to disregard anything that has no syntactic significance in XML. (Another example of this is how text is treated identically whether created using character entities, `CDATA` sections, or regular characters.)

Today's next project is the `DomainWriter` application, which adds a comment to the beginning of the XML document `feeds2.rss` and serializes it with indented lines, producing the version shown in Listing 19.5.

LISTING 19.5 The Full Text of `feeds2.rss`

```
<?xml version="1.0"?>
<rss version="2.0">
  <channel>
    <title>Workbench</title>
    <link>http://www.cadenhead.org/</link>
    <item>
      <title>Fuzzy Zoeller Sues Over Libelous Wikipedia Page</title>
    </item>
  </channel>
</rss>
```

The Serializer class in nu.xom offers control over how an XML document is formatted when it is displayed or stored serially. Indentation, character encoding, line breaks, and other formatting are established by objects of this class.

A Serializer object can be created by specifying an output stream and character encoding as arguments to the constructor:

```
File inFile = new File(arguments[0]);
FileOutputStream fos = new FileOutputStream("new_" +
    inFile.getName());
Serializer output = new Serializer(fos, "ISO-8859-1");
```

These statements serialize a file using the ISO-8859-1 character encoding. The file is given a name based on a command-line argument.

Serializer supports 20 encodings, including ISO-10646-UCS-2, ISO-8859-1 through ISO-8859-10, ISO-8859-13 through ISO-8859-16, UTF-8, and UTF-16. There's also a Serializer() constructor that takes only an output stream as an argument; this uses the UTF-8 encoding by default.

Indentation is set by calling the serializer's setIndentation() method with an integer argument specifying the number of spaces:

```
output.setIndentation(2);
```

An entire XML document is written to the serializer destination by calling the serializer's write() method with the document as an argument:

```
output.write(doc);
```

The DomainWriter application inserts a comment atop the XML document instead of appending it at the end of a parent node's children. This requires another method of the parent node, insertChild(), which is called with two arguments—the element to add and the integer position of the insertion:

```
Builder builder = new Builder();
Document doc = builder.build(arguments[0]);
Comment timestamp = new Comment("File created " +
    new java.util.Date());
doc.insertChild(timestamp, 0);
```

The comment is placed at position 0 atop the document, moving the domains tag down one line but remaining below the XML declaration.

Listing 19.6 contains the source code of the application.

LISTING 19.6 The Full Text of `DomainWriter.java`

```
import java.io.*;
import nu.xom.*;

public class DomainWriter {
    public static void main(String[] arguments) throws IOException {
        try {
            // Create a tree from an XML document
            // specified as a command-line argument
            Builder builder = new Builder();
            Document doc = builder.build(arguments[0]);

            // Create a comment with the current time and date
            Comment timestamp = new Comment("File created "
                + new java.util.Date());

            // Add the comment above everything else in the
            // document
            doc.insertChild(timestamp, 0);

            // Create a file output stream to a new file
            File inFile = new File(arguments[0]);
            FileOutputStream fos = new FileOutputStream("new_" +
inFile.getName());

            // Using a serializer with indention set to 2 spaces,
            // write the XML document to the file
            Serializer output = new Serializer(fos, "ISO-8859-1");
            output.setIndent(2);
            output.write(doc);
        } catch (ParsingException pe) {
            System.out.println("Error parsing document: " + pe.getMessage());
            pe.printStackTrace();
            System.exit(-1);
        }
    }
}
```

The `DomainWriter` application takes an XML filename as a command-line argument when run:

```
java DomainWriter feeds2.rss
```

This command produces a file called `new_feeds2.rss` that contains an indented copy of the XML document with a time stamp inserted as a comment.

Evaluating XOM

These three sample applications cover the core features of the main XOM package and are representative of its straightforward approach to XML processing.

There also are smaller `nu.xom.canonical`, `nu.xom.converters`, `nu.xom.xinclude`, and `nu.xom.xslt` packages to support XInclude, XSLT, canonical XML serialization, and conversions between the XOM model for XML and the one used by DOM and SAX.

Listing 19.7 contains an application that works with XML from a dynamic source: RSS feeds of recently updated web content from the producer of the feed. The `RssFilter` application searches the feed for specified text in headlines, producing a new XML document that contains only the matching items and shorter indentation. It also modifies the feed's title and adds an RSS 0.91 document type declaration if one is needed in an RSS 0.91 format feed.

LISTING 19.7 The Full Text of `RssFilter.java`

```
import nu.xom.*;

public class RssFilter {
    public static void main(String[] arguments) {

        if (arguments.length < 2) {
            System.out.println("Usage: java RssFilter rssFile searchTerm");
            System.exit(-1);
        }

        // Save the RSS location and search term
        String rssFile = arguments[0];
        String searchTerm = arguments[1];

        try {
            // Fill a tree with an RSS file's XML data
            // The file can be local or something on the
            // Web accessible via a URL.
            Builder bob = new Builder();
            Document doc = bob.build(rssFile);

            // Get the file's root element (<rss>)
            Element rss = doc.getRootElement();

            // Get the element's version attribute
            Attribute rssVersion = rss.getAttribute("version");
            String version = rssVersion.getValue();

            // Add the DTD for RSS 0.91 feeds, if needed
            if ( (version.equals("0.91")) & (doc.getDocType() == null) ) {
```

