The Java™ Web Services Tutorial

Eric Armstrong
Stephanie Bodoff
Debbie Carson
Maydene Fisher
Scott Fordin
Dale Green
Kim Haase
Eric Jendrock

February 20, 2003

Contents

About This Tutorial

THIS tutorial is a beginner's guide to developing Web services and Web applications using the Java™ Web Services Developer Pack (Java WSDP). The Java WSDP is an all-in-one download containing key technologies to simplify building of Web services using the Java 2 Platform. This tutorial requires a full installation (Typical, not Custom) of the Java WSDP.

Who Should Use This Tutorial

This tutorial is intended for programmers interested in developing and deploying Web services and Web applications on the Java WSDP.

How to Read This Tutorial

This tutorial is organized into five parts:

- Introduction

  The first five chapters introduce basic concepts and technologies and we suggest that you read these first in their entirety. In particular, many of the Java WSDP examples run on the Tomcat Java servlet and JSP container and the Getting Started with Tomcat chapter tells you how to start, stop, and manage Tomcat.

- Java XML Technology

  These chapters cover all the Java XML APIs.

  - The Java API for XML Processing (JAXP)
  - The Java Architecture for XML Binding (JAXB)

    - The Java API for XML Messaging (JAXM) and Soap with Attachments API for Java (SAAJ)
    - The Java API for XML-based RPC (JAX-RPC)
    - The Java API for XML Registries (JAXR) and the Registry Server, a UDDI-compliant registry accessible via JAXR

- Web Technology

  These chapters cover the technologies used in developing presentation-oriented Web applications.

    - Java Servlets
    - JavaServer™ Pages (JSP)
    - Custom tags and the JSP Standard Tag Library (JSTL)

- Case Study

  The Coffee Break Application chapter in this part describes an application that ties together most of the APIs discussed in this tutorial.

- Appendixes

  The appendixes cover the tools shipped with the Java WSDP.

    - Tomcat Server Administration Tool
    - Tomcat Web Application Manager
    - JAXM Provider Admin
    - Registry Browser

  This part also includes appendixes on HTTP and Java encoding schemes.

About the Examples

Prerequisites for the Examples

To understand the examples you will need a good knowledge of the Java programming language, SQL, and relational database concepts. The topics in the *The Java Tutorial* that are particularly relevant are listed in Table P–1:

Table P–1 Relevant Topics in *The Java™ Tutorial*

Topic	**Web Page**
JDBC™ technology	`http://java.sun.com/docs/books/tutorial/jdbc`
Threads	`http://java.sun.com/docs/books/tutorial/essential/threads`
JavaBeans™ architecture	`http://java.sun.com/docs/books/tutorial/javabeans`
Security	`http://java.sun.com/docs/books/tutorial/security1.2`

Running the Examples

This section tells you everything you need to know to obtain, build, install, and run the examples.

Required Software

If you are viewing this online, you need to download *The Java Web Services Tutorial* from:

```
http://java.sun.com/webservices/downloads/webservicestutorial.html
```

Once you have installed the tutorial bundle using a Typical installation (installed all of the components), the example source code is in the `<JWSDP_HOME>/docs/tutorial/examples` directory, with subdirectories for each of the technologies included in the pack.

This tutorial documents the Java WSDP 1.1. To build, deploy, and run the examples you need a copy of the Java WSDP and the Java 2 Software Development Kit, Standard Edition (J2SE™ SDK) 1.3.1_07, 1.4.0_03, or 1.4.1_01. You download the Java WSDP from:

```
http://java.sun.com/webservices/downloads/webservicespack.html
```

the J2SE 1.3.1 SDK from

```
http://java.sun.com/j2se/1.3/
```

or the J2SE 1.4 SDK from

```
http://java.sun.com/j2se/1.4/
```

Add the `bin` directories of the Java WSDP and J2SE SDK installations to the front of your `PATH` environment variable so that the Java WSDP startup scripts for Tomcat overrides other installations.

Building the Examples

Most of the examples are distributed with a build file for `Ant` 1.5.1, a portable build tool contained in the Java WSDP. Directions for building the examples are provided in each chapter.

The version of `Ant` shipped with the Java WSDP sets the `jwsdp.home` environment variable, which is required by the example build files. To ensure that you use this version of `Ant`, you must add `<`*`JWSDP_HOME`*`>/jakarta-ant-1.5.1/bin` to the front of your `PATH`.

Managing the Examples

Many of the Java WSDP examples run on the Tomcat Java servlet and JSP container. You use the `manager` tool to install, list, reload, and remove Web applications. See Appendix B for information on this tool.

How to Print This Tutorial

To print this tutorial, follow these steps:

1. Ensure that Adobe Acrobat Reader is installed on your system.

2. Open the PDF version of this book.
3. Click the printer icon in Adobe Acrobat Reader.

Typographical Conventions

Table P–2 lists the typographical conventions used in this tutorial.

Table P–2 Typographical Conventions

Font Style	**Uses**
italic	Emphasis, titles, first occurrence of terms
`monospace`	URLs, code examples, file names, command names, programming language keywords
`italic monospace`	Variable file names

Menu selections indicated with the right-arrow character →, for example, First→Second, should be interpreted as: select the First menu, then choose Second from the First submenu.

1

Introduction to Web Services

Maydene Fisher and Eric Jendrock

WEB services, in the general meaning of the term, are services offered via the Web. In a typical Web services scenario, a business application sends a request to a service at a given URL using the SOAP protocol over HTTP. The service receives the request, processes it, and returns a response. An often-cited example of a Web service is that of a stock quote service, in which the request asks for the current price of a specified stock, and the response gives the stock price. This is one of the simplest forms of a Web service in that the request is filled almost immediately, with the request and response being parts of the same method call.

Another example could be a service that maps out an efficient route for the delivery of goods. In this case, a business sends a request containing the delivery destinations, which the service processes to determine the most cost-effective delivery route. The time it takes to return the response depends on the complexity of the routing, so the response will probably be sent as an operation that is separate from the request.

Web services and consumers of Web services are typically businesses, making Web services predominantly business-to-business (B-to-B) transactions. An enterprise can be the provider of Web services and also the consumer of other Web services. For example, a wholesale distributor of spices could be in the consumer role when it uses a Web service to check on the availability of vanilla beans and in the provider role when it supplies prospective customers with different vendors' prices for vanilla beans.

The Role of XML and the Java™ Platform

Web services depend on the ability of parties to communicate with each other even if they are using different information systems. XML (Extensible Markup Language), a markup language that makes data portable, is a key technology in addressing this need. Enterprises have discovered the benefits of using XML for the integration of data both internally for sharing legacy data among departments and externally for sharing data with other enterprises. As a result, XML is increasingly being used for enterprise integration applications, both in tightly coupled and loosely coupled systems. Because of this data integration ability, XML has become the underpinning for Web-related computing.

Web services also depend on the ability of enterprises using different computing platforms to communicate with each other. This requirement makes the Java platform, which makes code portable, the natural choice for developing Web services. This choice is even more attractive as the new Java APIs for XML become available, making it easier and easier to use XML from the Java programming language. These APIs are summarized later in this introduction and explained in detail in the tutorials for each API.

In addition to data portability and code portability, Web services need to be scalable, secure, and efficient, especially as they grow. The Java 2 Platform, Enterprise Edition (J2EE™), is specifically designed to fill just such needs. It facilitates the really hard part of developing Web services, which is programming the infrastructure, or "plumbing." This infrastructure includes features such as security, distributed transaction management, and connection pool management, all of which are essential for industrial strength Web services. And because components are reusable, development time is substantially reduced.

Because XML and the Java platform work so well together, they have come to play a central role in Web services. In fact, the advantages offered by the Java APIs for XML and the J2EE platform make them the ideal combination for deploying Web services.

The APIs described in this tutorial complement and layer on top of the J2EE APIs. These APIs enable the Java community, developers, and tool and container vendors to start developing Web services applications and products using standard Java APIs that maintain the fundamental Write Once, Run Anywhere™ proposition of Java technology. The Java Web Services Developer Pack (Java WSDP) makes all these APIs available in a single bundle. The Java WSDP includes JAR files implementing these APIs as well as documentation and examples. The examples in the Java WSDP will run in the Tomcat container (included in the Java WSDP to help with ease of use), as well as in a Web container in a J2EE server once the Java WSDP JAR files are installed in the J2EE server, such as the Sun™ ONE Application Server (S1AS). Instructions on how to install the JAR files on the S1AS7 server are available in the Java WSDP documentation at `<JWSDP_HOME>/docs/jwsdpons1as7.html`.

The remainder of this introduction first gives a quick look at XML and how it makes data portable. Then it gives an overview of the Java APIs for XML, explaining what they do and how they make writing Web applications easier. It describes each of the APIs individually and then presents a scenario that illustrates how they can work together.

The tutorials that follow give more detailed explanations and walk you through how to use the Java APIs for XML to build applications for Web services. They also provide sample applications that you can run.

What Is XML?

The goal of this section is to give you a quick introduction to XML and how it makes data portable so that you have some background for reading the summaries of the Java APIs for XML that follow. Chapter 1 includes a more thorough and detailed explanation of XML and how to process it.

XML is an industry-standard, system-independent way of representing data. Like HTML (HyperText Markup Language), XML encloses data in tags, but there are significant differences between the two markup languages. First, XML tags relate to the meaning of the enclosed text, whereas HTML tags specify how to display the enclosed text. The following XML example shows a price list with the name and price of two coffees.

```
<priceList>
  <coffee>
    <name>Mocha Java</name>
    <price>11.95</price>
  </coffee>
  <coffee>
    <name>Sumatra</name>
    <price>12.50</price>
  </coffee>
</priceList>
```

The `<coffee>` and `</coffee>` tags tell a parser that the information between them is about a coffee. The two other tags inside the `<coffee>` tags specify that the enclosed information is the coffee's name and its price per pound. Because XML tags indicate the content and structure of the data they enclose, they make it possible to do things like archiving and searching.

A second major difference between XML and HTML is that XML is extensible. With XML, you can write your own tags to describe the content in a particular type of document. With HTML, you are limited to using only those tags that have been predefined in the HTML specification. Another aspect of XML's extensibility is that you can create a file, called a *schema*, to describe the structure of a particular type of XML document. For example, you can write a schema for a price list that specifies which tags can be used and where they can occur. Any XML document that follows the constraints established in a schema is said to conform to that schema.

Probably the most-widely used schema language is still the Document Type Definition (DTD) schema language because it is an integral part of the XML 1.0 specification. A schema written in this language is commonly referred to as a DTD. The DTD that follows defines the tags used in the price list XML document. It specifies four tags (elements) and further specifies which tags may occur (or are required to occur) in other tags. The DTD also defines the hierarchical structure of an XML document, including the order in which the tags must occur.

```
<!ELEMENT priceList (coffee)+>
<!ELEMENT coffee (name, price) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT price (#PCDATA) >
```

The first line in the example gives the highest level element, `priceList`, which means that all the other tags in the document will come between the `<priceList>` and `</priceList>` tags. The first line also says that the `priceList` element must contain one or more `coffee` elements (indicated by the plus sign). The second line specifies that each `coffee` element must contain both a `name` element and a `price` element, in that order. The third and fourth lines specify that the data between the tags `<name>` and `</name>` and between `<price>` and `</price>` is character data that should be parsed. The name and price of each coffee are the actual text that makes up the price list.

Another popular schema language is XML Schema, which is being developed by the World Wide Web (W3C) consortium. XML Schema is a significantly more powerful language than DTD, and with its passage into a W3C Recommendation in May of 2001, its use and implementations have increased. The community of developers using the Java platform has recognized this, and the expert group for the Java API for XML Processing (JAXP) has been working on adding support for XML Schema to the JAXP 1.2 specification. This release of the Java Web Services Developer Pack includes support for XML Schema.

What Makes XML Portable?

A schema gives XML data its portability. The `priceList` DTD, discussed previously, is a simple example of a schema. If an application is sent a `priceList` document in XML format and has the `priceList` DTD, it can process the document according to the rules specified in the DTD. For example, given the `priceList` DTD, a parser will know the structure and type of content for any XML document based on that DTD. If the parser is a validating parser, it will know that the document is not valid if it contains an element not included in the DTD, such as the element `<tea>`, or if the elements are not in the prescribed order, such as having the `price` element precede the `name` element.

Other features also contribute to the popularity of XML as a method for data interchange. For one thing, it is written in a text format, which is readable by both human beings and text-editing software. Applications can parse and process XML documents, and human beings can also read them in case there is an error in processing. Another feature is that because an XML document does not include formatting instructions, it can be displayed in various ways. Keeping data separate from formatting instructions means that the same data can be published to different media.

XML enables document portability, but it cannot do the job in a vacuum; that is, parties who use XML must agree to certain conditions. For example, in addition to agreeing to use XML for communicating, two applications must agree on the set of elements they will use and what those elements mean. For them to use Web services, they must also agree on which Web services methods they will use, what those methods do, and the order in which they are invoked when more than one method is needed.

Enterprises have several technologies available to help satisfy these requirements. They can use DTDs and XML schemas to describe the valid terms and XML documents they will use in communicating with each other. Registries provide a means for describing Web services and their methods. For higher level concepts, enterprises can use partner agreements and workflow charts and choreographies. There will be more about schemas and registries later in this document.

Overview of the Java APIs for XML

The Java APIs for XML let you write your Web applications entirely in the Java programming language. They fall into two broad categories: those that deal directly with processing XML documents and those that deal with procedures.

- Document-oriented
    - Java API for XML Processing (JAXP) — processes XML documents using various parsers
    - Java Architecture for XML Binding (JAXB) — processes XML documents using schema-derived JavaBeans™ component classes
- Procedure-oriented
    - Java API for XML-based RPC (JAX-RPC) — sends SOAP method calls to remote parties over the Internet and receives the results
    - Java API for XML Messaging (JAXM) — sends SOAP messages over the Internet in a standard way
    - Java API for XML Registries (JAXR) — provides a standard way to access business registries and share information

Perhaps the most important feature of the Java APIs for XML is that they all support industry standards, thus ensuring interoperability. Various network interoperability standards groups, such as the World Wide Web Consortium (W3C) and the Organization for the Advancement of Structured Information Standards (OASIS), have been defining standard ways of doing things so that businesses who follow these standards can make their data and applications work together.

Another feature of the Java APIs for XML is that they allow a great deal of flexibility. Users have flexibility in how they use the APIs. For example, JAXP code can use various tools for processing an XML document, and JAXM code can use various messaging protocols on top of SOAP. Implementers have flexibility as well. The Java APIs for XML define strict compatibility requirements to ensure that all implementations deliver the standard functionality, but they also give developers a great deal of freedom to provide implementations tailored to specific uses.

The following sections discuss each of these APIs, giving an overview and a feel for how to use them.

JAXP

The Java API for XML Processing (page 115) (JAXP) makes it easy to process XML data using applications written in the Java programming language. JAXP leverages the parser standards SAX (Simple API for XML Parsing) and DOM (Document Object Model) so that you can choose to parse your data as a stream of events or to build a tree-structured representation of it. The latest versions of JAXP also support the XSLT (XML Stylesheet Language Transformations) standard, giving you control over the presentation of the data and enabling you to convert the data to other XML documents or to other formats, such as HTML. JAXP also provides namespace support, allowing you to work with schemas that might otherwise have naming conflicts.

Designed to be flexible, JAXP allows you to use any XML-compliant parser from within your application. It does this with what is called a pluggability layer, which allows you to plug in an implementation of the SAX or DOM APIs. The pluggability layer also allows you to plug in an XSL processor, which lets you transform your XML data in a variety of ways, including the way it is displayed.

JAXP 1.2.2, which includes support for XML Schema, is in the Java WSDP.

The SAX API

The Simple API for XML (page 125) (SAX) defines an API for an event-based parser. Being event-based means that the parser reads an XML document from beginning to end, and each time it recognizes a syntax construction, it notifies the application that is running it. The SAX parser notifies the application by calling methods from the `ContentHandler` interface. For example, when the parser comes to a less than symbol ("<"), it calls the `startElement` method; when it comes to character data, it calls the `characters` method; when it comes to the less than symbol followed by a slash ("</"), it calls the `endElement` method, and so on. To illustrate, let's look at part of the example XML document from the first section and walk through what the parser does for each line. (For simplicity, calls to the method `ignorableWhiteSpace` are not included.)

```
<priceList>   [parser calls startElement]
  <coffee>    [parser calls startElement]
    <name>Mocha Java</name>    [parser calls startElement,
                  characters, and endElement]
    <price>11.95</price>    [parser calls startElement,
                  characters, and endElement]
  </coffee>    [parser calls endElement]
```

The default implementations of the methods that the parser calls do nothing, so you need to write a subclass implementing the appropriate methods to get the functionality you want. For example, suppose you want to get the price per pound for Mocha Java. You would write a class extending `DefaultHandler` (the default implementation of `ContentHandler`) in which you write your own implementations of the methods `startElement` and `characters`.

You first need to create a `SAXParser` object from a `SAXParserFactory` object. You would call the method `parse` on it, passing it the price list and an instance of your new handler class (with its new implementations of the methods `startElement` and `characters`). In this example, the price list is a file, but the `parse` method can also take a variety of other input sources, including an `InputStream` object, a URL, and an `InputSource` object.

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
saxParser.parse("priceList.xml", handler);
```

The result of calling the method `parse` depends, of course, on how the methods in `handler` were implemented. The SAX parser will go through the file `priceList.xml` line by line, calling the appropriate methods. In addition to the methods already mentioned, the parser will call other methods such as `startDocument`, `endDocument`, `ignorableWhiteSpace`, and `processingInstructions`, but these methods still have their default implementations and thus do nothing.

The following method definitions show one way to implement the methods `characters` and `startElement` so that they find the price for Mocha Java and print it out. Because of the way the SAX parser works, these two methods work together to look for the `name` element, the characters “Mocha Java”, and the `price` element immediately following Mocha Java. These methods use three flags to keep track of which conditions have been met. Note that the SAX parser will have to invoke both methods more than once before the conditions for printing the price are met.

```
public void startElement(..., String elementName, ...){
  if(elementName.equals("name")){
    inName = true;
  } else if(elementName.equals("price") && inMochaJava ){
```

```
            inPrice = true;
            inName = false;
        }
    }

    public void characters(char [] buf, int offset, int len) {
        String s = new String(buf, offset, len);
        if (inName && s.equals("Mocha Java")) {
            inMochaJava = true;
            inName = false;
        } else if (inPrice) {
            System.out.println("The price of Mocha Java is: " + s);
            inMochaJava = false;
            inPrice = false;
            }
        }
    }
```

Once the parser has come to the Mocha Java coffee element, here is the relevant state after the following method calls:

next invocation of `startElement` -- `inName` is `true`

next invocation of `characters` -- `inMochaJava` is `true`

next invocation of `startElement` -- `inPrice` is `true`

next invocation of `characters` -- prints price

The SAX parser can perform validation while it is parsing XML data, which means that it checks that the data follows the rules specified in the XML document's schema. A SAX parser will be validating if it is created by a `SAXParserFactory` object that has had validation turned on. This is done for the `SAXParserFactory` object `factory` in the following line of code.

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

So that the parser knows which schema to use for validation, the XML document must refer to the schema in its `DOCTYPE` declaration. The schema for the price list is `priceList.DTD`, so the `DOCTYPE` declaration should be similar to this:

```
    <!DOCTYPE PriceList SYSTEM "priceList.DTD">
```

The DOM API

The Document Object Model (page 211) (DOM), defined by the W3C DOM Working Group, is a set of interfaces for building an object representation, in the form of a tree, of a parsed XML document. Once you build the DOM, you can manipulate it with DOM methods such as `insert` and `remove`, just as you would manipulate any other tree data structure. Thus, unlike a SAX parser, a DOM parser allows random access to particular pieces of data in an XML document. Another difference is that with a SAX parser, you can only read an XML document, but with a DOM parser, you can build an object representation of the document and manipulate it in memory, adding a new element or deleting an existing one.

In the previous example, we used a SAX parser to look for just one piece of data in a document. Using a DOM parser would have required having the whole document object model in memory, which is generally less efficient for searches involving just a few items, especially if the document is large. In the next example, we add a new coffee to the price list using a DOM parser. We cannot use a SAX parser for modifying the price list because it only reads data.

Let's suppose that you want to add Kona coffee to the price list. You would read the XML price list file into a DOM and then insert the new coffee element, with its name and price. The following code fragment creates a `DocumentBuilderFactory` object, which is then used to create the `DocumentBuilder` object `builder`. The code then calls the `parse` method on `builder`, passing it the file `priceList.xml`.

```
DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.parse("priceList.xml");
```

At this point, `document` is a DOM representation of the price list sitting in memory. The following code fragment adds a new coffee (with the name "Kona" and a price of "13.50") to the price list document. Because we want to add the new coffee right before the coffee whose name is "Mocha Java", the first step is to get a list of the coffee elements and iterate through the list to find "Mocha Java". Using the `Node` interface included in the `org.w3c.dom` package, the code then creates a `Node` object for the new coffee element and also new nodes for the name and price elements. The name and price elements contain character data, so the code creates a Text object for each of them and appends the text nodes to the nodes representing the name and price elements.

```
Node rootNode = document.getDocumentElement();
NodeList list = document.getElementsByTagName("coffee");

// Loop through the list.
for (int i=0; i < list.getLength(); i++) {
  thisCoffeeNode = list.item(i);
  Node thisNameNode = thisCoffeeNode.getFirstChild();
  if (thisNameNode == null) continue;
  if (thisNameNode.getFirstChild() == null) continue;
  if (! thisNameNode.getFirstChild() instanceof
                     org.w3c.dom.Text) continue;

  String data = thisNameNode.getFirstChild().getNodeValue();
  if (! data.equals("Mocha Java")) continue;

  //We’re at the Mocha Java node. Create and insert the new
  //element.

  Node newCoffeeNode = document.createElement("coffee");

  Node newNameNode = document.createElement("name");
  Text tnNode = document.createTextNode("Kona");
  newNameNode.appendChild(tnNode);

  Node newPriceNode = document.createElement("price");
  Text tpNode = document.createTextNode("13.50");
  newPriceNode.appendChild(tpNode);

  newCoffeeNode.appendChild(newNameNode);
  newCoffeeNode.appendChild(newPriceNode);
  rootNode.insertBefore(newCoffeeNode, thisCoffeeNode);
  break;
}
```

Note that this code fragment is a simplification in that it assumes that none of the nodes it accesses will be a comment, an attribute, or ignorable white space. For information on using DOM to parse more robustly, see Increasing the Complexity (page 215).

You get a DOM parser that is validating the same way you get a SAX parser that is validating: You call setValidating(true) on a DOM parser factory before using it to create your DOM parser, and you make sure that the XML document being parsed refers to its schema in the DOCTYPE declaration.

XML Namespaces

All the names in a schema, which includes those in a DTD, are unique, thus avoiding ambiguity. However, if a particular XML document references multiple schemas, there is a possibility that two or more of them contain the same name. Therefore, the document needs to specify a namespace for each schema so that the parser knows which definition to use when it is parsing an instance of a particular schema.

There is a standard notation for declaring an XML Namespace, which is usually done in the root element of an XML document. In the following namespace declaration, the notation `xmlns` identifies `nsName` as a namespace, and `nsName` is set to the URL of the actual namespace:

```
<priceList xmlns:nsName="myDTD.dtd"
        xmlns:otherNsName="myOtherDTD.dtd">
...
</priceList>
```

Within the document, you can specify which namespace an element belongs to as follows:

```
<nsName:price> ...
```

To make your SAX or DOM parser able to recognize namespaces, you call the method `setNamespaceAware(true)` on your `ParserFactory` instance. After this method call, any parser that the parser factory creates will be namespace aware.

The XSLT API

XML Stylesheet Language for Transformations (page 289) (XSLT), defined by the W3C XSL Working Group, describes a language for transforming XML documents into other XML documents or into other formats. To perform the transformation, you usually need to supply a style sheet, which is written in the XML Stylesheet Language (XSL). The XSL style sheet specifies how the XML data will be displayed, and XSLT uses the formatting instructions in the style sheet to perform the transformation.

JAXP supports XSLT with the `javax.xml.transform` package, which allows you to plug in an XSLT transformer to perform transformations. The subpackages have SAX-, DOM-, and stream-specific APIs that allow you to perform transformations directly from DOM trees and SAX events. The following two examples illustrate how to create an XML document from a DOM tree and how to transform the resulting XML document into HTML using an XSL style sheet.

Transforming a DOM Tree to an XML Document

To transform the DOM tree created in the previous section to an XML document, the following code fragment first creates a `Transformer` object that will perform the transformation.

```
TransformerFactory transFactory =
          TransformerFactory.newInstance();
Transformer transformer = transFactory.newTransformer();
```

Using the DOM tree root node, the following line of code constructs a `DOMSource` object as the source of the transformation.

```
DOMSource source = new DOMSource(document);
```

The following code fragment creates a `StreamResult` object to take the results of the transformation and transforms the tree into an XML file.

```
File newXML = new File("newXML.xml");
FileOutputStream os = new FileOutputStream(newXML);
StreamResult result = new StreamResult(os);
transformer.transform(source, result);
```

Transforming an XML Document to an HTML Document

You can also use XSLT to convert the new XML document, `newXML.xml`, to HTML using a style sheet. When writing a style sheet, you use XML Namespaces to reference the XSL constructs. For example, each style sheet has a root element identifying the style sheet language, as shown in the following line of code.

```
<xsl:stylesheet version="1.0" xmlns:xsl=
            "http://www.w3.org/1999/XSL/Transform">
```

When referring to a particular construct in the style sheet language, you use the namespace prefix followed by a colon and the particular construct to apply. For example, the following piece of style sheet indicates that the name data must be inserted into a row of an HTML table.

```
<xsl:template match="name">
  <tr><td>
    <xsl:apply-templates/>
  </td></tr>
</xsl:template>
```

The following style sheet specifies that the XML data is converted to HTML and that the coffee entries are inserted into a row in a table.

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="priceList">
    <html><head>Coffee Prices</head>
      <body>
        <table>
          <xsl:apply-templates />
        </table>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="name">
    <tr><td>
      <xsl:apply-templates />
    </td></tr>
  </xsl:template>
  <xsl:template match="price">
    <tr><td>
      <xsl:apply-templates />
    </td></tr>
  </xsl:template>
</xsl:stylesheet>
```

To perform the transformation, you need to obtain an XSLT transformer and use it to apply the style sheet to the XML data. The following code fragment obtains a transformer by instantiating a `TransformerFactory` object, reading in the style sheet and XML files, creating a file for the HTML output, and then finally obtaining the `Transformer` object `transformer` from the `TransformerFactory` object `tFactory`.

```
TransformerFactory tFactory =
            TransformerFactory.newInstance();
String stylesheet = "prices.xsl";
String sourceId = "newXML.xml";
```

```
File pricesHTML = new File("pricesHTML.html");
FileOutputStream os = new FileOutputStream(pricesHTML);
Transformer transformer =
  tFactory.newTransformer(new StreamSource(stylesheet));
```

The transformation is accomplished by invoking the `transform` method, passing it the data and the output stream.

```
transformer.transform(
    new StreamSource(sourceId), new StreamResult(os));
```

JAXB

The Java Architecture for XML Binding (JAXB) is a Java technology that enables you to generate Java classes from XML schemas. As part of this process, the JAXB technology also provides methods for *unmarshalling* an XML instance document into a content tree of Java objects, and then *marshalling* the content tree back into an XML document. JAXB provides a fast and convenient way to bind an XML schemas to a representation in Java code, making it easy for Java developers to incorporate XML data and processing functions in Java applications without having to know much about XML itself.

One benefit of the JAXB technology is that it hides the details and gets rid of the extraneous relationships in SAX and DOM—generated JAXB classes describe only the relationships actually defined in the source schemas. The result is highly portable XML data joined with highly portable Java code that can be used to create flexible, lightweight applications and Web services.

See Chapter 9 for a description of the JAXB architecture, functions, and core concepts and then see Chapter 10, which provides sample code and step-by-step procedures for using the JAXB technology.

JAXB Binding Process

Figure 1–1 shows the JAXB data binding process.

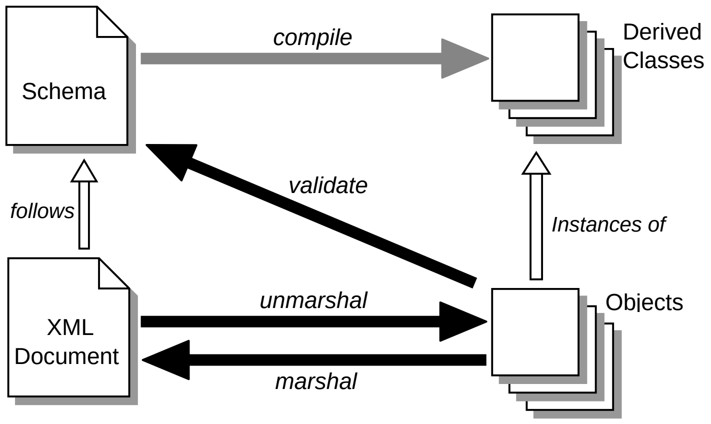


Figure 1–1 Data Binding Process

The JAXB data binding process involves the following steps:

1. Generate classes from a source XML schema, and then compile the generated classes.
2. Unmarshal XML documents conforming to the schema. Unmarshalling generates a content tree of data objects instantiated from the schema-derived JAXB classes; this content tree represents the structure and content of the source XML documents.
3. Unmarshalling optionally involves validation of the source XML documents before generating the content tree. If your application modifies the content tree, you can also use the validate operation to validate the changes before marshalling the content back to an XML document.
4. The client application can modify the XML data represented by a content tree by means of interfaces generated by the binding compiler.
5. The processed content tree is marshalled out to one or more XML output documents.

Validation

There are two types of validation that a JAXB client can perform:

- **Unmarshal-Time** – Enables a client application to receive information about validation errors and warnings detected while unmarshalling XML data into a content tree, and is completely orthogonal to the other types of validation.
- **On-Demand** – Enables a client application to receive information about validation errors and warnings detected in the content tree. At any point, client applications can call the `Validator.validate` method on the content tree (or any sub-tree of it).

Representing XML Content

Representing XML content as Java objects involves two kinds of mappings: binding XML names to Java identifiers, and representing XML schemas as sets of Java classes.

XML schema languages use XML names to label schema components, however this set of strings is much larger than the set of valid Java class, method, and constant identifiers. To resolve this discrepancy, the JAXB technology uses several name-mapping algorithms. Specifically, the name-mapping algorithm maps XML names to Java identifiers in a way that adheres to standard Java API design guidelines, generates identifiers that retain obvious connections to the corresponding schema, and is unlikely to result in many collisions.

Customizing JAXB Bindings

The default JAXB bindings can be overridden at a global scope or on a case-by-case basis as needed by using custom binding declarations. JAXB uses default binding rules that can be customized by means of binding declarations that can either be inlined or external to an XML Schema. Custom JAXB binding declarations also allow you to customize your generated JAXB classes beyond the XML-specific constraints in an XML schema to include Java specific refinements such as class and package name mappings.

Example

The following table illustrates some default XML Schema-to-JAXB bindings.

Table 1–1 Schema to JAXB Bindings

XML Schema	Java Class Files
`<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">`	
`<xsd:element name="purchaseOrder" type="PurchaseOrderType"/>`	`PurchaseOrder.java`
`<xsd:element name="comment" type="xsd:string"/>`	`Comment.java`
`<xsd:complexType name="PurchaseOrderType"> <xsd:sequence> <xsd:element name="shipTo" type="USAd- dress"/> <xsd:element name="billTo" type="USAd- dress"/> <xsd:element ref="comment" minOccurs="0"/> </xsd:sequence> <xsd:attribute name="orderDate" type="xsd:date"/> </xsd:complexType>`	`PurchaseOrder- Type.java`
`<xsd:complexType name="USAddress"> <xsd:sequence> <xsd:element name="name" type="xsd:string"/> <xsd:element name="street" type="xsd:string"/> <xsd:element name="city" type="xsd:string"/> <xsd:element name="state" type="xsd:string"/> <xsd:element name="zip" type="xsd:decimal"/> </xsd:sequence> <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/> </xsd:complexType>`	`USAddress.java`
`</xsd:schema>`	

Schema-derived Class for USAddress.java

Only a portion of the schema-derived code is shown, for brevity. The following code shows the schema-derived class for the schema's complex type `USAddress`.

```
public interface USAddress {
  String  getName();      void setName(String);
  String  getStreet();    void setStreet(String);
  String  getCity();      void setCity(String);
  String  getState();     void setState(String);
  int     getZip();       void setZip(int);
  static final String COUNTRY=”USA”;
};
```

Unmarshalling XML Content

To unmarshal XML content into a content tree of data objects, you first create a `JAXBContext` instance for handling schema-derived classes, then create an `Unmarshaller` instance, and then finally unmarshal the XML content. For example, if the generated classes are in a package named `primer.po` and the XML content is in a file named `po.xml`:

```
JAXBContext jc = JAXBContext.newInstance( "primer.po" );
Unmarshaller u = jc.createUnmarshaller();
PurchaseOrder po =
     (PurchaseOrder)u.unmarshal( new FileInputStream( "po.xml"
) );
```

To enable *unmarshal-time* validation, you create the `Unmarshaller` instance normally, as shown above, and then enable the `ValidationEventHandler`:

```
u.setValidating( true );
```

The default configuration causes the unmarshal operation to fail upon encountering the first validation error. The default validation event handler processes a validation error, generates output to `system.out`, and then throws an exception:

```
} catch( UnmarshalException ue ) {
System.out.println( "Caught UnmarshalException" );
   } catch( JAXBException je ) {
      je.printStackTrace();
   } catch( IOException ioe ) {
      ioe.printStackTrace();
```

Modifying the Content Tree

Use the schema-derived JavaBeans component set and get methods to manipulate the data in the content tree.

```
USAddress address = po.getBillTo();
address.setName( "John Bob" );
address.setStreet( "242 Main Street" );
address.setCity( "Beverly Hills" );
address.setState( "CA" );
address.setZip( 90210 );
```

Validating the Content Tree

After the application modifies the content tree, it can verify that the content tree is still valid by calling the `Validator.validate` method on the content tree (or any subtree of it). This operation is called *on-demand* validation.

```
try{
  Validator v = jc.createValidator();
  boolean valid = v.validateRoot( po );
  ...
} catch( ValidationException ue ) {
  System.out.println( "Caught ValidationException" );
  ...
}
```

Marshalling XML Content

Finally, to marshal a content tree to XML format, create a `Marshaller` instance, and then marshal the XML content:

```
Marshaller m = jc.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,Boolean.TRUE);
m.marshal( po, System.out );
```

JAX-RPC

The Java API for XML-based RPC (JAX-RPC) is the Java API for developing and using Web services. See Chapter 11 for more information about JAX-RPC and learn how to build a simple Web service and client.

Overview of JAX-RPC

An RPC-based Web service is a collection of procedures that can be called by a remote client over the Internet. For example, a typical RPC-based Web service is a stock quote service that takes a SOAP (Simple Object Access Protocol) request for the price of a specified stock and returns the price via SOAP.

Note: The SOAP 1.1 specification, available from `http://www.w3.org/`, defines a framework for the exchange of XML documents. It specifies, among other things, what is required and optional in a SOAP message and how data can be encoded and transmitted. JAX-RPC and JAXM are both based on SOAP.

A Web service, a server application that implements the procedures that are available for clients to call, is deployed on a server-side container. The container can be a servlet container such as Tomcat or a Web container in a Java 2 Platform, Enterprise Edition (J2EE) server.

A Web service can make itself available to potential clients by describing itself in a Web Services Description Language (WSDL) document. A WSDL description is an XML document that gives all the pertinent information about a Web service, including its name, the operations that can be called on it, the parameters for those operations, and the location of where to send requests. A consumer (Web client) can use the WSDL document to discover what the service offers and how to access it. How a developer can use a WSDL document in the creation of a Web service is discussed later.

Interoperability

Perhaps the most important requirement for a Web service is that it be interoperable across clients and servers. With JAX-RPC, a client written in a language other than the Java programming language can access a Web service developed and deployed on the Java platform. Conversely, a client written in the Java programming language can communicate with a service that was developed and deployed using some other platform.

What makes this interoperability possible is JAX-RPC's support for SOAP and WSDL. SOAP defines standards for XML messaging and the mapping of data types so that applications adhering to these standards can communicate with each other. JAX-RPC adheres to SOAP standards, and is, in fact, based on SOAP messaging. That is, a JAX-RPC remote procedure call is implemented as a request-response SOAP message.

The other key to interoperability is JAX-RPC's support for WSDL. A WSDL description, being an XML document that describes a Web service in a standard way, makes the description portable. WSDL documents and their uses will be discussed more later.

Ease of Use

Given the fact that JAX-RPC is based on a remote procedure call (RPC) mechanism, it is remarkably developer friendly. RPC involves a lot of complicated infrastructure, or "plumbing," but JAX-RPC mercifully makes the underlying implementation details invisible to both the client and service developer. For example, a Web services client simply makes Java method calls, and all the internal marshalling, unmarshalling, and transmission details are taken care of automatically. On the server side, the Web service simply implements the services it offers and, like the client, does not need to bother with the underlying implementation mechanisms.

Largely because of its ease of use, JAX-RPC is the main Web services API for both client and server applications. JAX-RPC focuses on point-to-point SOAP messaging, the basic mechanism that most clients of Web services use. Although it can provide asynchronous messaging and can be extended to provide higher quality support, JAX-RPC concentrates on being easy to use for the most common tasks. Thus, JAX-RPC is a good choice for applications that wish to avoid the more complex aspects of SOAP messaging and for those that find communication using the RPC model a good fit. The more heavy-duty alternative for SOAP messaging, the Java API for XML Messaging (JAXM), is discussed later in this introduction.

Advanced Features

Although JAX-RPC is based on the RPC model, it offers features that go beyond basic RPC. For one thing, it is possible to send complete documents and also document fragments. In addition, JAX-RPC supports SOAP message handlers, which make it possible to send a wide variety of messages. And JAX-RPC can be extended to do one-way messaging in addition to the request-response style of messaging normally done with RPC. Another advanced feature is extensible type mapping, which gives JAX-RPC still more flexibility in what can be sent.

Using JAX-RPC

In a typical scenario, a business might want to order parts or merchandise. It is free to locate potential sources however it wants, but a convenient way is through a business registry and repository service such as a Universal Description, Discovery and Integration (UDDI) registry. Note that the Java API for XML Registries (JAXR), which is discussed later in this introduction, offers an easy way to search for Web services in a business registry and repository. Web services generally register themselves with a business registry and store relevant documents, including their WSDL descriptions, in its repository.

After searching a business registry for potential sources, the business might get several WSDL documents, one for each of the Web services that meets its search criteria. The business client can use these WSDL documents to see what the services offer and how to contact them.

Another important use for a WSDL document is as a basis for creating stubs, the low-level classes that are needed by a client to communicate with a remote service. In the JAX-RPC implementation, the tool that uses a WSDL document to generate stubs is called `wscompile`.

The JAX-RPC implementation has another tool, called `wsdeploy`, that creates ties, the low-level classes that the server needs to communicate with a remote client. Stubs and ties, then, perform analogous functions, stubs on the client side and ties on the server side. And in addition to generating ties, `wsdeploy` can be used to create WSDL documents.

A JAX-RPC runtime system, such as the one included in the JAX-RPC implementation, uses the stubs and ties created by `wscompile` and `wsdeploy` behind the scenes. It first converts the client's remote method call into a SOAP message and sends it to the service as an HTTP request. On the server side, the JAX-RPC runtime system receives the request, translates the SOAP message into a method call, and invokes it. After the Web service has processed the request, the runtime system goes through a similar set of steps to return the result to the client. The point to remember is that as complex as the implementation details of communication between the client and server may be, they are invisible to both Web services and their clients.

Creating a Web Service

Developing a Web service using JAX-RPC is surprisingly easy. The service itself is basically two files, an interface that declares the service's remote procedures and a class that implements those procedures. There is a little more to it, in that the service needs to be configured and deployed, but first, let's take a look at the two main components of a Web service, the interface definition and its implementation class.

The following interface definition is a simple example showing the methods a wholesale coffee distributor might want to make available to its prospective customers. Note that a service definition interface extends `java.rmi.Remote` and its methods throw a `java.rmi.RemoteException` object.

```
package coffees;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface CoffeeOrderIF extends Remote {
  public Coffee [] getPriceList()
                    throws RemoteException;
  public String orderCoffee(String coffeeName, int quantity)
                    throws RemoteException;
}
```

The method `getPriceList` returns an array of `Coffee` objects, each of which contains a `name` field and a `price` field. There is one `Coffee` object for each of the coffees the distributor currently has for sale. The method `orderCoffee` returns a `String` that might confirm the order or state that it is on back order.

The following example shows what the implementation might look like (with implementation details omitted). Presumably, the method `getPriceList` will query the company's database to get the current information and return the result as an array of `Coffee` objects. The second method, `orderCoffee`, will also need to query the database to see if the particular coffee specified is available in the quantity ordered. If so, the implementation will set the internal order process in motion and send a reply informing the customer that the order will be filled. If the quantity ordered is not available, the implementation might place its own order to replenish its supply and notify the customer that the coffee is backordered.

```
package coffees;

public class CoffeeOrderImpl implements CoffeeOrderIF {
  public Coffee [] getPriceList() throws RemoteException; {
    . . .
  }

  public String orderCoffee(String coffeeName, int quantity)
                     throws RemoteException; {
    . . .
  }
}
```

After writing the service's interface and implementation class, the developer's next step is to run the mapping tool. The tool can use the interface and its implementation as a basis for generating the `stub` and `tie` classes plus other classes as necessary. And, as noted before, the developer can also use the tool to create the WSDL description for the service.