LISTING 19.7 Continued

```
                DocType rssDtd = new DocType("rss",
                    "http://my.netscape.com/publish/formats/rss-0.91.dtd");
                doc.insertChild(rssDtd, 0);
            }

            // Get the first (and only) <channel> element
            Element channel = rss.getFirstChildElement("channel");

            // Get its <title> element
            Element title = channel.getFirstChildElement("title");
            Text titleText = (Text)title.getChild(0);

            // Change the title to reflect the search term
            titleText.setValue(titleText.getValue() + ": Search for " +
                searchTerm + " articles");

            // Get all of the <item> elements and loop through them
            Elements items = channel.getChildElements("item");
            for (int i = 0; i < items.size(); i++) {
                // Get an <item> element
                Element item = items.get(i);

                // Look for a <title> element inside it
                Element itemTitle = item.getFirstChildElement("title");

                // If found, look for its contents
                if (itemTitle != null) {
                    Text itemTitleText = (Text) itemTitle.getChild(0);

                    // If the search text is not found in the item,
                    // delete it from the tree
                    if (itemTitleText.toString().indexOf(searchTerm) == -1)
                        channel.removeChild(item);
                }
            }

            // Display the results with a serializer
            Serializer output = new Serializer(System.out);
            output.setIndent(2);
            output.write(doc);
        } catch (Exception exc) {
            System.out.println("Error: " + exc.getMessage());
            exc.printStackTrace();
        }
    }
}
```

One feed that can be used to test the application is the one from the *Toronto Star* newspaper. The following command searches it for items with titles that mention the word "snow":

```
java RssFilter http://www.thestar.com/rss/000-082-672?searchMode=Lineup snow
```

Comments in the application's source code describe its functionality.

XOM's design is strongly informed by one overriding principle: enforced simplicity.

On the website for the class library, Harold states that XOM "should help inexperienced developers do the right thing and keep them from doing the wrong thing. The learning curve needs to be really shallow, and that includes not relying on best practices that are known in the community but are not obvious at first glance."

The new class library is useful for Java programmers whose Java programs require a steady diet of XML.

Summary

Today, you learned the basics of another popular format for data representation, Extensible Markup Language (XML), by exploring one of the most popular uses of XML—RSS feeds.

In many ways, Extensible Markup Language is the data equivalent of the Java language. It liberates data from the software used to create it and the operating system the software ran on, just as Java can liberate software from a particular operating system.

By using a class library such as the open source XML Object Model (XOM) library, you can easily create and retrieve data from an XML file.

A big advantage to representing data using XML is that you will always be able to get that data back. If you decide to move the data into a relational database or some other form, you can easily retrieve the information. The data being produced as RSS feeds can be mined by software in countless ways, today and in the future.

You also can transform XML into other forms such as HTML through a variety of technology, both in Java and through tools developed in other languages.

Q&A

Q What's the difference between RSS 1.0, RSS 2.0, and Atom?

A RSS 1.0 is a syndication format that employs the Resource Description Framework (RDF) to describe items in the feed. RSS 2.0 shares a common origin with RSS 1.0 but does not make use of RDF. Atom is another syndication format that was created after the preceding two formats and has been adopted as an Internet standard by the IETF.

All three formats are suitable for offering web content in XML that can be read with a reader such as Bloglines or My Yahoo or read by software and stored, manipulated, or transformed.

Q Why is Extensible Markup Language called XML instead of EML?

A None of the founders of the language appears to have documented the reason for choosing XML as the acronym. The general consensus in the XML community is that it was chosen because it "sounds cooler" than EML. Before anyone snickers at that distinction, Sun Microsystems chose the name Java for its programming language using the same criteria, turning down more technical-sounding alternatives such as DNA and WRL.

There is a possibility that the founders of XML were trying to avoid confusion with a programming language called EML (Extended Machine Language), which predates Extensible Markup Language.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. What does RSS stand for?
    a. Really Simple Syndication
    b. RDF Site Summary
    c. Both
2. What method cannot be used to add text to an XML element using XOM?
    a. `addAttribute(`*`String`*`, `*`String`*`)`
    b. `appendChild(`*`Text`*`)`
    c. `appendChild(`*`String`*`)`

3. When all the opening element tags, closing element tags, and other markup are applied consistently in a document, what adjective describes the document?
   a. Validating
   b. Parsable
   c. Well-formed

Answers

1. **c.** One version, RSS 2.0, claims Really Simple Syndication as its name. The other, RSS 1.0, claims RDF Site Summary.
2. **a.** Answers b. and c. both work successfully. One adds the contents of a `Text` element as the element's character data. The other adds the string.
3. **c.** For data to be considered XML, it must be well-formed.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

```
public class NameDirectory {
    String[] names;
    int nameCount;

    public NameDirectory() {
        names = new String[20];
        nameCount = 0;
    }

    public void addName(String newName) {
        if (nameCount < 20)
            // answer goes here
    }
}
```

The `NameDirectory` class must be able to hold 20 different names. What statement should replace `// answer goes here` for the class to function correctly?

a. `names[nameCount] = newName;`

b. `names[nameCount] == newName;`

c. `names[nameCount++] = newName;`

d. `names[++nameCount] = newName;`

The answer is available on the book's website at http://www.java21days.com. Visit the Day 19 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Create a simple XML format to represent a book collection with three books and a Java application that searches for books with Joseph Heller as the author, displaying any that it finds.
2. Create two applications: one that retrieves records from a database and produces an XML file that contains the same information and a second application that reads data from that XML file and displays it.

Where applicable, exercise solutions are offered on the book's website at http://www.java21days.com.