The final steps in creating a Web service are packaging and deployment. Packaging a Web service definition is done via a Web application archive (WAR). A `WAR` file is a `JAR` file for Web applications, that is, a file that contains all the files needed for the Web application in compressed form. For example, the CoffeeOrder service could be packaged in the file `jaxrpc-coffees.war`, which makes it easy to distribute and install.

One file that must be in every `WAR` file is an XML file called a *deployment descriptor*. This file, by convention named `web.xml`, contains information needed for deploying a service definition. For example, if it is being deployed on a servlet engine such as Tomcat, the deployment descriptor will include the servlet name and description, the servlet class, initialization parameters, and other startup information. One of the files referenced in a `web.xml` file is a configuration file that is automatically generated by the mapping tool. In our example, this file would be called `CoffeeOrder_Config.properties`.

Deploying our CoffeeOrder Web service example in a Tomcat container can be accomplished by simply copying the `jaxrpc-coffees.war` file to Tomcat's `webapps` directory. Deployment in a J2EE server is facilitated by using the deployment tools supplied by application server vendors.

Coding a Client

Writing the client application for a Web service entails simply writing code that invokes the desired method. Of course, much more is required to build the remote method call and transmit it to the Web service, but that is all done behind the scenes and is invisible to the client.

The following class definition is an example of a Web services client. It creates an instance of CoffeeOrderIF and uses it to call the method getPriceList. Then it accesses the price and name fields of each Coffee object in the array returned by the method getPriceList in order to print them out.

The class CoffeeOrderServiceImpl is one of the classes generated by the mapping tool. It is a stub factory whose only method is getCoffeeOrderIF; in other words, its whole purpose is to create instances of CoffeeOrderIF. The instances of CoffeeOrderIF that are created by CoffeeOrderServiceImpl are client side stubs that can be used to invoke methods defined in the interface CoffeeOrderIF. Thus, the variable coffeeOrder represents a client stub that can be used to call getPriceList, one of the methods defined in CoffeeOrderIF.

The method getPriceList will block until it has received a response and returned it. Because a WSDL document is being used, the JAX-RPC runtime will get the service endpoint from it. Thus, in this case, the client class does not need to specify the destination for the remote procedure call. When the service endpoint does need to be given, it can be supplied as an argument on the command line. Here is what a client class might look like:

```
package coffees;

public class CoffeeClient {
  public static void main(String[] args) {
    try {
      CoffeeOrderIF coffeeOrder = new
         CoffeeOrderServiceImpl().getCoffeeOrderIF();
      Coffee [] priceList =
               coffeeOrder.getPriceList():
      for (int i = 0; i < priceList.length; i++) {
        System.out.print(priceList[i].getName() + " ");
        System.out.println(priceList[i].getPrice());
      }
    } catch (Exception ex) {
    ex.printStackTrace();
    }
  }
}
```

Invoking a Remote Method

Once a client has discovered a Web service, it can invoke one of the service's methods. The following example makes the remote method call `getPriceList`, which takes no arguments. As noted previously, the JAX-RPC runtime can determine the endpoint for the CoffeeOrder service (which is its URI) from its WSDL description. If a WSDL document had not been used, you would need to supply the service's URI as a command line argument. After you have compiled the file `CoffeeClient.java`, here is all you need to type at the command line to invoke its `getPriceList` method.

```
java coffees.CoffeeClient
```

The remote procedure call made by the previous line of code is a static method call. In other words, the RPC was determined at compile time. It should be noted that with JAX-RPC, it is also possible to call a remote method dynamically at run time. This can be done using either the Dynamic Invocation Interface (DII) or a dynamic proxy.

JAXM

The Java API for XML Messaging (JAXM) provides a standard way to send XML documents over the Internet from the Java platform. It is based on the SOAP 1.1 and SOAP with Attachments specifications, which define a basic framework for exchanging XML messages. JAXM can be extended to work with higher level messaging protocols, such as the one defined in the ebXML (electronic business XML) Message Service Specification, by adding the protocol's functionality on top of SOAP.

Note: The ebXML Message Service Specification is available from `http://www.oasis-open.org/committees/ebxml-msg/`. Among other things, it provides a more secure means of sending business messages over the Internet than the SOAP specifications do.

See Chapter 12 to see how to use the JAXM API, then run the sample JAXM applications that are included with the Java WSDP.

Typically, a business uses a messaging provider service, which does the behind-the-scenes work required to transport and route messages. When a messaging provider is used, all JAXM messages go through it, so when a business sends a message, the message first goes to the sender's messaging provider, then to the recipient's messaging provider, and finally to the intended recipient. It is also possible to route a message to go to intermediate recipients before it goes to the ultimate destination.

Because messages go through it, a messaging provider can take care of housekeeping details like assigning message identifiers, storing messages, and keeping track of whether a message has been delivered before. A messaging provider can also try resending a message that did not reach its destination on the first attempt at delivery. The beauty of a messaging provider is that the client using JAXM technology ("JAXM client") is totally unaware of what the provider is doing in the background. The JAXM client simply makes Java method calls, and the messaging provider in conjunction with the messaging infrastructure makes everything happen behind the scenes.

Though in the typical scenario a business uses a messaging provider, it is also possible to do JAXM messaging without using a messaging provider. In this case, the JAXM client (called a *standalone* client) is limited to sending point-to-point messages directly to a Web service that is implemented for request-response messaging. Request-response messaging is synchronous, meaning that a request is sent and its response is received in the same operation. A request-response message is sent over a `SOAPConnection` object via the method `SOAPConnection.call`, which sends the message and blocks until it receives a response. A standalone client can operate only in a client role, that is, it can only send requests and receive their responses. In contrast, a JAXM client that uses a messaging provider may act in either the client or server (service) role. In the client role, it can send requests; in the server role, it can receive requests, process them, and send responses.

Though it is not required, JAXM messaging usually takes place within a container, such as a servlet container. A Web service that uses a messaging provider and is deployed in a container has the capability of doing one-way messaging, meaning that it can receive a request as a one-way message and can return a response some time later as another one-way message.

Because of the features that a messaging provider can supply, JAXM can sometimes be a better choice for SOAP messaging than JAX-RPC. The following list includes features that JAXM can provide and that RPC, including JAX-RPC, does not generally provide:

- One-way (asynchronous) messaging
- Routing of a message to more than one party
- Reliable messaging with features such as guaranteed delivery

A SOAPMessage object represents an XML document that is a SOAP message. A SOAPMessage object always has a required SOAP part, and it may also have one or more attachment parts. The SOAP part must always have a SOAPEnvelope object, which must in turn always contain a SOAPBody object. The SOAPEnvelope object may also contain a SOAPHeader object, to which one or more headers can be added.

The SOAPBody object can hold XML fragments as the content of the message being sent. If you want to send content that is not in XML format or that is an entire XML document, your message will need to contain an attachment part in addition to the SOAP part. There is no limitation on the content in the attachment part, so it can include images or any other kind of content, including XML fragments and documents.

Getting a Connection

The first thing a JAXM client needs to do is get a connection, either a SOAPConnection object or a ProviderConnection object.

Getting a Point-to-Point Connection

A standalone client is limited to using a SOAPConnection object, which is a point-to-point connection that goes directly from the sender to the recipient. All JAXM connections are created by a connection factory. In the case of a SOAPConnection object, the factory is a SOAPConnectionFactory object. A client obtains the default implementation for SOAPConnectionFactory by calling the following line of code.

```
SOAPConnectionFactory factory =
          SOAPConnectionFactory.newInstance();
```

The client can use factory to create a SOAPConnection object.

```
SOAPConnection con = factory.createConnection();
```

Getting a Connection to the Messaging Provider

In order to use a messaging provider, an application must obtain a ProviderConnection object, which is a connection to the messaging provider rather than to a specified recipient. There are two ways to get a ProviderConnection object, the first being similar to the way a standalone client gets a SOAPConnection object. This way involves obtaining an instance of the default implementation for ProviderConnectionFactory, which is then used to create the connection.

```
ProviderConnectionFactory pcFactory =
        ProviderConnectionFactory.newInstance();
ProviderConnection pcCon = pcFactory.createConnection();
```

The variable pcCon represents a connection to the default implementation of a JAXM messaging provider.

The second way to create a ProviderConnection object is to retrieve a ProviderConnectionFactory object that is implemented to create connections to a specific messaging provider. The following code demonstrates getting such a ProviderConnectionFactory object and using it to create a connection. The first two lines use the Java Naming and Directory Interface™ (JNDI) API to retrieve the appropriate ProviderConnectionFactory object from the naming service where it has been registered with the name "CoffeeBreakProvider". When this logical name is passed as an argument, the method lookup returns the ProviderConnectionFactory object to which the logical name was bound. The value returned is a Java Object, which must be narrowed to a ProviderConnectionFactory object so that it can be used to create a connection. The third line uses a JAXM method to actually get the connection.

```
Context ctx = getInitialContext();
ProviderConnectionFactory pcFactory =
(ProviderConnectionFactory)ctx.lookup("CoffeeBreakProvider");

ProviderConnection con = pcFactory.createConnection();
```

The ProviderConnection instance con represents a connection to The Coffee Break's messaging provider.

Creating a Message

As is true with connections, messages are created by a factory. And similar to the case with connection factories, MessageFactory objects can be obtained in two ways. The first way is to get an instance of the default implementation for the

`MessageFactory` class. This instance can then be used to create a basic `SOAPMessage` object.

```
MessageFactory messageFactory = MessageFactory.newInstance();
SOAPMessage m = messageFactory.createMessage();
```

All of the `SOAPMessage` objects that `messageFactory` creates, including `m` in the previous line of code, will be basic SOAP messages. This means that they will have no pre-defined headers.

Part of the flexibility of the JAXM API is that it allows a specific usage of a SOAP header. For example, protocols such as ebXML can be built on top of SOAP messaging to provide the implementation of additional headers, thus enabling additional functionality. This usage of SOAP by a given standards group or industry is called a *profile*. (See the JAXM tutorial section Profiles, page 492 for more information on profiles.)

In the second way to create a `MessageFactory` object, you use the `ProviderConnection` method `createMessageFactory` and give it a profile. The `SOAPMessage` objects produced by the resulting `MessageFactory` object will support the specified profile. For example, in the following code fragment, in which `schemaURI` is the URI of the schema for the desired profile, `m2` will support the messaging profile that is supplied to `createMessageFactory`.

```
MessageFactory messageFactory2 =
            con.createMessageFactory(<schemaURI>);
SOAPMessage m2 = messageFactory2.createMessage();
```

Each of the new `SOAPMessage` objects `m` and `m2` automatically contains the required elements `SOAPPart`, `SOAPEnvelope`, and `SOAPBody`, plus the optional element `SOAPHeader` (which is included for convenience). The `SOAPHeader` and `SOAPBody` objects are initially empty, and the following sections will illustrate some of the typical ways to add content.

Populating a Message

Content can be added to the `SOAPPart` object, to one or more `AttachmentPart` objects, or to both parts of a message.

Populating the SOAP Part of a Message

As stated earlier, all messages have a SOAPPart object, which has a SOAPEnvelope object containing a SOAPHeader object and a SOAPBody object. One way to add content to the SOAP part of a message is to create a SOAPHeaderElement object or a SOAPBodyElement object and add an XML fragment that you build with the method SOAPElement.addTextNode. The first three lines of the following code fragment access the SOAPBody object body, which is used to create a new SOAPBodyElement object and add it to body. The argument passed to the createName method is a Name object identifying the SOAPBodyElement being added. The last line adds the XML string passed to the method addTextNode.

```
SOAPPart sp = m.getSOAPPart();
SOAPEnvelope envelope = sp.getSOAPEnvelope();
SOAPBody body = envelope.getSOAPBody();
SOAPBodyElement bodyElement = body.addBodyElement(
          envelope.createName("text", "hotitems",
          "http://hotitems.com/products/gizmo");
bodyElement.addTextNode("some-xml-text");
```

Another way is to add content to the SOAPPart object by passing it a javax.xml.transform.Source object, which may be a SAXSource, DOMSource, or StreamSource object. The Source object contains content for the SOAP part of the message and also the information needed for it to act as source input. A StreamSource object will contain the content as an XML document; the SAXSource or DOMSource object will contain content and instructions for transforming it into an XML document.

The following code fragments illustrates adding content as a DOMSource object. The first step is to get the SOAPPart object from the SOAPMessage object. Next the code uses methods from the JAXP API to build the XML document to be added. It uses a DocumentBuilderFactory object to get a DocumentBuilder object. Then it parses the given file to produce the document that will be used to initialize a new `DOMSource` object. Finally, the code passes the `DOMSource` object `domSource` to the method `SOAPPart.setContent`.

```
SOAPPart soapPart = message.getSOAPPart();

DocumentBuilderFactory dbf=
       DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse("file:///foo.bar/soap.xml");
DOMSource domSource = new DOMSource(doc);

soapPart.setContent(domSource);
```

Populating the Attachment Part of a Message

A `Message` object may have no attachment parts, but if it is to contain anything that is not in XML format, that content must be contained in an attachment part. There may be any number of attachment parts, and they may contain anything from plain text to image files. In the following code fragment, the content is an image in a JPEG file, whose URL is used to initialize the `javax.activation.DataHandler` object `dh`. The `Message` object `m` creates the `AttachmentPart` object `attachPart`, which is initialized with the data handler containing the URL for the image. Finally, the message adds `attachPart` to itself.

```
URL url = new URL("http://foo.bar/img.jpg");
DataHandler dh = new DataHandler(url);
AttachmentPart attachPart = m.createAttachmentPart(dh);
m.addAttachmentPart(attachPart);
```

A `SOAPMessage` object can also give content to an `AttachmentPart` object by passing an `Object` and its content type to the method `createAttachmentPart`.

```
AttachmentPart attachPart =
  m.createAttachmentPart("content-string", "text/plain");
m.addAttachmentPart(attachPart);
```

A third alternative is to create an empty `AttachmentPart` object and then to pass the `AttachmentPart.setContent` method an `Object` and its content type. In this code fragment, the Object is a ByteArrayInputStream initialized with a jpeg image.

```
AttachmentPart ap = m.createAttachmentPart();
byte[] jpegData =  ...;
ap.setContent(new ByteArrayInputStream(jpegData),
                    "image/jpeg");
m.addAttachmentPart(ap);
```

Sending a Message

Once you have populated a SOAPMessage object, you are ready to send it. A standalone client uses the SOAPConnection method call to send a message. This method sends the message and then blocks until it gets back a response. The arguments to the method call are the message being sent and a URL object that contains the URL specifying the endpoint of the receiver. .

```
SOAPMessage response =
          soapConnection.call(message, endpoint);
```

An application that is using a messaging provider uses the ProviderConnection method send to send a message. This method sends the message asynchronously, meaning that it sends the message and returns immediately. The response, if any, will be sent as a separate operation at a later time. Note that this method takes only one parameter, the message being sent. The messaging provider will use header information to determine the destination.

```
providerConnection.send(message);
```

JAXR

The Java API for XML Registries (JAXR) provides a convenient way to access standard business registries over the Internet. Business registries are often described as electronic yellow pages because they contain listings of businesses and the products or services the businesses offer. JAXR gives developers writing applications in the Java programming language a uniform way to use business registries that are based on open standards (such as ebXML) or industry consortium-led specifications (such as UDDI).

Businesses can register themselves with a registry or discover other businesses with which they might want to do business. In addition, they can submit material to be shared and search for material that others have submitted. Standards groups have developed schemas for particular kinds of XML documents, and two businesses might, for example, agree to use the schema for their industry's standard purchase order form. Because the schema is stored in a standard business registry, both parties can use JAXR to access it.

Registries are becoming an increasingly important component of Web services because they allow businesses to collaborate with each other dynamically in a loosely coupled way. Accordingly, the need for JAXR, which enables enterprises to access standard business registries from the Java programming language, is also growing.

See Chapter 13 for additional information about the JAXR technology, including instructions for implementing a JAXR client to publish an organization and its web services to a registry and to query a registry to find organizations and services. The chapter also explains how to run the examples that are provided with this tutorial.

Using JAXR

The following sections give examples of two of the typical ways a business registry is used. They are meant to give you an idea of how to use JAXR rather than to be complete or exhaustive.

Registering a Business

An organization that uses the Java platform for its electronic business would use JAXR to register itself in a standard registry. It would supply its name, a description of itself, and some classification concepts to facilitate searching for it. This is shown in the following code fragment, which first creates the `RegistryService` object `rs` and then uses it to create the `BusinessLifeCycleManager` object `lcm` and the `BusinessQueryManager` object *`bqm`*. The business, a chain of coffee houses called The Coffee Break, is represented by the `Organization` object `org`, to which The Coffee Break adds its name, a description of itself, and its classification within the North American Industry Classification System (NAICS). Then `org`, which now contains the properties and classifications for The Coffee

Break, is added to the `Collection` object `orgs`. Finally, `orgs` is saved by `lcm`, which will manage the life cycle of the `Organization` objects contained in `orgs`.

```
RegistryService rs = connection.getRegistryService();

BusinessLifeCycleManager lcm =
            rs.getBusinessLifeCycleManager();
BusinessQueryManager bqm =
            rs.getBusinessQueryManager();

Organization org = lcm.createOrganization("The Coffee Break");
org.setDescription(
  "Purveyor of only the finest coffees. Established 1895");

ClassificationScheme cScheme =
  bqm.findClassificationSchemeByName("ntis-gov:naics");

Classification classification =
  (Classification)lcm.createClassification(cScheme,
  "Snack and Nonalcoholic Beverage Bars", "722213");

Collection classifications = new ArrayList();
classifications.add(classification);

org.addClassifications(classifications);
Collection orgs = new ArrayList();
orgs.add(org);
lcm.saveOrganizations(orgs);
```

Searching a Registry

A business can also use JAXR to search a registry for other businesses. The following code fragment uses the `BusinessQueryManager` object `bqm` to search for The Coffee Break. Before `bqm` can invoke the method `findOrganizations`, the code needs to define the search criteria to be used. In this case, three of the possible six search parameters are supplied to `findOrganizations`; because `null` is supplied for the third, fifth, and sixth parameters, those criteria are not used to limit the search. The first, second, and fourth arguments are all `Collection` objects, with `findQualifiers` and `namePatterns` being defined here. The only element in `findQualifiers` is a `String` specifying that no organization be returned unless its name is a case-sensitive match to one of the names in the `namePatterns` parameter. This parameter, which is also a `Collection` object with only one element, says that businesses with “Coffee” in their names are a match. The other `Collection` object is `classifications`, which was defined when The Coffee Break registered itself. The previous code fragment, in which the industry for The Coffee Break was provided, is an example of defining classifications.

```
BusinessQueryManager bqm = rs.getBusinessQueryManager();

//Define find qualifiers
Collection findQualifiers = new ArrayList();
findQualifiers.add(FindQualifier.CASE_SENSITIVE_MATCH);
Collection namePatterns = new ArrayList();
namePatterns.add("%Coffee%"); // Find orgs with name containing
//'Coffee'

//Find using only the name and the classifications
BulkResponse response = bqm.findOrganizations(findQualifiers,
     namePatterns, null, classifications, null, null);
Collection orgs = response.getCollection();
```

JAXR also supports using an SQL query to search a registry. This is done using a `DeclarativeQueryManager` object, as the following code fragment demonstrates.

```
DeclarativeQueryManager dqm = rs.getDeclarativeQueryManager();
Query query = dqm.createQuery(Query.QUERY_TYPE_SQL,
"SELECT id FROM RegistryEntry WHERE name LIKE %Coffee% " +
  "AND majorVersion >= 1 AND " +
  "(majorVersion >= 2 OR minorVersion >= 3)");
BulkResponse response2 = dqm.executeQuery(query);
```

The `BulkResponse` object `response2` will contain a value for `id` (a uuid) for each entry in `RegistryEntry` that has “Coffee” in its name and that also has a version number of 1.3 or greater.

To ensure interoperable communication between a JAXR client and a registry implementation, the messaging is done using JAXM. This is done completely behind the scenes, so as a user of JAXR, you are not even aware of it.

Sample Scenario

The following scenario is an example of how the Java APIs for XML might be used and how they work together. Part of the richness of the Java APIs for XML is that in many cases they offer alternate ways of doing something and thus let you tailor your code to meet individual needs. This section will point out some instances in which an alternate API could have been used and will also give the reasons why one API or the other might be a better choice.

Scenario

Suppose that the owner of a chain of coffee houses, called The Coffee Break, wants to expand by selling coffee online. He instructs his business manager to find some new coffee suppliers, get their wholesale prices, and then arrange for orders to be placed as the need arises. The Coffee Break can analyze the prices and decide which new coffees it wants to carry and which companies it wants to buy them from.

Discovering New Distributors

The business manager assigns the task of finding potential new sources of coffee to the company's software engineer. She decides that the best way to locate new coffee suppliers is to search a Universal Description, Discovery, and Integration (UDDI) registry, where The Coffee Break has already registered itself.

The engineer uses JAXR to send a query searching for wholesale coffee suppliers. The JAXR implementation uses JAXM behind the scenes to send the query to the registry, but this is totally transparent to the engineer.

The UDDI registry will receive the query and apply the search criteria transmitted in the JAXR code to the information it has about the organizations registered with it. When the search is completed, the registry will send back information on how to contact the wholesale coffee distributors that met the specified criteria. Although the registry uses JAXM behind the scenes to transmit the information, the response the engineer gets back is JAXR code.

Requesting Price Lists

The engineer's next step is to request price lists from each of the coffee distributors. She has obtained a WSDL description for each one, which tells her the procedure to call to get prices and also the URI where the request is to be sent. Her code makes the appropriate remote procedure calls using JAX-RPC API and gets back the responses from the distributors. The Coffee Break has been doing business with one distributor for a long time and has made arrangements with it to exchange JAXM messages using agreed-upon XML schemas. Therefore, for this distributor, the engineer's code uses JAXM API to request current prices, and the distributor returns the price list in a JAXM message.

Comparing Prices and Ordering Coffees

Upon receiving the response to her request for prices, the engineer processes the price lists using SAX. She uses SAX rather than DOM because for simply comparing prices, it is more efficient. (To modify the price list, she would have needed to use DOM.) After her application gets the prices quoted by the different vendors, it compares them and displays the results.

When the owner and business manager decide which suppliers to do business with, based on the engineer's price comparisons, they are ready to send orders to the suppliers. The orders to new distributors are sent via JAX-RPC; orders to the established distributor are sent via JAXM. Each supplier, whether using JAX-RPC or JAXM, will respond by sending a confirmation with the order number and shipping date.

Selling Coffees on the Internet

Meanwhile, The Coffee Break has been preparing for its expanded coffee line. It will need to publish a price list/order form in HTML for its Web site. But before that can be done, the company needs to determine what prices it will charge. The engineer writes an application that will multiply each wholesale price by 135% to arrive at the price that The Coffee Break will charge. With a few modifications, the list of retail prices will become the online order form.

The engineer uses JavaServer Pages™ (JSP™) technology to create an HTML order form that customers can use to order coffee online. From the JSP page, she gets the name and price of each coffee, and then she inserts them into an HTML table on the JSP page. The customer enters the quantity of each coffee desired and clicks the "Submit" button to send the order.

Conclusion

Although this scenario is simplified for the sake of brevity, it illustrates how XML technologies can be used in the world of Web services. With the availability of the Java APIs for XML and the J2EE platform, creating Web services and writing applications that use them have both gotten easier.

Chapter 19 demonstrates a simple implementation of this scenario.

2

Understanding XML

Eric Armstrong

THIS chapter describes the Extensible Markup Language (XML) and its related specifications.

Introduction to XML

This section covers the basics of XML. The goal is to give you just enough information to get started, so you understand what XML is all about. (You'll learn about XML in later sections of the tutorial.) We then outline the major features that make XML great for information storage and interchange, and give you a general idea of how XML can be used.

What Is XML?

XML is a text-based markup language that is fast becoming the standard for data interchange on the Web. As with HTML, you identify data using tags (identifiers enclosed in angle brackets, like this: `<...>`). Collectively, the tags are known as "markup".

But unlike HTML, XML tags *identify* the data, rather than specifying how to display it. Where an HTML tag says something like "display this data in bold font" (`<b>...</b>`), an XML tag acts like a field name in your program. It puts a label on a piece of data that identifies it (for example: `<message>...</message>`).

Note: Since identifying the data gives you some sense of what *means* (how to interpret it, what you should do with it), XML is sometimes described as a mechanism for specifying the *semantics* (meaning) of the data.

In the same way that you define the field names for a data structure, you are free to use any XML tags that make sense for a given application. Naturally, though, for multiple applications to use the same XML data, they have to agree on the tag names they intend to use.

Here is an example of some XML data you might use for a messaging application:

```
<message>
  <to>you@yourAddress.com</to>
  <from>me@myAddress.com</from>
  <subject>XML Is Really Cool</subject>
  <text>
    How many ways is XML cool? Let me count the ways...
  </text>
</message>
```

Note: Throughout this tutorial, we use boldface text to highlight things we want to bring to your attention. XML does not require anything to be in bold!

The tags in this example identify the message as a whole, the destination and sender addresses, the subject, and the text of the message. As in HTML, the `<to>` tag has a matching end tag: `</to>`. The data between the tag and its matching end tag defines an element of the XML data. Note, too, that the content of the `<to>` tag is entirely contained within the scope of the `<message>..</message>` tag. It is this ability for one tag to contain others that gives XML its ability to represent hierarchical data structures.

Once again, as with HTML, whitespace is essentially irrelevant, so you can format the data for readability and yet still process it easily with a program. Unlike HTML, however, in XML you could easily search a data set for messages containing "cool" in the subject, because the XML tags identify the content of the data, rather than specifying its representation.

Tags and Attributes

Tags can also contain attributes—additional information included as part of the tag itself, within the tag's angle brackets. The following example shows an email message structure that uses attributes for the `"to"`, `"from"`, and `"subject"` fields:

```
<message to="you@yourAddress.com" from="me@myAddress.com"
    subject="XML Is Really Cool">
  <text>
    How many ways is XML cool? Let me count the ways...
  </text>
</message>
```

As in HTML, the attribute name is followed by an equal sign and the attribute value, and multiple attributes are separated by spaces. Unlike HTML, however, in XML commas between attributes are not ignored—if present, they generate an error.

Since you could design a data structure like `<message>` equally well using either attributes or tags, it can take a considerable amount of thought to figure out which design is best for your purposes. Designing an XML Data Structure (page 63), includes ideas to help you decide when to use attributes and when to use tags.

Empty Tags

One really big difference between XML and HTML is that an XML document is always constrained to be well formed. There are several rules that determine when a document is well-formed, but one of the most important is that every tag has a closing tag. So, in XML, the `</to>` tag is not optional. The `<to>` element is never terminated by any tag other than `</to>`.

Note: Another important aspect of a well-formed document is that all tags are completely nested. So you can have `<message>..<to>..</to>..</message>`, but never `<message>..<to>..</message>..</to>`. A complete list of requirements is contained in the list of XML Frequently Asked Questions (FAQ) at `http://www.ucc.ie/xml/#FAQ-VALIDWF`. (This FAQ is on the w3c "Recommended Reading" list at `http://www.w3.org/XML/`.)

Sometimes, though, it makes sense to have a tag that stands by itself. For example, you might want to add a "`flag`" tag that marks message as important. A tag like that doesn't enclose any content, so it's known as an "empty tag". You can create an empty tag by ending it with `/>` instead of `>`. For example, the following message contains such a tag:

```
<message to="you@yourAddress.com" from="me@myAddress.com"
    subject="XML Is Really Cool">
  <flag/>
  <text>
    How many ways is XML cool? Let me count the ways...
  </text>
</message>
```

Note: The empty tag saves you from having to code `<flag></flag>` in order to have a well-formed document. You can control which tags are allowed to be empty by creating a Document Type Definition, or DTD. We'll talk about that in a few moments. If there is no DTD, then the document can contain any kinds of tags you want, as long as the document is well-formed.

Comments in XML Files

XML comments look just like HTML comments:

```
<message to="you@yourAddress.com" from="me@myAddress.com"
    subject="XML Is Really Cool">
  <!-- This is a comment -->
  <text>
    How many ways is XML cool? Let me count the ways...
  </text>
</message>
```

The XML Prolog

To complete this journeyman's introduction to XML, note that an XML file always starts with a prolog. The minimal prolog contains a declaration that identifies the document as an XML document, like this:

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

The declaration may also contain additional information, like this:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
```

The XML declaration is essentially the same as the HTML header, `<html>`, except that it uses `<?..?>` and it may contain the following attributes:

version
: Identifies the version of the XML markup language used in the data. This attribute is not optional.

encoding
: Identifies the character set used to encode the data. "ISO-8859-1" is "Latin-1" the Western European and English language character set. (The default is compressed Unicode: UTF-8.)

standalone
: Tells whether or not this document references an external entity or an external data type specification (see below). If there are no external references, then "yes" is appropriate

The prolog can also contain definitions of entities (items that are inserted when you reference them from within the document) and specifications that tell which tags are valid in the document, both declared in a Document Type Definition (DTD) that can be defined directly within the prolog, as well as with pointers to external specification files. But those are the subject of later tutorials. For more information on these and many other aspects of XML, see the Recommended Reading list of the w3c XML page at `http://www.w3.org/XML/`.

Note: The declaration is actually optional. But it's a good idea to include it whenever you create an XML file. The declaration should have the version number, at a minimum, and ideally the encoding as well. That standard simplifies things if the XML standard is extended in the future, and if the data ever needs to be localized for different geographical regions.

Everything that comes after the XML prolog constitutes the document's *content*.

Processing Instructions

An XML file can also contain *processing instructions* that give commands or information to an application that is processing the XML data. Processing instructions have the following format:

```
<?target instructions?>
```

where the *target* is the name of the application that is expected to do the processing, and *instructions* is a string of characters that embodies the information or commands for the application to process.

Since the instructions are application specific, an XML file could have multiple processing instructions that tell different applications to do similar things, though in different ways. The XML file for a slideshow, for example, could have processing instructions that let the speaker specify a technical or executive-level version of the presentation. If multiple presentation programs were used, the program might need multiple versions of the processing instructions (although it would be nicer if such applications recognized standard instructions).

Note: The target name “xml” (in any combination of upper or lowercase letters) is reserved for XML standards. In one sense, the declaration is a processing instruction that fits that standard. (However, when you’re working with the parser later, you’ll see that the method for handling processing instructions never sees the declaration.)

Why Is XML Important?

There are a number of reasons for XML’s surging acceptance. This section lists a few of the most prominent.

Plain Text

Since XML is not a binary format, you can create and edit files with anything from a standard text editor to a visual development environment. That makes it easy to debug your programs, and makes it useful for storing small amounts of data. At the other end of the spectrum, an XML front end to a database makes it possible to efficiently store large amounts of XML data as well. So XML provides scalability for anything from small configuration files to a company-wide data repository.

Data Identification

XML tells you what kind of data you have, not how to display it. Because the markup tags identify the information and break up the data into parts, an email program can process it, a search program can look for messages sent to particular people, and an address book can extract the address information from the rest of the message. In short, because the different parts of the information have been identified, they can be used in different ways by different applications.

Stylability

When display is important, the stylesheet standard, XSL (page 55), lets you dictate how to portray the data. For example, the stylesheet for:

```
<to>you@yourAddress.com</to>
```

can say:

1. Start a new line.
2. Display "To:" in bold, followed by a space
3. Display the destination data.

Which produces:

```
To: you@yourAddress
```

Of course, you could have done the same thing in HTML, but you wouldn't be able to process the data with search programs and address-extraction programs and the like. More importantly, since XML is inherently style-free, you can use a completely different stylesheet to produce output in postscript, TEX, PDF, or some new format that hasn't even been invented yet. That flexibility amounts to what one author described as "future-proofing" your information. The XML documents you author today can be used in future document-delivery systems that haven't even been imagined yet.

Inline Reusability

One of the nicer aspects of XML documents is that they can be composed from separate entities. You can do that with HTML, but only by linking to other documents. Unlike HTML, XML entities can be included "in line" in a document. The included sections look like a normal part of the document—you can search the whole document at one time or download it in one piece. That lets you modularize your documents without resorting to links. You can single-source a section so that an edit to it is reflected everywhere the section is used, and yet a document composed from such pieces looks for all the world like a one-piece document.

Linkability

Thanks to HTML, the ability to define links between documents is now regarded as a necessity. The next section of this tutorial, XML and Related Specs: Digesting the Alphabet Soup (page 51), discusses the link-specification initiative. This initiative lets you define two-way links, multiple-target links, "expanding" links (where clicking a link causes the targeted information to appear inline), and links between two existing documents that are defined in a third.

Easily Processed

As mentioned earlier, regular and consistent notation makes it easier to build a program to process XML data. For example, in HTML a `<dt>` tag can be delimited by `</dt>`, another `<dt>`, `<dd>`, or `</dl>`. That makes for some difficult programming. But in XML, the `<dt>` tag must always have a `</dt>` terminator, or else it will be defined as a `<dt/>` tag. That restriction is a critical part of the constraints that make an XML document well-formed. (Otherwise, the XML parser won't be able to read the data.) And since XML is a vendor-neutral standard, you can choose among several XML parsers, any one of which takes the work out of processing XML data.

Hierarchical

Finally, XML documents benefit from their hierarchical structure. Hierarchical document structures are, in general, faster to access because you can drill down to the part you need, like stepping through a table of contents. They are also easier to rearrange, because each piece is delimited. In a document, for example, you could move a heading to a new location and drag everything under it along with the heading, instead of having to page down to make a selection, cut, and then paste the selection into a new location.

How Can You Use XML?

There are several basic ways to make use of XML:

- Traditional data processing, where XML encodes the data for a program to process
- Document-driven programming, where XML documents are containers that build interfaces and applications from existing components
- Archiving—the foundation for document-driven programming, where the customized version of a component is saved (archived) so it can be used later
- Binding, where the DTD or schema that defines an XML data structure is used to automatically generate a significant portion of the application that will eventually process that data

Traditional Data Processing

XML is fast becoming the data representation of choice for the Web. It's terrific when used in conjunction with network-centric programs written in the Java™ programming language that send and retrieve information. So a client/server application, for example, could transmit XML-encoded data back and forth between the client and the server.

In the future, XML is potentially the answer for data interchange in all sorts of transactions, as long as both sides agree on the markup to use. (For example, should an e-mail program expect to see tags named `<FIRST>` and `<LAST>`, or `<FIRSTNAME>` and `<LASTNAME>`) The need for common standards will generate a lot of industry-specific standardization efforts in the years ahead. In the meantime, mechanisms that let you "translate" the tags in an XML document will be important. Such mechanisms include projects like the RDF (page 60) initiative, which defines "meat tags", and the XSL (page 55) specification, which lets you translate XML tags into other XML tags.

Document-Driven Programming (DDP)

The newest approach to using XML is to construct a document that describes how an application page should look. The document, rather than simply being displayed, consists of references to user interface components and business-logic components that are "hooked together" to create an application on the fly.

Of course, it makes sense to utilize the Java platform for such components. Both JavaBeans™ components for interfaces and Enterprise JavaBeans™ components for business logic can be used to construct such applications. Although none of the efforts undertaken so far are ready for commercial use, much preliminary work has already been done.

Note: The Java programming language is also excellent for writing XML-processing tools that are as portable as XML. Several Visual XML editors have been written for the Java platform. For a listing of editors, processing tools, and other XML resources, see the "Software" section of Robin Cover's SGML/XML Web Page at `http://www.oasis-open.org/cover/`.

Binding

Once you have defined the structure of XML data using either a DTD or the one of the schema standards, a large part of the processing you need to do has already been defined. For example, if the schema says that the text data in a `<date>` element must follow one of the recognized date formats, then one aspect of the validation criteria for the data has been defined—it only remains to write the code. Although a DTD specification cannot go the same level of detail, a DTD (like a schema) provides a grammar that tells which data structures can occur, in what sequences. That specification tells you how to write the high-level code that processes the data elements.

But when the data structure (and possibly format) is fully specified, the code you need to process it can just as easily be generated automatically. That process is known as *binding*—creating classes that recognize and process different data elements by processing the specification that defines those elements. As time goes on, you should find that you are using the data specification to generate significant chunks of code, so you can focus on the programming that is unique to your application.

Archiving

The Holy Grail of programming is the construction of reusable, modular components. Ideally, you'd like to take them off the shelf, customize them, and plug them together to construct an application, with a bare minimum of additional coding and additional compilation.

The basic mechanism for saving information is called *archiving*. You archive a component by writing it to an output stream in a form that you can reuse later. You can then read it in and instantiate it using its saved parameters. (For example, if you saved a table component, its parameters might be the number of rows and columns to display.) Archived components can also be shuffled around the Web and used in a variety of ways.

When components are archived in binary form, however, there are some limitations on the kinds of changes you can make to the underlying classes if you want to retain compatibility with previously saved versions. If you could modify the archived version to reflect the change, that would solve the problem. But that's hard to do with a binary object. Such considerations have prompted a number of investigations into using XML for archiving. But if an object's state were archived in text form using XML, then anything and everything in it could be changed as easily as you can say, "search and replace".

XML's text-based format could also make it easier to transfer objects between applications written in different languages. For all of these reasons, XML-based archiving is likely to become an important force in the not-too-distant future.

Summary

XML is pretty simple, and very flexible. It has many uses yet to be discovered—we are just beginning to scratch the surface of its potential. It is the foundation for a great many standards yet to come, providing a common language that different computer systems can use to exchange data with one another. As each industry-group comes up with standards for what they want to say, computers will begin to link to each other in ways previously unimaginable.

For more information on the background and motivation of XML, see this great article in Scientific American at

```
http://www.sciam.com/1999/0599issue/0599bosak.html.
```

XML and Related Specs: Digesting the Alphabet Soup

Now that you have a basic understanding of XML, it makes sense to get a high-level overview of the various XML-related acronyms and what they mean. There is a lot of work going on around XML, so there is a lot to learn.

The current APIs for accessing XML documents either serially or in random access mode are, respectively, SAX (page 53) and DOM (page 53). The specifications for ensuring the validity of XML documents are DTD (page 54) (the original mechanism, defined as part of the XML specification) and various Schema Standards (page 56) proposals (newer mechanisms that use XML syntax to do the job of describing validation criteria).

Other future standards that are nearing completion include the XSL (page 55) standard—a mechanism for setting up translations of XML documents (for example to HTML or other XML) and for dictating how the document is rendered. The transformation part of that standard, XSLT (+XPATH) (page 55), is completed and covered in this tutorial. Another effort nearing completion is the XML Link Language specification (XML Linking, page 58), which enables links between XML documents.

Those are the major initiatives you will want to be familiar with. This section also surveys a number of other interesting proposals, including the HTML-lookalike standard, XHTML (page 59), and the meta-standard for describing the information an XML document contains, RDF (page 60). There are also standards efforts that extend XML's capabilities, such as XLink and XPointer.

Finally, there are a number of interesting standards and standards-proposals that build on XML, including Synchronized Multimedia Integration Language (SMIL, page 61), Mathematical Markup Language (MathML, page 61), Scalable Vector Graphics (SVG, page 62), and DrawML (page 62), as well as a number of eCommerce standards.

The remainder of this section gives you a more detailed description of these initiatives. To help keep things straight, it's divided into:

- Basic Standards (page 53)
- Schema Standards (page 56)
- Linking and Presentation Standards (page 58)
- Knowledge Standards (page 60)
- Standards That Build on XML (page 61)

Skim the terms once, so you know what's here, and keep a copy of this document handy so you can refer to it whenever you see one of these terms in something you're reading. Pretty soon, you'll have them all committed to memory, and you'll be at least "conversant" with XML!

Basic Standards

These are the basic standards you need to be familiar with. They come up in pretty much any discussion of XML.

SAX

Simple API for XML

This API was actually a product of collaboration on the XML-DEV mailing list, rather than a product of the W3C. It's included here because it has the same "final" characteristics as a W3C recommendation.

You can also think of this standard as the "serial access" protocol for XML. This is the fast-to-execute mechanism you would use to read and write XML data in a server, for example. This is also called an event-driven protocol, because the technique is to register your handler with a SAX parser, after which the parser invokes your callback methods whenever it sees a new XML tag (or encounters an error, or wants to tell you anything else).

For more information on the SAX protocol, see Simple API for XML (page 125).

DOM

Document Object Model

The Document Object Model protocol converts an XML document into a collection of objects in your program. You can then manipulate the object model in any way that makes sense. This mechanism is also known as the "random access" protocol, because you can visit any part of the data at any time. You can then modify the data, remove it, or insert new data. For more information on the DOM specification, see Document Object Model (page 211).

JDOM and dom4j

While the Document Object Model (DOM) provides a lot of power for document-oriented processing, it doesn't provide much in the way of object-oriented simplification. Java developers who are processing more data-oriented structures — rather than books, articles, and other full-fledged documents — frequently find that object-oriented APIs like JDOM and dom4j are easier to use and more suited to their needs.

Here are the important differences to understand when choosing between the two:

- JDOM is somewhat cleaner, smaller API. Where “coding style” is an important consideration, JDOM is a good choice.
- JDOM is a Java Community Process^{SM} (JCP^{SM}) initiative. When completed, it will be an endorsed standard.
- dom4j is a smaller, faster implementation that has been in wide use for a number of years.
- dom4j is a factory-based implementation. That makes it easier to modify for complex, special-purpose applications. At the time of this writing, JDOM does not yet use a factory to instantiate an instance of the parser (although the standard appears to be headed in that direction). So, with JDOM, you always get the original parser. (That’s fine for the majority of applications, but may not be appropriate if your application has special needs.)

For more information on JDOM, see `http://www.jdom.org/`.

For more information on dom4j, see `http://dom4j.org/`.

DTD

Document Type Definition

The DTD specification is actually part of the XML specification, rather than a separate entity. On the other hand, it is optional—you can write an XML document without it. And there are a number of Schema Standards (page 56) proposals that offer more flexible alternatives. So it is treated here as though it were a separate specification.

A DTD specifies the kinds of tags that can be included in your XML document, and the valid arrangements of those tags. You can use the DTD to make sure you don’t create an invalid XML structure. You can also use it to make sure that the XML structure you are reading (or that got sent over the net) is indeed valid.

Unfortunately, it is difficult to specify a DTD for a complex document in such a way that it prevents all invalid combinations and allows all the valid ones. So constructing a DTD is something of an art. The DTD can exist at the front of the document, as part of the prolog. It can also exist as a separate entity, or it can be split between the document prolog and one or more additional entities.

However, while the DTD mechanism was the first method defined for specifying valid document structure, it was not the last. Several newer schema specifications have been devised. You'll learn about those momentarily.

For more information, see Creating a Document Type Definition (DTD) (page 168).

Namespaces

The namespace standard lets you write an XML document that uses two or more sets of XML tags in modular fashion. Suppose for example that you created an XML-based parts list that uses XML descriptions of parts supplied by other manufacturers (online!). The "price" data supplied by the subcomponents would be amounts you want to total up, while the "price" data for the structure as a whole would be something you want to display. The namespace specification defines mechanisms for qualifying the names so as to eliminate ambiguity. That lets you write programs that use information from other sources and do the right things with it.

The latest information on namespaces can be found at `http://www.w3.org/TR/REC-xml-names`.

XSL

Extensible Stylesheet Language

The XML standard specifies how to identify data, not how to display it. HTML, on the other hand, told how things should be displayed without identifying what they were. The XSL standard has two parts, XSLT (the transformation standard, described next) and XSL-FO (the part that covers *formatting objects*, also known as *flow objects*). XSL-FO gives you the ability to define multiple areas on a page and then link them together. When a text stream is directed at the collection, it fills the first area and then "flows" into the second when the first area is filled. Such objects are used by newsletters, catalogs, and periodical publications.

The latest W3C work on XSL is at `http://www.w3.org/TR/WD-xsl`.

XSLT (+XPATH)

Extensible Stylesheet Language for Transformations

The XSLT transformation standard is essentially a translation mechanism that lets you specify what to convert an XML tag into so that it can be displayed—for example, in HTML. Different XSL formats can then be used to display the same data in different ways, for different uses. (The XPATH standard is an addressing mechanism that you use when constructing transformation instructions, in order to specify the parts of the XML structure you want to transform.)

For more information, see XML Stylesheet Language for Transformations (page 289).

Schema Standards

A DTD makes it possible to validate the structure of relatively simple XML documents, but that's as far as it goes.

A DTD can't restrict the content of elements, and it can't specify complex relationships. For example, it is impossible to specify with a DTD that a <heading> for a <book> must have both a <title> and an <author>, while a <heading> for a <chapter> only needs a <title>. In a DTD, once you only get to specify the structure of the <heading> element one time. There is no context-sensitivity.

This issue stems from the fact that a DTD specification is not hierarchical. For a mailing address that contained several "parsed character data" (PCDATA) elements, for example, the DTD might look something like this:

```
<!ELEMENT mailAddress (name, address, zipcode)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT zipcode (#PCDATA)>
```

As you can see, the specifications are linear. That fact forces you to come up with new names for similar elements in different settings. So if you wanted to add another "name" element to the DTD that contained the <firstname>, <middleInitial>, and <lastName>, then you would have to come up with another identifier. You could not simply call it "name" without conflicting with the <name> element defined for use in a <mailAddress>.

Another problem with the non hierarchical nature of DTD specifications is that it is not clear what comments are meant to explain. A comment at the top like `<!-- Address used for mailing via the postal system -->` would apply to all of the elements that constitute a mailing address. But a comment like `<!-- Addressee -->` would apply to the `name` element only. On the other hand, a comment like `<!-- A 5-digit string -->` would apply specifically to the

`#PCDATA` part of the `zipcode` element, to describe the valid formats. Finally, DTDs do not allow you to formally specify field-validation criteria, such as the 5-digit (or 5 and 4) limitation for the `zipcode` field.

Finally, a DTD uses syntax which substantially different from XML, so it can't be processed with a standard XML parser. That means you can't read a DTD into a DOM, for example, modify it, and then write it back out again.

To remedy these shortcomings, a number of proposals have been made for a more database-like, hierarchical "schema" that specifies validation criteria. The major proposals are shown below.

XML Schema

A large, complex standard that has two parts. One part specifies structure relationships. (This is the largest and most complex part.) The other part specifies mechanisms for validating the content of XML elements by specifying a (potentially very sophisticated) *datatype* for each element. The good news is that XML Schema for Structures lets you specify any kind of relationship you can conceive of. The bad news is that it takes a lot of work to implement, and it takes a bit of learning to use. Most of the alternatives provide for simpler structure definitions, while incorporating the XML Schema datatype standard.

For more information on the XML Schema, see the W3C specs XML Schema (Structures) and XML Schema (Datatypes), as well as other information accessible at http://www.w3c.org/XML/Schema.

RELAX NG

Regular Language description for XML

Simpler than XML Structure Schema, is an emerging standard under the auspices of OASIS (Organization for the Advancement of Structured Information Systems). RELAX NG use regular expression patterns to express constraints on structure relationships, and it is designed to work with the XML Schema datatyping mechanism to express content constraints. This standard also uses XML syntax, and it includes a DTD to RELAX converter. ("NG" stands for "Next Generation". It's a newer version of the RELAX schema mechanism that integrates TREX.)

For more information on RELAX NG, see http://www.oasis-open.org/committees/relax-ng/.

TREX

Tree Regular Expressions for XML

A means of expressing validation criteria by describing a *pattern* for the structure and content of an XML document. Now part of the RELAX NG specification.

For more information on TREX, see `http://www.thaiopensource.com/trex/`.

SOX

Schema for Object-oriented XML

SOX is a schema proposal that includes extensible data types, namespaces, and embedded documentation.

For more information on SOX, see `http://www.w3.org/TR/NOTE-SOX`.

Schematron

Schema for Object-oriented XML

An assertion-based schema mechanism that allows for sophisticated validation.

For more information on the Schematron validation mechanism, see `http://www.ascc.net/xml/resource/schematron/schematron.html`.

Linking and Presentation Standards

Arguably the two greatest benefits provided by HTML were the ability to link between documents, and the ability to create simple formatted documents (and, eventually, very complex formatted documents). The following standards aim at preserving the benefits of HTML in the XML arena, and to adding additional functionality, as well.

XML Linking

These specifications provide a variety of powerful linking mechanisms, and are sure to have a big impact on how XML documents are used.

XLink

The XLink protocol is a specification for handling links between XML documents. This specification allows for some pretty sophisticated linking, including two-way links, links to multiple documents, “expanding” links that insert the linked information into your document rather than replacing your document with a new page, links between two documents that are created in a third, independent document, and indirect links (so you can point to an “address book” rather than directly to the target document—updating the address book then automatically changes any links that use it).

XML Base

This standard defines an attribute for XML documents that defines a “base” address, that is used when evaluating a relative address specified in the document. (So, for example, a simple file name would be found in the base-address directory.)

XPointer

In general, the XLink specification targets a document or document-segment using its ID. The XPointer specification defines mechanisms for “addressing into the internal structures of XML documents”, without requiring the author of the document to have defined an ID for that segment. To quote the spec, it provides for “reference to elements, character strings, and other parts of XML documents, whether or not they bear an explicit ID attribute”.

For more information on the XML Linking standards, see `http://www.w3.org/XML/Linking`.

XHTML

The XHTML specification is a way of making XML documents that look and act like HTML documents. Since an XML document can contain any tags you care to define, why not define a set of tags that look like HTML? That’s the thinking behind the XHTML specification, at any rate. The result of this specification is a document that can be displayed in browsers and also treated as XML data. The data may not be quite as identifiable as “pure” XML, but it will be a heck of a lot easier to manipulate than standard HTML, because XML specifies a good deal more regularity and consistency.

For example, every tag in a well-formed XML document must either have an end-tag associated with it or it must end in `/>`. So you might see `<p>...</p>`, or you might see `<p/>`, but you will never see `<p>` standing by itself. The upshot of that requirement is that you never have to program for the weird kinds of cases you see in HTML where, for example, a `<dt>` tag might be terminated by `</DT>`, by another `<DT>`, by `<dd>`, or by `</dl>`. That makes it a lot easier to write code!

The XHTML specification is a reformulation of HTML 4.0 into XML. The latest information is at `http://www.w3.org/TR/xhtml1`.

Knowledge Standards

When you start looking down the road five or six years, and visualize how the information on the Web will begin to turn into one huge knowledge base (the "semantic Web"). For the latest on the semantic Web, visit `http://www.w3.org/2001/sw/`.

In the meantime, here are the fundamental standards you'll want to know about:

RDF

Resource Description Framework

RDF is a standard for defining *meta* data -- information that describes what a particular data item is, and specifies how it can be used. Used in conjunction with the XHTML specification, for example, or with HTML pages, RDF could be used to describe the content of the pages. For example, if your browser stored your ID information as `FIRSTNAME`, `LASTNAME`, and `EMAIL`, an RDF description could make it possible to transfer data to an application that wanted `NAME` and `EMAILADDRESS`. Just think: One day you may not need to type your name and address at every Web site you visit!

For the latest information on RDF, see `http://www.w3.org/TR/REC-rdf-syntax`.

RDF Schema

RDF Schema allows the specification of consistency rules and additional information that describe how the statements in a Resource Description Framework (RDF) should be interpreted.

For more information on the RDF Schema recommendation, see `http://www.w3.org/TR/rdf-schema`.

XTM

XML Topic Maps

In many ways a simpler, more readily usable knowledge-representation than RDF, the topic maps standard is one worth watching. So far, RDF is the W3C standard for knowledge representation, but topic maps could possibly become the “developer’s choice” among knowledge representation standards.

For more information on XML Topic Maps, `http://www.topicmaps.org/xtm/index.html`. For information on topic maps and the Web, see `http://www.topicmaps.org/`.

Standards That Build on XML

The following standards and proposals build on XML. Since XML is basically a language-definition tool, these specifications use it to define standardized languages for specialized purposes.

Extended Document Standards

These standards define mechanisms for producing extremely complex documents—books, journals, magazines, and the like—using XML.

SMIL

Synchronized Multimedia Integration Language

SMIL is a W3C recommendation that covers audio, video, and animations. It also addresses the difficult issue of synchronizing the playback of such elements.

For more information on SMIL, see `http://www.w3.org/TR/REC-smil`.

MathML

Mathematical Markup Language

MathML is a W3C recommendation that deals with the representation of mathematical formulas.

For more information on MathML, see `http://www.w3.org/TR/REC-MathML`.

SVG

Scalable Vector Graphics

SVG is a W3C working draft that covers the representation of vector graphic images. (Vector graphic images that are built from commands that say things like “draw a line (square, circle) from point xi to point m,n” rather than encoding the image as a series of bits. Such images are more easily scalable, although they typically require more processing time to render.)

For more information on SVG, see `http://www.w3.org/TR/WD-SVG`.

DrawML

Drawing Meta Language

DrawML is a W3C note that covers 2D images for technical illustrations. It also addresses the problem of updating and refining such images.

For more information on DrawML, see `http://www.w3.org/TR/NOTE-drawml`.

eCommerce Standards

These standards are aimed at using XML in the world of business-to-business (B2B) and business-to-consumer (B2C) commerce.

ICE

Information and Content Exchange

ICE is a protocol for use by content syndicators and their subscribers. It focuses on “automating content exchange and reuse, both in traditional publishing contexts and in business-to-business relationships”.

For more information on ICE, see `http://www.w3.org/TR/NOTE-ice`.

ebXML

Electronic Business with XML

This standard aims at creating a modular electronic business framework using XML. It is the product of a joint initiative by the United Nations (UN/CEFACT) and the Organization for the Advancement of Structured Information Systems (OASIS).

For more information on ebXML, see `http://www.ebxml.org/`.

cxml

Commerce XML

cxml is a RosettaNet (www.rosettanet.org) standard for setting up interactive online catalogs for different buyers, where the pricing and product offerings are company specific. Includes mechanisms to handle purchase orders, change orders, status updates, and shipping notifications.

For more information on cxml, see http://www.cxml.org/.

CBL

Common Business Library

CBL is a library of element and attribute definitions maintained by CommerceNet (www.commerce.net).

For more information on CBL and a variety of other initiatives that work together to enable eCommerce applications, see http://www.commerce.net/projects/currentprojects/eco/wg/eCo_Framework_Specifications.html.

UBL

Universal Business Library

An OASIS initiative aimed at compiling a standard library of XML business documents (purchase orders, invoices, etc.) that are defined with XML Schema definitions.

For more information on UBL, see http://www.oasis-open.org/committees/ubl.

Summary

XML is becoming a widely-adopted standard that is being used in a dizzying variety of application areas.

Designing an XML Data Structure

This section covers some heuristics you can use when making XML design decisions.

Saving Yourself Some Work

Whenever possible, use an existing schema definition. It's usually a lot easier to ignore the things you don't need than to design your own from scratch. In addition, using a standard DTD makes data interchange possible, and may make it possible to use data-aware tools developed by others.

So, if an industry standard exists, consider referencing that DTD with an external parameter entity. One place to look for industry-standard DTDs is at the repository created by the Organization for the Advancement of Structured Information Standards (OASIS) at `http://www.XML.org`. Another place to check is CommerceOne's XML Exchange at `http://www.xmlx.com`, which is described as "a repository for creating and sharing document type definitions".

Note: Many more good thoughts on the design of XML structures are at the OASIS page, `http://www.oasis-open.org/cover/elementsAndAttrs.html`.

Attributes and Elements

One of the issues you will encounter frequently when designing an XML structure is whether to model a given data item as a subelement or as an attribute of an existing element. For example, you could model the title of a slide either as:

```
<slide>
  <title>This is the title</title>
</slide>
```

or as:

```
<slide title="This is the title">...</slide>
```

In some cases, the different characteristics of attributes and elements make it easy to choose. Let's consider those cases first, and then move on to the cases where the choice is more ambiguous.

Forced Choices

Sometimes, the choice between an attribute and an element is forced on you by the nature of attributes and elements. Let's look at a few of those considerations:

The data contains substructures
In this case, the data item must be modeled as an *element*. It can't be modeled as an attribute, because attributes take only simple strings. So if the title can contain emphasized text like this: `The <em>Best</em> Choice`, then the title must be an element.

The data contains multiple lines
Here, it also makes sense to use an *element*. Attributes need to be simple, short strings or else they become unreadable, if not unusable.

Multiple occurrences are possible
Whenever an item can occur multiple times, like paragraphs in an article, it must be modeled as an *element*. The element that contains it can only have one attribute of a particular kind, but it can have many subelements of the same type.

The data changes frequently
When the data will be frequently modified with an editor, it may make sense to model it as an *element*. Many XML-aware editors make it easy modify element data, while attributes can be somewhat harder to get to.

The data is a small, simple string that rarely if ever changes
This is data that can be modeled as an *attribute*. However, just because you *can* does not mean that you should. Check the "Stylistic Choices" section next, to be sure.

Using DTDs when the data is confined to a small number of fixed choices
Here is one time when it really makes sense to use an *attribute*. A DTD can prevent an attribute from taking on any value that is not in the preapproved list, but it cannot similarly restrict an element. (With a schema on the other hand, both attributes and elements can be restricted.)

Stylistic Choices

As often as not, the choices are not as cut and dried as those shown above. When the choice is not forced, you need a sense of "style" to guide your thinking. The question to answer, then, is what makes good XML style, and why.

Defining a sense of style for XML is, unfortunately, as nebulous a business as defining "style" when it comes to art or music. There are a few ways to approach it, however. The goal of this section is to give you some useful thoughts on the subject of "XML style".

Visibility

One heuristic for thinking about XML elements and attributes uses the concept of *visibility*. If the data is intended to be shown—to be displayed to some end user—then it should be modeled as an element. On the other hand, if the information guides XML processing but is never seen by a user, then it may be better to model it as an attribute. For example, in order-entry data for shoes, shoe size would definitely be an element. On the other hand, a manufacturer's code number would be reasonably modeled as an attribute.

Consumer / Provider

Another way of thinking about the visibility heuristic is to ask who is the consumer and/or provider of the information. The shoe size is entered by a human sales clerk, so it's an element. The manufacturer's code number for a given shoe model, on the other hand, may be wired into the application or stored in a database, so that would be an attribute. (If it were entered by the clerk, though, it should perhaps be an element.)

Container vs. Contents

Perhaps the best way of thinking about elements and attributes is to think of an element as a *container*. To reason by analogy, the *contents* of the container (water or milk) correspond to XML data modeled as elements. Such data is essentially variable. On the other hand, *characteristics* of the container (blue or white pitcher) can be modeled as attributes. That kind of information tends to be more immutable. Good XML style will, in some consistent way, separate each container's contents from its characteristics.

To show these heuristics at work: In a slideshow the type of the slide (executive or technical) is best modeled as an attribute. It is a characteristic of the slide that lets it be selected or rejected for a particular audience. The title of the slide, on the other hand, is part of its contents. The visibility heuristic is also satisfied here. When the slide is displayed, the title is shown but the type of the slide isn't. Finally, in this example, the consumer of the title information is the presentation audience, while the consumer of the type information is the presentation program.

Normalizing Data

The section Designing an XML Data Structure (page 63) shows how to create an external entity that you can reference in an XML document. Such an entity has all the advantages of a modularized routine—changing that one copy affects every document that references it. The process of eliminating redundancies is known as *normalizing*, so defining entities is one good way to normalize your data.

In an HTML file, the only way to achieve that kind of modularity is with HTML links—but of course the document is then fragmented, rather than whole. XML entities, on the other hand, suffer no such fragmentation. The entity reference acts like a macro—the entity's contents are expanded in place, producing a whole document, rather than a fragmented one. And when the entity is defined in an external file, multiple documents can reference it.

The considerations for defining an entity reference, then, are pretty much the same as those you would apply to modularized program code:

- Whenever you find yourself writing the same thing more than once, think entity. That lets you write it one place and reference it multiple places.
- If the information is likely to change, especially if it is used in more than one place, definitely think in terms of defining an entity. An example is defining `productName` as an entity so that you can easily change the documents when the product name changes.
- If the entity will never be referenced anywhere except in the current file, define it in the local_subset of the document's DTD, much as you would define a method or inner class in a program.
- If the entity will be referenced from multiple documents, define it as an external entity, the same way that would define any generally usable class as an external class.

External entities produce modular XML that is smaller, easier to update and maintain. They can also make the resulting document somewhat more difficult to visualize, much as a good OO design can be easy to change, once you understand it, but harder to wrap your head around at first.

You can also go overboard with entities. At an extreme, you could make an entity reference for the word "the"—it wouldn't buy you much, but you could do it.

Note: The larger an entity is, the less likely it is that changing it will have unintended effects. When you define an external entity that covers a whole section on installation instructions, for example, making changes to the section is unlikely to make any of the documents that depend on it come out wrong. Small inline substitutions can be more problematic, though. For example, if `productName` is defined as an entity, the name change can be to a different part of speech, and that can produce! Suppose the product name is something like "HtmlEdit". That's a verb. So you write a sentence that becomes, "You can HtmlEdit your file..." after the entity-substitution occurs. That sentence reads fine, because the verb fits well in that context. But if the name is eventually changed to “HtmlEditor”, the sentence becomes “You can HtmlEditor your file...”, which clearly doesn’t work. Still, even if such simple substitutions can sometimes get you in trouble, they can potentially save a lot of time. (One alternative would be to set up entities named `productNoun`, `productVerb`, `productAdj`, and `productAdverb`!)

Normalizing DTDs

Just as you can normalize your XML document, you can also normalize your DTD declarations by factoring out common pieces and referencing them with a parameter entity. This process is described in the SAX tutorial in Defining Parameter Entities and Conditional Sections (page 193). Factoring out the DTDs (also known as modularizing or normalizing) gives the same advantages and disadvantages as normalized XML—easier to change, somewhat more difficult to follow.

You can also set up conditionalized DTDs, as described in the SAX tutorial section Conditional Sections (page 196). If the number and size of the conditional sections is small relative to the size of the DTD as a whole, that can let you “single source” a DTD that you can use for multiple purposes. If the number of conditional sections gets large, though, the result can be a complex document that is difficult to edit.

3

Getting Started With Tomcat

Debbie Carson

THIS chapter shows you how to develop, deploy, and run a simple Web application that consists of a currency conversion JavaBeans™ component and a Web page client created with JavaServer Pages™ (JSP™) technology. This application will be deployed to, and run on, Tomcat, the Java™ Servlet and JSP container developed by The Apache Software Foundation (www.apache.org), and included with the Java Web Services Developer Pack (Java WSDP). This chapter is intended as an introduction to using Tomcat to deploy Web services and Web applications. The material in this chapter provides a basis for other chapters in this tutorial.

Setting Up

Note: Before you start developing the example applications, follow the instructions in About This Tutorial (page xi), then continue with this section.

Getting the Example Code

The source code for the example is in `<`*`JWSDP_HOME`*`>/docs/tutorial/examples/gs/`, a directory that is created when you unzip the tutorial bundle. If you are viewing this tutorial online, you can download the tutorial bundle from:

```
http://java.sun.com/webservices/downloads/webservicestutorial.html
```

Layout of the Example Code

In this example application, the source code directories are organized according to the "best practices approach to Web services programming", which is described in more detail in the file `<JWSDP_HOME>/docs/tomcat/appdev/deployment.html`. Basically, the document explains that it is useful to examine the runtime organization of a Web application when creating the application. A Web application is defined as a hierarchy of directories and files in a standard layout. Such a hierarchy can be accessed in its unpacked form, where each directory and file exists in the file system separately, or in a packed form known as a Web Application Archive, or WAR file. The former format is more useful during development, while the latter is used when you distribute your application to be installed.

To facilitate creation of a WAR file in the required format, it is convenient to arrange the files that Tomcat uses when executing your application in the same organization as required by the WAR format itself. In the example application, `<JWSDP_HOME>/docs/tutorial/examples/gs/` is the root directory for the source code for this application. The application consists of the following files that are either in the `/gs` directory or a subdirectory of `/gs`.

- `/src/converterApp/ConverterBean.java` - The JavaBeans component that contains the `get` and `set` methods for the `yenAmount` and `euroAmount` properties used to convert U.S. dollars to Yen and convert Yen to Euros.
- `/web/index.jsp` - The Web client, which is a JavaServer Pages page that accepts the value to be converted, the buttons to submit the value, and the result of the conversion.
- `/web/WEB-INF/web.xml` - the deployment descriptor for this application. In this simple example, it contains a description of the example application.
- `build.xml` - The build file that uses the `Ant` tool to build and deploy the Web application.

More information about WAR files can be found in Web Application Archives (page 96).

A key recommendation of the *Tomcat Application Developer's Manual* is to separate the directory hierarchy containing the source code from the directory hierarchy containing the deployable application. Maintaining this separation has the following advantages:

- The contents of the source directories can be more easily administered, moved, and backed up if the executable version of the application is not intermixed.
- Source code control is easier to manage on directories that contain only source files.
- The files that make up an installable distribution of your application are much easier to select when the deployment hierarchy is separate.

As discussed in Creating the Build and Deploy File for Ant (page 78), the `Ant` development tool makes the creation and processing of this type of directory hierarchies relatively simple.

The rest of this document shows how this example application is created, built, deployed, and run. If you would like to skip the information on creating the example application, you can go directly to Quick Overview (page 72).

Setting the PATH Variable

It is very important that you add the `bin` directories of the Java WSDP, Java 2 Software Development Kit, Standard Edition (J2SE™ SDK), and Ant installations to the front of your `PATH` environment variable so that the Java WSDP startup scripts for Tomcat override other installations.

In addition, most of the examples are distributed with a configuration file for version 1.5.1 of `Ant`, a portable build tool contained in the Java WSDP. The version of `Ant` shipped with the Java WSDP sets the `jwsdp.home` environment variable, which is required by the example build files. To ensure that you use this version of `Ant`, you must add `<`*`JWSDP_HOME`*`>/jakarta-ant-1.5.1/bin` to the front of your `PATH`.

Creating the Build Properties File

In order to invoke many of the `Ant` tasks, you need to put a file named `build.properties` in your home directory. On the Solaris operating system, your home directory is generally of the format `/home/`*`your_login_name`*. In the

Windows operating environment (for example on Windows 2000), your home directory is generally `C:\Documents and Settings\`*`yourProfile`*.

The `build.properties` file contains a user name and password in plain text format that match the user name and password set up during installation. The user name and password that you entered during installation of the Java WSDP are stored in <*`JWSDP_HOME`*>`/conf/tomcat-users.xml`.

For security purposes, the Tomcat Manager application verifies that you (as defined in the `build.properties` file) are a user who is authorized to install and reload applications (as defined in `tomcat-users.xml`) before granting you access to the server.

If you have not already created a `build.properties` file in your home directory, do so now. The file will look like this:

```
username=your_username
password=your_password
```

Note: For security purposes, make the `build.properties` file unreadable to anyone but yourself.

The `tomcat-users.xml` file, which is created by the installer, looks like this:

```
<?xml version='1.0'?>
<tomcat-users>
<role rolename="admin"/>
<role rolename="manager"/>
<role rolename="provider"/>
<user username="your_username" password="your_password"
     roles="admin,manager,provider"/>
</tomcat-users>
```

Quick Overview

Now that you've downloaded the application and gotten your environment set up for running the example application, this section will show you a quick overview of the steps needed to run the application. Each step is discussed in more detail on the page referenced.

1. Follow the steps in Setting Up (page 69).

2. Change to the directory for this application, (`<JWSDP_HOME>/docs/tutorial/examples/gs` (see Creating the Getting Started Application (page 73)).
3. Compile the source files by typing the following at the terminal prompt (see Building the Getting Started Application Using Ant, page 77):

   ```
   ant build
   ```

   Compile errors are listed in Compilation Errors (page 88).
4. Start Tomcat by typing the following at the terminal prompt (see Starting Tomcat, page 80):

   ```
   <JWSDP_HOME>/bin/startup.sh        (Unix platform)

   <JWSDP_HOME>\bin\startup           (Microsoft Windows)
   ```
5. Deploy the Web application using `Ant` by typing the following at the terminal prompt (see Installing the Application using Ant, page 81).

   ```
   ant install
   ```

   Deployment errors are discussed in Deployment Errors (page 89).
6. Start a Web browser. Enter the following URL to run the example application (see Running the Getting Started Application, page 82):

   ```
   http://localhost:8080/GSApp
   ```
7. Shutdown Tomcat by typing the following at the terminal prompt (see Shutting Down Tomcat, page 83):

   ```
   <JWSDP_HOME>/bin/shutdown.sh        (Unix platform)

   <JWSDP_HOME>\bin\shutdown           (Microsoft Windows)
   ```

Creating the Getting Started Application

The example application contains a `ConverterBean` class, a Web component, a file to build and run the application, and a deployment descriptor. For this example, we will create a top-level *project source directory* named `gs/`. All of the files in this example application are created from this root directory.

The ConverterBean Component

The ConverterBean component used in the example application is used in conjunction with a JSP page. The resulting application is a form that enables you to convert American dollars to Yen, and convert Yen to Euros. The source code for the ConverterBean component is in the <*JWSDP_HOME*>/docs/tutorial/examples/gs/src/converterApp/ directory.

Coding the ConverterBean Component

The ConverterBean component for this example contains two properties, yenAmount and euroAmount, and the set and get methods for these properties. The source code for ConverterBean follows.

```
//ConverterBean.java
package converterApp;

import java.math.*;

public class ConverterBean{

  private BigDecimal yenRate;
  private BigDecimal euroRate;
  private BigDecimal yenAmount;
  private BigDecimal euroAmount;

  /** Creates new ConverterBean */
  public ConverterBean() {
     yenRate = new BigDecimal ("138.78");
     euroRate = new BigDecimal (".0084");
     yenAmount = new BigDecimal("0.0");
     euroAmount = new BigDecimal("0.0");
  }
  public BigDecimal getYenAmount () {
     return yenAmount;
  }
  public void setYenAmount(BigDecimal amount) {
     yenAmount = amount.multiply(yenRate);
     yenAmount =  yenAmount.setScale(2,BigDecimal.ROUND_UP);
  }
  public BigDecimal getEuroAmount () {
     return euroAmount;
  }
  public void setEuroAmount (BigDecimal amount) {
     euroAmount = amount.multiply(euroRate);
```

```
            euroAmount =
               euroAmount.setScale(2,BigDecimal.ROUND_UP);
         }
      }
```

The Web Client

The Web client is contained in the JSP page <*JWSDP_HOME*>/docs/tutorial/examples/gs/web/index.jsp. A JSP page is a text-based document that contains both static and dynamic content. The static content is the template data that can be expressed in any text-based format, such as HTML, WML, or XML. JSP elements construct the dynamic content.

Coding the Web Client

The JSP page, index.jsp, is used to create the form that will appear in the Web browser when the application client is running. This JSP page is a typical mixture of static HTML markup and JSP elements. If you have developed Web pages, you are probably familiar with the HTML document structure statements (<head>, <body>, and so on) and the HTML statements that create a form <form> and a menu <select>. The highlighted lines in the example contain the following types of JSP constructs:

- Directives (**<%@page ... %>**) import classes in the ConverterBean class, and set the content type returned by the page.
- The **jsp:useBean** element declares that the page will use a bean that is stored within and accessible from the specified scope. The default scope is page, so we do not explicitly set it in this example.
- The **jsp:setProperty** element is used to set JavaBeans component properties in a JSP page.
- The **jsp:getProperty** element is used to retrieve JavaBeans component properties in a JSP page.
- Scriptlets (**<% ... %>**) retrieve the value of the amount request parameter, convert it to a BigDecimal, and convert the value to Yen or Euro.
- Expressions (**<%= ... %>**) insert the value of the amount into the response.

The source code for `index.jsp` follows.

```
<%-- index.jsp --%>
<%@ page import="converterApp.ConverterBean,java.math.*" %>
<%@ page contentType="text/html; charset=ISO-8859-5" %>

<html>
<head>
<title>Currency Conversion Application</title>
</head>

<body bgcolor="white">
“<jsp:useBean id="converter"
class="converterApp.ConverterBean"/>

<h1><FONT FACE="ARIAL" SIZE=12>Currency Conversion Application
</FONT></h1>
<hr>
<p><FONT FACE="ARIAL" SIZE=10>Enter an amount to convert:</p>
</FONT>
<form method="get">
<input type="text" name="amount" size="25">
<br>
<p>
<input type="submit" value="Submit">
<input type="reset" value="Reset">
</form>
<%
String amount = request.getParameter("amount");

if ( amount != null && amount.length() > 0 ) {

%>
<p><FONT FACE="ARIAL" SIZE=10><%= amount %> dollars are

<jsp:setProperty name="converter" property="yenAmount"
value="<%= new BigDecimal(amount)%>" />
<jsp:getProperty name="converter" property="yenAmount"  /> Yen.

<p><%= amount %> Yen are

<jsp:setProperty name="converter" property="euroAmount"
value="<%= new BigDecimal(amount)%>" />
<jsp:getProperty name="converter" property="euroAmount"  />
Euro. </FONT>

<%
}
```

```
%>

</body>
</html>
```

Building the Getting Started Application Using Ant

Now the example Web application is ready to build.

This release of the Java Web Services Developer Pack includes `Ant`, a make tool that is portable across platforms, and which is developed by the Apache Software Foundation (`http://www.apache.org`). Documentation for the `Ant` tool can be found in the file `index.html` from the `<`*`JWSDP_HOME`*`>/jakarta-ant-1.5.1/docs/` directory of your Java WSDP installation.

The version of `Ant` shipped with the Java WSDP sets the `jwsdp.home` environment variable, which is required by the example build files. To ensure that you use this version of `Ant`, rather than other installations, you must add `<`*`JWSDP_HOME`*`>/jakarta-ant-1.5.1/bin` to the front of your `PATH`.

This example uses the `Ant` tool to manage the compilation of our Java source code files and creation of the deployment hierarchy. `Ant` operates under the control of a build file, normally called `build.xml`, that defines the processing steps required. This file is stored in the top-level directory of your source code hierarchy.

Like a `Makefile`, the `build.xml` file provides several targets that support optional development activities (such as erasing the deployment home directory so you can build your project from scratch). This build file includes targets for compiling the application, installing the application on a running server, reloading the modified application onto the running server, and removing old copies of the application to regenerate their content.

When we use the `build.xml` file in this example application to compile the source files, a *temporary* `/build` directory is created beneath the root. This directory contains an exact image of the binary distribution for your Web application. This directory is deleted and recreated as needed during development, so don’t edit the files in this directory.

Creating the Build and Deploy File for Ant

This example discusses how to use Ant to build and deploy this example. The first step is to create the file build.xml in the gs/ directory. The code for this file follows:>

```
<!-- Setting up the Getting Started example to prepare to
      build and deploy -->
<project name="gs-example" default="build" basedir=".">
  <target name="init">
    <tstamp/>
  </target>

<!-- Configure the context PATH for this application -->
<property name="example" value="GSApp" />
<property name="path" value="/${example}"/>
<property name="build"
value="${jwsdp.home}/docs/tutorial/examples/${example}/build"
/>

<!-- Configure properties to access the Manager application --
> <property name="url" value="http://localhost:8080/manager"/>
<property file="build.properties"/>
<property file="${user.home}/build.properties"/>

<!-- Configure custom Ant tasks for the Manager application -->

<path id="classpath">
  <fileset dir="${jwsdp.home}/common/lib">
    <include name="*.jar"/>
  </fileset>
</path>
<taskdef name="install"
      classname="org.apache.catalina.ant.InstallTask" />
<taskdef name="reload"
      classname="org.apache.catalina.ant.ReloadTask" />
<taskdef name="remove"
      classname="org.apache.catalina.ant.RemoveTask"/>

<target name="prepare" depends="init" description="Create
      build directories.">
  <mkdir dir="${build}" />
  <mkdir dir="${build}/WEB-INF" />
  <mkdir dir="${build}/WEB-INF/classes" />
</target>
```

```
<!-- Executable Targets -->

<target name="install" description="Install Web application"
    depends="build">
  <install url="${url}" username="${username}"
      password="${password}" path="${path}"
war="file:${build}"/>
</target>

<target name="reload" description="Reload Web application"
      depends="build">
  <reload url="${url}" username="${username}"
      password="${password}" path="${path}"/>
</target>

<target name="remove" description="Remove Web application">
<remove url="${url}" username="${username}"
      password="${password}" path="${path}"/>
</target>

<target name="build" depends="prepare" description="Compile
      app Java files and copy HTML and JSP pages" >
  <javac srcdir="src" destdir="${build}/WEB-INF/classes">
    <include name="**/*.java" />
    <classpath refid="classpath"/>
  </javac>
  <copy todir="${build}/WEB-INF">
    <fileset dir="web/WEB-INF" >
      <include name="web.xml" />
    </fileset>
  </copy>
  <copy todir="${build}">
    <fileset dir="web">
      <include name="*.html"/>
      <include name="*.jsp" />
      <include name="*.gif" />
    </fileset>
  </copy>
</target>

</project>
```

Compiling the Source Files

To compile the JavaBeans component (`ConverterBean.java`), we will use the `Ant` tool and run the `build` target in the `build.xml` file. The steps for doing this follow.

1. In a terminal window, go to the `gs/` directory if you are creating the application on your own, or go to the <*JWSDP_HOME*>`/docs/tutorial/examples/gs/` directory if you are compiling the example files downloaded with the tutorial.
2. Type the following command to build the Java files:

   ```
   ant build
   ```

   This command compiles the source files for the `ConverterBean`. It places the resulting class files in the <*JWSDP_HOME*>`/docs/tutorial/examples/GSApp/build/WEB-INF/classes/converterApp` directory as specified in the `build` target in `build.xml`. It also places the `index.jsp` file in the `GSApp/build` directory and places the `web.xml` file in the `GSApp/build/WEB-INF` directory. Tomcat allows you to deploy an application in an unpacked directory like this. Deploying the application is discussed in Deploying the Application (page 80).

Deploying the Application

In this release of the Java WSDP, applications are deployed using the `Ant` tool. You must start Tomcat before you can install this application using the `Ant` tool. For further information on deploying Web applications, please read Deploying Web Applications (page 103).

Starting Tomcat

To start Tomcat, type the following command in a terminal window.

```
<JWSDP_HOME>/bin/startup.sh        (Unix platform)

<JWSDP_HOME>\bin\startup          (Microsoft Windows)
```

The startup script starts the task in the background and then returns the user to the command line prompt immediately. The startup script does not completely start Tomcat for several minutes.

Note: The startup script for Tomcat can take several minutes to complete. To verify that Tomcat is running, point your browser to `http://localhost:8080`. When the Tomcat splash screen displays, you may continue. If the splash screen does not load immediately, wait up to several minutes and then retry. If, after several minutes, the Tomcat splash screen does not display, refer to the troubleshooting tips in "Unable to Locate the Server localhost:8080" Error (page 87).

Documentation for Tomcat can be found at <*JWSDP_HOME*>`/docs/tomcat/index.html`.

Installing the Application using Ant

A Web application is defined as a hierarchy of directories and files in a standard layout. In this example, the hierarchy is accessed in an unpacked form, where each directory and file exists in the file system separately. This section discusses deploying your application using the `Ant` tool defined in Creating the Build and Deploy File for Ant (page 78).

A context is a name that gets mapped to the document root of a Web application. The context of the Getting Started application is `/GSApp`. The request URL `http://localhost:8080/GSApp/index.html` retrieves the file `index.html` from the document root. To install an application to Tomcat, you notify Tomcat that a new context is available.

You notify Tomcat of a new context with the `Ant install` task from the `build.xml` file. The `Ant install` task does not require Tomcat to be restarted, but an installed application is also not remembered after Tomcat is restarted. To permanently deploy an application, see Deploying Web Applications (page 103).

The Ant install task tells a Tomcat manager application to install an application at the context specified by the path attribute and the location containing the Web application files. Read Installing Web Applications (page 102) for more information on this procedure. The steps for deploying this Web application follow.

1. In a terminal window, go to the `gs/` directory.
2. Type the following command to deploy the Web application files:

```
ant install
```

Running the Getting Started Application

To run the application, you need to make sure that Tomcat is running, then run the JSP page from a Web browser.

Running the Web Client

To run the Web client, point your browser at the URL:

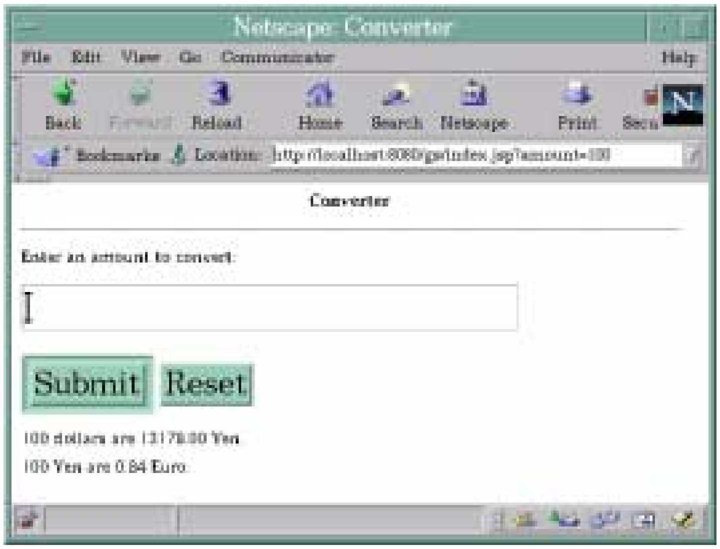
```
http://localhost:8080/GSApp
```

In this release of the Java WSDP, Tomcat requires that the host be `localhost`, which is the machine on which Tomcat is running. In this example, the context for this application is "`GSApp`", which was defined in the `build.xml` file.

To test the application,

1. Enter 100 in the "Enter an amount to convert" field.
2. Click Submit.

Figure 3–1 shows the running application.



http://localhost:8080/gs/index.jsp?amount=100

Converter

Enter an amount to convert:

100 dollars are 13178.00 Yen

100 Yen are 0.84 Euro

Figure 3–1 ConverterBean Web Client

Shutting Down Tomcat

When you are finished testing and developing your application, you should shut down Tomcat.

```
<JWSDP_HOME>/bin/shutdown.sh        (Unix platform)

<JWSDP_HOME>\bin\shutdown           (Microsoft Windows)
```

Using admintool

The Java Web Services Developer Pack includes the Tomcat Web Server Administration Tool, referred to hereafter as `admintool` for ease of reference. The `admintool` Web application can be used to manipulate Tomcat while it is running. For example, you can add and/or configure contexts, hosts, realms, and connectors, or set up users and roles for container-managed security.

To start `admintool`, follow these steps.

1. Start Tomcat as described in Starting Tomcat (page 80).
2. Start a Web browser.
3. In the Web browser, point to the following URL:

   ```
   http://localhost:8080/admin
   ```

   This command invokes the `admin` Web application. Before you can use this application you must add your user name/password combination and associate the role name `admin` with it. The initial user name and password necessary to access this tool are set up during Java WSDP installation. If you've forgotten the user name and password, you can view `<JWSDP_HOME>/conf/tomcat-users.xml` with any text editor. This file contains an element `<user>` for each individual user, which might look something like this:

   ```
   <user name="adeveloper" password="secret"
     roles="admin, manager" />
   ```

4. Log in to `admintool` using a user name and password combination that has been assigned the role of `admin`. This user name and password must match the user name and password in the `build.properties` file.
5. When you have finished, log out of `admintool` by selecting Logout from the upper pane.

This section discussing setting up roles, groups, and users using `admintool`. See Appendix A, Tomcat Administration Tool, for information on using `admintool` to create, delete, and/or configure:

- The Tomcat Server.
- Services that run on the Tomcat Server, plus the elements that are nested within the Services, such as Hosts, Contexts, Realms, Connectors, Loggers, and Valves.
- Resources such as Data Sources, Environment Entries, and User Database.

Understanding Roles, Groups, and Users

The Tomcat server authentication service includes the following components:

- *Role* - an abstract name for the permission to access a particular set of resources. A *role* can be compared to a key that can open a lock. Many people might have a copy of the key, and the lock doesn't care who you are, just that you have the right key.
- *User* - an individual (or application program) identity that has been authenticated (authentication was discussed in the previous section). A user can have a set of *roles* associated with that identity, which entitles them to access all resources protected by those roles.
- *Group* - a set of authenticated *users* classified by common traits such as job title or customer profile. Groups are also associated with a set of *roles*, and every user that is a member of a group inherits all of the roles assigned to that group.
- *Realm* - a complete database of *roles*, *users*, and *groups* that identify valid users of a Web application (or a set of Web applications).

These concepts are addressed in more detail in Managing Roles and Users (page 704). More information on `admintool` is available in Appendix A, Tomcat Administration Tool.

Adding Roles Using admintool

To set up new roles for container-managed security, follow these instructions. Additions, deletions, and changes made in `admintool` are written to the `tomcat-users.xml file`.

1. Scroll down the left pane of `admintool` to the User and Group Administration node.
2. Select Role Administration.
3. From the Roles List, select Create New Role.
4. Enter a Role Name and Description, for example `Customer` or `User`.
5. Select Save.

Adding Users Using admintool

To set up new users for container-managed security, follow these instructions. Additions, deletions, and changes made in `admintool` are written to the `tomcat-users.xml file`.

1. Scroll down the left pane of `admintool` to the User and Group Administration node.
2. Select User Administration.
3. From the Users List, select Create New User.
4. Enter a User Name, Password, and select a Role for the new user. If you select the `admin` role for the new user, the user will be able to access `admintool`.
5. Select Save.

Modifying the Application

Since the Java Web Services Developer Pack is intended for experimentation, it supports iterative development. Whenever you make a change to an application, you must redeploy and reload the application. The tasks we defined in the `build.xml` file make it simple to deploy changes to both the `ConverterBean` and the JSP page.

In the `build.xml` file, we set up a target to install the application on the running Tomcat server and a target to reload the application onto the running Tomcat server. These tasks are accomplished using the Tomcat Server Manager Tool, which is the `manager` Web application. You may use the user name/password combination that you set up during Java WSDP installation because it will have the role name of manager associated with it. If you've forgotten the user name/password combination that you set up during installation, you can look it up in <*JWSDP_HOME*>`/conf/tomcat-users.xml`, which can be viewed with any text editor.

The Tomcat reference documentation distributed with the Java WSDP contains information about the manager application.

Modifying a Class File

To modify a class file in a Java component, you change the source code, recompile it, and redeploy the application. When using the Tomcat `manager` Web application, you do not need to stop and restart Tomcat in order to redeploy the changed application. For example, suppose that you want to change the exchange rate in the `yenRate` property of the `ConverterBean` component:

1. Edit `ConverterBean.java` in the source directory.
2. Recompile `ConverterBean.java` by typing `ant build`.
3. Redeploy `ConverterBean.java` by typing `ant reload`.
4. Reload the JSP page in the Web browser.

Modifying the Web Client

To modify a JSP page, you change the source code and redeploy the application.When using the Tomcat `manager` Web application, you do not need to stop and restart Tomcat in order to redeploy the changed Web client. For example, suppose you wanted to modify a font or add additional descriptive text to the JSP page. To modify the Web client:

1. Edit `index.jsp` in the source directory.
2. Reload the Web application by typing `ant reload`.
3. Reload the JSP page in the Web browser.

Common Problems and Their Solutions

Use the following guidelines for troubleshooting any problems you have creating, compiling, installing, deploying, and running the example application.

Errors Starting Tomcat

“Out of Environment Space” Error

Symptom: An “out of environment space” error when running the startup and shutdown batch files in Microsoft Windows 9X/ME-based operating systems.

Solution: In the Microsoft Windows Explorer, right-click on the `startup.bat` and `shutdown.bat` files. Select Properties, then select the Memory tab. Increase the Initial Environment field to something like 4096. Select Apply.

After you select Apply, shortcuts will be created in the directory you use to start and stop the container.

“Unable to Locate the Server localhost:8080” Error

Symptom: an “unable to locate server” error when trying to load a Web application in a browser.

Solution: Tomcat can take quite some time before fully loading, so first of all, make sure you’ve allowed at least 5 minutes for Tomcat to load before continuing troubleshooting. To verify that Tomcat is running, point your browser to `http://localhost:8080`. When the Tomcat index screen displays, you may continue. If the index screen does not load immediately, wait up to several minutes and then retry. If Tomcat still has not loaded, check the log files, as explained below, for further troubleshooting information.

When Tomcat starts up, it initializes itself and then loads all the Web applications in <*JWSDP_HOME*>`/webapps`. When you run Tomcat by calling `startup.sh`, the server messages are logged to <*JWSDP_HOME*>`/logs/launcher.server.log`. The progress of loading Web applications can be viewed in the file <*JWSDP_HOME*>`/logs/jwsdp_log.`<*date*>`.txt`.

Compilation Errors

Ant Cannot Locate the Build File

Symptom: When you type `ant build`, these messages appear:

```
Buildfile: build.xml does not exist!
Build failed.
```

Solution: Start `Ant` from the *<JWSDP_HOME>*`/docs/tutorial/examples/gs/` directory, or from the directory where you created the application. If you want to run `Ant` from your current directory, then you must specify the build file on the command line. For example, you would type this command on a single line:

```
ant -buildfile
<JWSDP_HOME>/docs/tutorial/examples/gs/build.xml
build
```

The Compiler Cannot Resolve Symbols

Symptom: When you type `ant build`, the compiler reports many errors, including these:

```
cannot resolve symbol
. . .
BUILD FAILED
. . .
Compile failed, messages should have been provided
```

Solution: Make sure you are using the version of `Ant` that ships with this version of the Java WSDP. The best way to ensure that you are using this version is to use the full PATH to the `Ant` files to build the application, *<JWSDP_HOME>*`/jakarta-ant-1.5.1/bin/ant build`. Other versions may not include all of the functionality expected by the example application build files.

"Connection refused" Error

Symptom: When you type `ant install` at the terminal prompt, you get the following message:

```
<JWSDP_HOME>/docs/tutorial/examples/gs/build.xml:82:
java.net.ConnectException: Connection refused
```

Solution: Tomcat has not fully started. Wait a few minutes, and then attempt to install the application again. For more information on troubleshooting Tomcat startup, see "Unable to Locate the Server localhost:8080" Error (page 87).

When attempting to run the install task, the system appears to hang.

Symptom: When you type `ant install`, the system appears to hang.

Solution: The Tomcat startup script starts Tomcat in the background and then returns the user to the command line prompt immediately. Even though you are returned to the command line, the startup script may not have completely started Tomcat. If the install task does not run immediately, wait up to several minutes and then retry the install task. To verify that Tomcat is running, point your browser to `http://localhost:8080`. When the Tomcat index screen displays, you may continue. If the splash screen does not load immediately, wait up to several minutes and then retry. If Tomcat still has not loaded, check the log files, as explained below, for further troubleshooting information.

When Tomcat starts up, it initializes itself and then loads all the Web applications in *<JWSDP_HOME>*`/webapps`. When you run Tomcat by calling `startup.sh`, the server messages are logged to *<JWSDP_HOME>*`/logs/launcher.server.log`. The progress of loading Web applications can be viewed in the file *<JWSDP_HOME>*`/logs/jwsdp_log.`*<date>*`.txt`.

Deployment Errors

Server returned HTTP response code: 401 for URL ...

Symptom: When you type `ant install`, these message appear:

```
BUILD FAILED
/home/you/gs/build.xml:44:
java.io.IOException: Server returned HTTP response code: 401
for URL: http://localhost:8080/manager/install?path= ...
```

Solution: Make sure that the user name and password in your `build.properties` file match a user name and password with the role of `manager` in the `tomcat-users.xml` file. For more information on setting up this information, see Creating the Build Properties File (page 71).

Failure to run client application

Symptom: The browser reports that the page cannot be found (HTTP 404).

Solution: The startup script starts the task in the background and then returns the user to the command line prompt immediately. Even though you are returned to the command line, the startup script may not have completely started Tomcat. If the Web Client does not run immediately, wait up to a minute and then retry to load the Web client. For more information on troubleshooting the startup of Tomcat, see "Unable to Locate the Server localhost:8080" Error (page 87).

The localhost Machine Is Not Found

Symptom: The browser reports that the page cannot be found (HTTP 404).

Solution: Sometimes when you are behind a proxy and the firewall will not let you access the `localhost` machine. To fix this, change the proxy setting so that it does not use the proxy to access `localhost`.

To do this in the Netscape Navigator™ browser, select Edit -> Preferences -> Advanced -> Proxies and select `No Proxy for: localhost`. In Internet Explorer, select Tools -> Internet Options -> Connections -> LAN Settings.

The Application Has Not Been Deployed

Symptom: The browser reports that the page cannot be found (HTTP 404).

Solution: Deploy the application. For more detail, see Deploying the Application (page 80).

"Build Failed: Application Already Exists at Path" Error

Symptom: When you enter `ant install` at a terminal prompt, you get this message:

```
[install] FAIL - Application already exists at path /GSApp

BUILD FAILED

<JWSDP_HOME>/docs/tutorial/examples/gs/build.xml:82: FAIL -
Application already exists at path /GSApp
```

This application has already been installed. If you've made changes to the application since it was installed, use `ant reload` to update the application in Tomcat.

HTTP 500: No Context Error

Symptom: Get a No Context Error when attempting to run a deployed application.

Solution: This error means that Tomcat is loaded, but it doesn't know about your application. If you have not deployed the application by running `ant remove`, `ant build`, `ant install`, `ant reload`, do so now.

Solution: If Tomcat is loading, but has not yet loaded all of the existing contexts, you will get this error. Continue to select the Reload or Refresh button on your browser until either the application loads or you get a different error message.

Further Information

- *Tomcat Administration Tool*. Read Tomcat Administration Tool (page 785) for further information about using `admintool` to configure the behavior of Tomcat without having to stop and restart it.
- *Tomcat Configuration Reference*. For further information on the elements that can be used to configure the behavior of Tomcat, read the Tomcat Configuration Reference, which can be found at `<JWSDP_HOME>/docs/tomcat/config/index.html`.
- *Class Loader How-To*. This document discusses decisions that application developers and deployers must make about where to place class and resource files to make them available to Web applications. This document can be found at `<JWSDP_HOME>/docs/tomcat/class-loader-howto.html`.
- *JNDI Resources How-To*. This document discusses configuring JNDI Resources, Tomcat Standard Resource Factories, JDBC Data Sources, and Custom Resource Factories. This document can be found at `<JWSDP_HOME>/docs/tomcat/jndi-resources-howto.html`.
- *Manager Application How-To*. This document describes using the Manager Application to deploy a new Web application, undeploy an existing application, or reload an existing application without having to shut down and restart Tomcat. This document can be found at `<JWSDP_HOME>/docs/tomcat/manager-howto.html`.
- *Proxy Support How-To*. This document discusses running behind a proxy server (or a web server that is configured to behave like a proxy server). In particular, this document discusses how to manage the values returned by the calls from Web applications that ask for the server name and port num-

ber to which the request was directed for processing. This document can be found at <*JWSDP_HOME*>/docs/tomcat/proxy-howto.html.

- *Realm Configuration How-To*. This document discusses how to configure Tomcat to support container-managed security by connecting to an existing database of user names, passwords, and user roles. This document can be found at <*JWSDP_HOME*>/docs/tomcat/realm-howto.html.
- *Security Manager How-To*. This document discusses the use of a SecurityManager while running Tomcat to protect your server from unauthorized servlets, JSPs, JSP beans, and tag libraries. This document can be found at <*JWSDP_HOME*>/docs/tomcat/security-manager-howto.html.
- *SSL Configuration How-To*. This document discusses how to install and configure SSL support on Tomcat. Configuring SSL support on Tomcat using Java WSDP is discussed in Installing and Configuring SSL Support (page 721). The Tomcat documentation at <*JWSDP_HOME*>/docs/tomcat/ssl-howto.html also discusses this topic, however, the information in this tutorial is more up-to-date for the version of Tomcat shipped with the Java WSDP.

4

Web Applications

Stephanie Bodoff

A Web application is a dynamic extension of a Web server. There are two types of Web applications:

- Presentation-oriented. A presentation-oriented Web application generates dynamic Web pages containing various types of markup language (HTML, XML, and so on) in response to requests.
- Service-oriented. A service-oriented Web application implements the endpoint of a fine-grained Web service. Service-oriented Web applications are often invoked by presentation-oriented applications.

In the Java 2 Platform, *Web components* provide the dynamic extension capabilities for a Web server. Web components are either Java Servlets or JSP pages. Servlets are Java programming language classes that dynamically process requests and construct responses. JSP pages are text-based documents that execute as servlets but allow a more natural approach to creating static content. Although servlets and JSP pages can be used interchangeably, each has its own strengths. Servlets are best suited to service-oriented Web applications and managing the control functions of a presentation-oriented application, such as dispatching requests and handling nontextual data. JSP pages are more appropriate for generating text-based markup such as HTML, SVG, WML, and XML.

Web components are supported by the services of a runtime platform called a *Web container*. In the Java Web Services Developer Pack (Java WSDP) Web components run in the Tomcat Web container. The Web container provides services such as request dispatching, security, concurrency, and life cycle management. It also gives Web components access to APIs such as naming, transactions, and e-mail.

This chapter describes the organization, configuration, and installation and deployment procedures for Web applications. Chapters 12 and 11 cover how to develop Web components for service-oriented Web applications. Chapters 14 and 15 cover how to develop the Web components for presentation-oriented Web applications. Many features of JSP technology are determined by Java Servlet technology, so you should familiarize yourself with that material even if you do not intend to write servlets.

Most Web applications use the HTTP protocol, and support for HTTP is a major aspect of Web components. For a brief summary of HTTP protocol features see HTTP Overview (page 849).

Web Application Life Cycle

A Web application consists of Web components, static resource files such as images, and helper classes and libraries. The Java WSDP provides many supporting services that enhance the capabilities of Web components and make them easier to develop. However, because it must take these services into account, the process for creating and running a Web application is different from that of traditional stand-alone Java classes.

Certain aspects of Web application behavior can be configured when the application is deployed. The configuration information is maintained in a text file in XML format called a *Web application deployment descriptor*. A deployment descriptor must conform to the schema described in the Java Servlet specification.

The process for creating, deploying, and executing a Web application can be summarized as follows:

1. Develop the Web component code (including possibly a deployment descriptor).
2. Build the Web application components along with any static resources (for example, images) and helper classes referenced by the component.
3. Install or deploy the application into a Web container.
4. Access a URL that references the Web application.

Developing Web component code is covered in the later chapters. Steps 2 through 4 are expanded on in the following sections and illustrated with a Hello, World style presentation-oriented application. This application allows a user to enter a name into an HTML form (Figure 4–1) and then displays a greeting after the name is submitted (Figure 4–2):

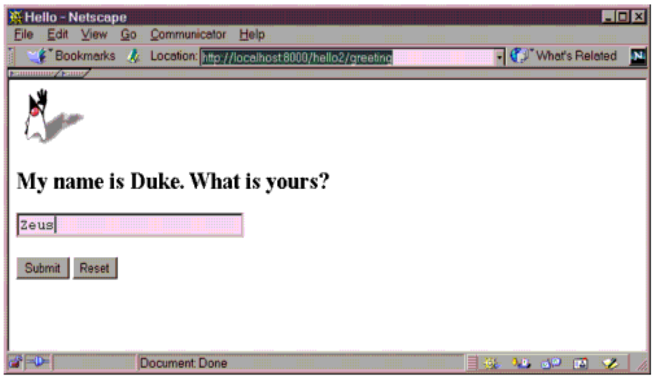

Hello - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://localhost:8000/hello2/greeting
What's Related
My name is Duke. What is yours?
Zeus
Submit Reset
Document: Done

Figure 4–1 Greeting Form

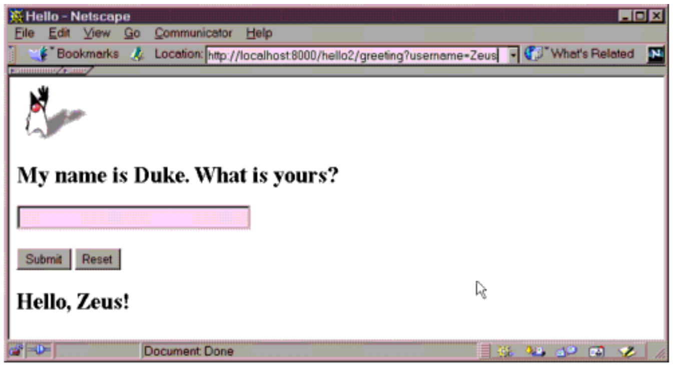

Hello - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://localhost:8000/hello2/greeting?username=Zeus
What's Related
My name is Duke. What is yours?
Submit Reset
Hello, Zeus!
Document: Done

Figure 4–2 Response

The Hello application contains two Web components that generate the greeting and the response. This tutorial has two versions of the application: a servlet version called Hello1, in which the components are implemented by two servlet classes, `GreetingServlet.java` and `ResponseServlet.java`, and a JSP version called Hello2, in which the components are implemented by two JSP pages, `greeting.jsp` and `response.jsp`. The two versions are used to illustrate the tasks involved in packaging, deploying, and running an application that contains Web components. If you are viewing this tutorial online, you must download the tutorial bundle to get the source code for this example. See Running the Examples (page xiii).

Web Application Archives

If you want to distribute a Web application, you package it in a Web application archive (WAR), which is a JAR similar to the package used for Java class libraries. In addition to Web components, a Web application archive can contain other files including the following:

- Server-side utility classes (database beans, shopping carts, and so on). Often these classes conform to the JavaBeans component architecture.
- Static Web presentation content (HTML, image, and sound files, and so on)
- Client-side classes (applets and utility classes)

Web components and static Web content files are called *Web resources*.

A Web application can run from a WAR file or from an unpacked directory laid out in the same format as a WAR.

WAR Directory Structure

The top-level directory of a WAR is the *document root* of the application. The document root is where JSP pages, client-side classes and archives, and static Web resources are stored.

The document root contains a subdirectory called `WEB-INF`, which contains the following files and directories:

- `web.xml` - The Web application deployment descriptor
- Tag library descriptor files (see Tag Library Descriptors, page 647)
- `classes` - A directory that contains server-side classes: servlets, utility classes, and JavaBeans components
- `lib` - A directory that contains JAR archives of libraries (tag libraries and any utility libraries called by server-side classes)

You can also create application-specific subdirectories (that is, package directories) in either the document root or the `WEB-INF/classes` directory.

Tutorial Example Directory Structure

To facilitate iterative development and keep Web application source separate from compiled files, the source code for the tutorial examples is stored in the following structure under each application directory *`mywebapp`*:

- `build.xml` - `Ant` build file
- `context.xml` - Optional application configuration file
- `src` - Java source of servlets and JavaBeans components
- `web` - JSP pages and HTML pages, images

The `Ant` build files (`build.xml`) distributed with the examples contain targets to create an unpacked WAR structure in the `build` subdirectory of *`mywebapp`*, copy and compile files into that directory, and invoke the `manager` (see Tomcat Web Application Manager, page 825) commands via special `Ant` tasks to install, reload, remove, deploy, and undeploy applications. The tutorial example `Ant` targets are:

- `prepare` - Creates `build` directory and WAR subdirectories.
- `build` - Compiles and copies the *`mywebapp`* Web application files into the `build` directory.
- `install` - Notifies Tomcat to install an application (see Installing Web Applications, page 102) using the `Ant` `install` task.
- `reload` - Notifies Tomcat to reload the application (see Updating Web Applications, page 105) using the `Ant` `reload` task.
- deploy - Notifies Tomcat to deploy the application (see Deploying Web Applications, page 103) using the `Ant` `deploy` task.
- undeploy - Notifies Tomcat to undeploy the application (see Undeploying Web Applications, page 107) using the `Ant` `undeploy` task.
- `remove` - Notifies Tomcat to remove the application (see Removing Web Applications, page 107) using the `Ant` `remove` task.

Creating a WAR

You can manually create a WAR file in two ways:

- With the JAR tool distributed with the J2SE SDK. You simply execute the following command in the build directory of a tutorial example:

```
jar cvf mywebapp.war .
```

- With the `Ant` `war` task

Both of these methods require you to have created a Web application deployment descriptor.

Configuring Web Applications

Web applications are configured via elements contained in Web application deployment descriptors. You can manually create descriptors using a text editor. The following sections give a brief introduction to the Web application features you will usually want to configure. A number of security parameters can be specified; these are covered in Chapter 18. For a complete listing and description of the features, see the Java Servlet specification.

In the following sections, some examples demonstrate procedures for configuring the Hello, World application. If Hello,World does not use a specific configuration feature, the section gives uses other examples for illustrating the deployment descriptor element and describes generic procedures for specifying the feature.

Note: Descriptor elements must appear in the deployment descriptor in the following order: `icon`, `display-name`, `description`, `distributable`, `context-param`, `filter`, `filter-mapping`, `listener`, `servlet`, `servlet-mapping`, `session-config`, `mime-mapping`, `welcome-file-list`, `error-page`, `taglib`, `resource-env-ref`, `resource-ref`, `security-constraint`, `login-config`, `security-role`, `env-entry`.

Prolog

Since the deployment descriptor is an XML document, it requires a prolog. The prolog of the Web application deployment descriptor is as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN" "http://java.sun.com/dtd/web-
app_2_3.dtd">
```

Alias Paths

When a request is received by Tomcat it must determine which Web component should handle the request. It does so by mapping the URL path contained in the request to a Web component. A URL path contains the context root (described in Installing Web Applications, page 102) and an *alias* path

```
http://<host>:8080/context_root/alias_path
```

Before a servlet can be accessed, the Web container must have least one alias path for the component. The alias path must start with a `/` and end with a string or a wildcard expression with an extension (`*.jsp`, for example). Since Web containers automatically map an alias path that ends with `*.jsp`, you do not have to specify an alias path for a JSP page unless you wish to refer to the page by a name other than its file name. In the example discussed in Updating Web Applications (page 105), the greeting page has an alias but `response.jsp` is referenced by its file name.

To set up the mappings servlet version of the Hello application in the Web deployment descriptor, you must add the following `servlet` and `servlet-mapping` elements to the Web application deployment descriptor. To define an alias for a JSP page, you must replace the `servlet-class` subelement with a `jsp-file` subelement in the `servlet` element.

```
<servlet>
  <servlet-name>greeting</servlet-name>
  <display-name>greeting</display-name>
  <description>no description</description>
  <servlet-class>GreetingServlet</servlet-class>
</servlet>
<servlet>
  <servlet-name>response</servlet-name>
  <display-name>response</display-name>
  <description>no description</description>
  <servlet-class>ResponseServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>greeting</servlet-name>
  <url-pattern>/greeting</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>response</servlet-name>
  <url-pattern>/response</url-pattern>
</servlet-mapping>
```

Context and Initialization Parameters

The Web components in a WAR share an object that represents their application context (see Accessing the Web Context, page 598). You can pass parameters to the context or Web component. To do so you must add a `context-param` or `init-param` element to the Web application deployment descriptor. `context-param` is a subelement of the top-level `web-app` element. `init-param` is a subelement of the `servlet` element. Here is the element used to declare a context parameter that sets the resource bundle used in the example discussed in Chapter 17:

```
<web-app>
  <context-param>
     <param-name>
        javax.servlet.jsp.jstl.fmt.localizationContext
     </param-name>
     <param-value>messages.BookstoreMessages</param-value>
  </context-param>
  ...
</web-app>
```

Event Listeners

To add an event listener class (described in Handling Servlet Life Cycle Events, page 575), you must add a `listener` element to the Web application deployment descriptor. Here is the element that declares the listener class used in chapters 14 and 17:

```
<listener>
  <listener-class>listeners.ContextListener</listener-class>
</listener>
```

Filter Mappings

A Web container uses filter mapping declarations to decide which filters to apply to a request, and in what order (see Specifying Filter Mappings, page 592). The container matches the request URI to a servlet as described in Alias Paths (page 99). To determine which filters to apply, it matches filter mapping declarations by servlet name or URL pattern. The order in which filters are invoked is the order in which filter mapping declarations that match a request URI for a servlet appear in the filter mapping list.

To specify a filter mapping, you must add an `filter` and `filter-mapping` elements to the Web application deployment descriptor. Here is the element used to declare the order filter and map it to the `ReceiptServlet` discussed in Chapter 14:

```
<filter>
  <filter-name>OrderFilter<filter-name>
  <filter-class>filters.OrderFilter<filter-class>
</filter>
<filter-mapping>
  <filter-name>OrderFilter</filter-name>
  <url-pattern>/receipt</url-pattern>
</filter-mapping>
```

Error Mappings

You can specify a mapping between the status code returned in an HTTP response or a Java programming language exception returned by any Web component and a Web resource (see Handling Errors, page 577). To set up the mapping, you must add an `<error-page>` element to the deployment descriptor. Here is the element use to map `OrderException` to the page `errorpage.html` used in Chapter 14:

```
<error-page>
  <exception-type>exception.OrderException</exception-type>
  <location>/errorpage.html</location>
</error-page>
```

Note: You can also define error pages for a JSP page contained in a WAR. If error pages are defined for both the WAR and a JSP page, the JSP page's error page takes precedence.

References to Environment Entries, Resource Environment Entries, or Resources

If your Web components reference environment entries, resource environment entries, or resources such as databases, you must declare the references with `<env-entry>`, `<resource-env-ref>`, or `<resource-ref>` elements in the Web application deployment descriptor. Here is the element used to declare a reference to the data source used in the Web technology chapters in this tutorial:

```
<resource-ref>
  <res-ref-name>jdbc/BookDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

Installing Web Applications

A *context* is a name that gets mapped to a Web application. For example, the context of the Hello1 application is `/hello1`. To install an application to Tomcat, you notify Tomcat that a new context is available.

You notify Tomcat of a new context with the `Ant install` task. Note that an installed application is not available after Tomcat is restarted. To permanently deploy an application, see Deploying Web Applications (page 103).

The `Ant install` task tells the `manager` running at the location specified by the `url` attribute to install an application at the context specified by the `path` attribute and the location containing the Web application files specified with the `war` attribute. The value of the `war` attribute can be a WAR file `jar:file:/path/to/bar.war!/` or an unpacked directory `file:/path/to/foo`.

```
<install url="url" path="mywebapp" war="file:build"
  username="username" password="password" />
```

The `username` and `password` attributes are discussed in Tomcat Web Application Manager (page 825).

Instead of providing a `war` attribute, you can specify configuration information with the `config` attribute:

```
<install url="url"
  path="mywebapp" config="file:build/context.xml"
  username="username" password="password"/>
```

The `config` attribute points to a configuration file that contains a context entry of the form:

```
<Context path="/bookstore1"
  docBase="../docs/tutorial/examples/web/bookstore1/build"
  debug="0">
```

Note that the context entry implicitly specifies the location of the Web application files through its `docBase` attribute.

The tutorial example build files contain an `Ant install` *target* that invokes the `Ant install` *task*:

```
<target name="install"
  description="Install web application" depends="build">
  <install url="${url}" path="${mywebapp}"
    config="file:build/context.xml"
    username="${username}" password="${password}"/>
</target>
```

The `Ant install` task requires that a Web application deployment descriptor (`web.xml`) be available. All of the tutorial example applications are distributed with a deployment descriptor.

To install the Hello1 application described in Web Application Life Cycle (page 94):

1. In a terminal window, go to *`<JWSDP_HOME>`*`/docs/tutorial/examples/web/hello1`.
2. Make sure Tomcat is started.
3. Execute `ant install`. The `install` target notifies Tomcat that the new context is available.

Deploying Web Applications

You can use the `Ant deploy` task to permanently deploy a context to Tomcat while Tomcat is running:

```
<deploy url="url" path="mywebapp"
  war="file:/path/to/mywebapp.war"
  username="username" password="password" />
```

Unlike the `install` task, which can reference an unpacked directory, the `deploy` task requires a WAR. The task uploads the WAR to Tomcat and starts the application. You can deploy to a remote server with this task.

The following other deployment methods are also available, but they require you to restart Tomcat:

- Copy a Web application directory or WAR to `<`*`JWSDP_HOME`*`>/webapps`.
- Copy a configuration file named *`mywebapp`*`.xml` containing a context entry to `<`*`JWSDP_HOME`*`>/webapps`. The format of a context entry is described in the *Server Configuration Reference* at `<`*`JWSDP_HOME`*`>/docs/tomcat/config/context.html`. Note that the context entry implicitly specifies the location of the Web application files through its `docBase` attribute. For example, here is the context entry for the application discussed in Chapter 14:

```
<Context path="/bookstore1"
   docBase="../docs/tutorial/examples/web/
      bookstore1/build" debug="0">
```

Some of the example build files contain an `Ant deploy` *target* that invokes the `Ant deploy` *task*.

Listing Installed and Deployed Web Applications

If you want to list all Web applications currently available on Tomcat you use the `Ant list` task:

```
<list url="url" username="username" password="password" />
```

The tutorial example build files contain an `Ant list` *target* that invokes the `Ant list` *task*.

You can also see list applications by running the Manager Application:

```
http://<host>:8080/manager/list
```

Running Web Applications

A Web application is executed when a Web browser references a URL that is mapped to component. Once you have installed or deployed the `Hello1` application, you can run the Web application by pointing a browser at

```
http://<host>:8080/hello1/greeting
```

Replace *`<host>`* with the name of the host running Tomcat. If your browser is running on the same host as Tomcat, you may replace *`<host>`* with `localhost`.

Updating Web Applications

During development, you will often need to make changes to Web applications. After you modify a servlet, you must

1. Recompile the servlet class.
2. Update the application in the server.
3. Reload the URL in the client.

When you update a JSP page, you do not need to recompile or reload the application, because Tomcat does this automatically.

To try this feature, modify the servlet version of the Hello application. For example, you could change the greeting returned by `GreetingServlet` to be:

```
<h2>Hi, my name is Duke. What's yours?</h2>
```

To update the file:

1. Edit `GreetingServlet.java` in the source directory `<JWSDP_HOME>/docs/tutorial/examples/web/hello1/src`.
2. Run `ant build`. This task recompiles the servlet into the `build` directory.

The procedure for updating the application in the server depends on whether you installed it using the `Ant install` task or deployed it using the `Ant deploy` task.

Reloading Web Applications

If you have installed an application using the `Ant install` command, you update the application in the server using the `Ant reload` task:

```
<reload url="url" path="mywebapp"
  username="username" password="password" />
```

The example build files contain an `Ant remove` *target* that invokes the `Ant remove` *task*. Thus to update the `Hello1` application in the server, execute `ant reload`. To view the updated application, reload the `Hello1` URL in the client. Note that the `reload` task only picks up changes to Java classes, not changes to the `web.xml` file. To reload `web.xml`, remove the application (see Removing Web Applications, page 107) and install it again.

You should see the screen in Figure 4–3 in the browser:

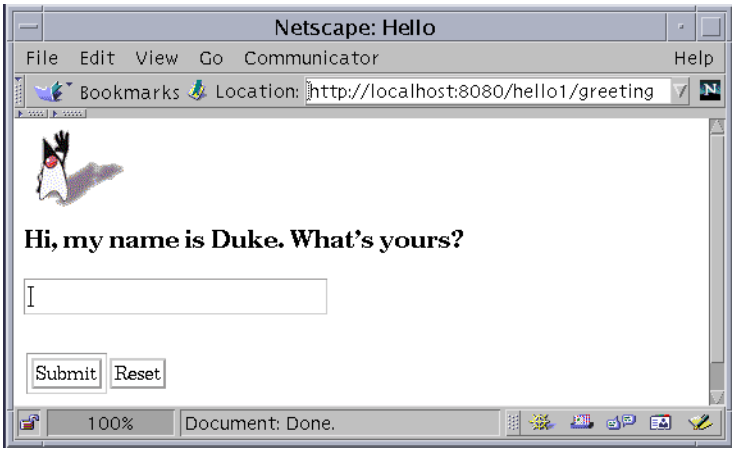

Netscape: Hello
File Edit View Go Communicator Help
Bookmarks Location: http://localhost:8080/hello1/greeting
Hi, my name is Duke. What's yours?
Submit
Reset
100%
Document: Done.

Figure 4–3 New Greeting

To try this on the JSP version of the example, first build and deploy the JSP version of the Hello application:

1. In a terminal window, go to *<JWSDP_HOME>*`/docs/tutorial/examples/web/hello2`.
2. Run `ant build`. The `build` target will spawn any necessary compilations and copy files to the *<JWSDP_HOME>*`/docs/tutorial/examples/web/hello2/build` directory.

3. Run `ant install`. The `install` target copies the build directory to `<JWSDP_HOME>/webapps` and notifies Tomcat that the new application is available.

Modify one of the JSP files. Then run `ant build` to *copy* the modified file into `docs/tutorial/examples/web/hello2/build`. Remember, you don't have to reload the application in the server, because Tomcat automatically detects when a JSP page has been modified. To view the modified application, reload the `Hello2` URL in the client.

Redeploying Web Applications

If you have deployed the application using the `Ant deploy` task you update the application by using the `Ant undeploy` task (see Undeploying Web Applications, page 107) and then using the `Ant deploy` task.

Removing Web Applications

If you want to take an installed Web application out of service, you invoke the `Ant remove` task:

```
<remove url="url" path="mywebapp"
  username="username" password="password" />
```

The example build files contain an `Ant remove` *target* that invokes the `Ant remove` *task*.

Undeploying Web Applications

If you want to remove a deployed Web application, you use the `Ant undeploy` task:

```
<undeploy url="url" path="mywebapp"
  username="username" password="password" />
```

Some of the example build files contain an `Ant undeploy` *target* that invokes the `Ant undeploy` *task*.

Internationalizing and Localizing Web Applications

Internationalization is the process of preparing an application to support various languages and data formats. *Localization* is the process of adapting an internationalized application to support a specific language or locale. Although all client user interfaces should be internationalized and localized, it is particularly important for Web applications because of the far-reaching nature of the Web. For a good overview of internationalization and localization, see

```
http://java.sun.com/docs/books/tutorial/i18n/index.html
```

There are two approaches to internationalizing a Web application:

- Provide a version of the JSP page in each of the target locales and have a controller servlet dispatch the request to the appropriate page (depending on the requested locale). This approach is useful if large amounts of data on a page or an entire Web application need to be internationalized.
- Isolate any locale-sensitive data on a page (such as error messages, string literals, or button labels) into resource bundles, and access the data so that the corresponding translated message is fetched automatically and inserted into the page. Thus, instead of creating strings directly in your code, you create a resource bundle that contains translations and read the translations from that bundle using the corresponding key. A resource bundle can be backed by a text file (properties resource bundle) or a class (list resource bundle) containing the mappings.

In the following chapters on Web technology, the Duke's Bookstore example is internationalized and localized into English and Spanish. The key and value pairs are contained in list resource bundles named `messages.BookMessage_*.class`. To give you an idea of what the key and string pairs in a resource bundle look like, here are a few lines from the file `messages.BookMessages.java`.

```
{"TitleCashier", "Cashier"},
{"TitleBookDescription", "Book Description"},
{"Visitor", "You are visitor number "},
{"What", "What We”re Reading"},
{"Talk", " talks about how Web components can transform the way
you develop applications for the Web. This is a must read for
any self respecting Web developer!"},
{"Start", "Start Shopping"},
```

To get the correct strings for a given user, a Web component retrieves the locale (set by a browser language preference) from the request, opens the resource bundle for that locale, and then saves the bundle as a session attribute (see Associating Attributes with a Session, page 599):

```
ResourceBundle messages = (ResourceBundle)session.
  getAttribute("messages");
  if (messages == null) {
    Locale locale=request.getLocale();
    messages = ResourceBundle.getBundle("WebMessages",
      locale);
    session.setAttribute("messages", messages);
  }
```

A Web component retrieves the resource bundle from the session:

```
ResourceBundle messages =
  (ResourceBundle)session.getAttribute("messages");
```

and looks up the string associated with the key `TitleCashier` as follows:

```
messages.getString(“TitleCashier”);
```

This has been a very brief introduction to internationalizing Web applications. For more information on this subject see the Java BluePrints:

```
http://java.sun.com/blueprints
```

Accessing Databases from Web Applications

Data that is shared between Web components and persistent between invocations of a Web application is usually maintained by a database. Web applications use the JDBC 2.0 API to access relational databases. For information on this API, see

```
http://java.sun.com/docs/books/tutorial/jdbc
```

The Examples

The examples discussed in the chapters 14, 15, 16, and 17 require a database. For this release we have tested the examples with the PointBase 4.5 database and we provide an `Ant` build file to create the database tables and populate the database. The remainder of this section describes how to

- Install and start the PointBase database server
- Populate the example tables
- Configure the Web application to reference a data source
- Define a data source in Tomcat
- Configure Tomcat to map the reference to the data source

Installing and Starting the Database Server

You can download an evaluation copy of the PointBase 4.5 database from:

```
http://www.pointbase.com
```

Make sure to choose a platform-specific (UNIX or Windows) installation package. Install the client and server components. After you have downloaded and installed the PointBase database, do the following:

1. Add a `pb.home` property to your `build.properties` file (discussed in Managing the Examples, page xiv) that points to your PointBase install directory. On Windows the syntax of the entry must be

   ```
   pb.home=drive:\\<PB_HOME>
   ```

2. Copy <*PB_HOME*>`/lib/pbclient45.jar` to <*JWSDP_HOME*>`/common/lib` to make the PointBase client library available to the example applications. If Tomcat is running, restart it so that it loads the client library.
3. In a terminal window, go to <*PB_HOME*>`/tools/server`.
4. Start the PointBase server by typing `start_server` on UNIX or `startserver` on Windows.

Populating the Database

1. In a terminal window, go to *<JWSDP_HOME>*/docs/tutorial/examples/web.
2. Execute ant. The default Ant task, create-book-db, uses the PointBase console tool to execute the SQL statements in books.sql. At the end of the processing, you should see the following output:

```
[java] ID
[java] ----------
[java] 201
[java] 202
[java] 203
[java] 204
[java] 205
[java] 206
[java] 207
[java]
[java] 7 Rows Selected.
[java]
[java] SQL>
[java]
[java] COMMIT;
[java] OK
```

Configuring the Web Application to Reference a Data Source

In order to access a database from a Web application, you must declare resource reference in the application's Web application deployment descriptor (see References to Environment Entries, Resource Environment Entries, or Resources, page 101). The resource reference declares a JNDI name, the type of the data resource, and the kind of authentication used when the resource is accessed:

```
<resource-ref>
  <res-ref-name>jdbc/BookDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

The JNDI name is used to create a data source object in the database helper class `database.BookDB` used by the tutorial examples. The `res-auth` element specifies that the container will manage logging on to the database.

Defining a Data Source in Tomcat

In order to use a database you must create a data source in Tomcat. The data source contains information about the driver class and URL used to connect to the database and database login parameters. To define a data source in Tomcat, you use `admintool` (see Configuring Data Sources, page 817) as follows:

1. Start `admintool` by opening a browser at:

   ```
   http://localhost:8080/admin/index.jsp
   ```

2. Log in using the user name and password you specified when you installed the Java WSDP.
3. Select the Data Sources entry under Resources.
4. Select Available Actions→Create New Data Source.
5. Enter `pointbase` in the JNDI Name field.
6. Enter `jdbc:pointbase:server://localhost/sample` in the Data Source URL field.
7. Enter `com.pointbase.jdbc.jdbcUniversalDriver` in the JDBC Driver Class field.
8. Enter `public` in the User Name and Password fields.
9. Click the Save button.
10. Click the Commit button.

Configuring Tomcat to Map the JNDI Name to a Data Source

Since the resource reference declared in the Web application deployment descriptor uses a JNDI name to refer to the data source, you must connect the name to a data source by providing a resource link entry in Tomcat's configuration. Here is the entry used by the application discussed in all the Web technology chapters:

```
<Context path="/bookstore1"
  docBase="../docs/tutorial/examples/web/bookstore1/build"
  debug="0">
  <ResourceLink name="jdbc/BookDB" global="pointbase"/>
</Context>
```

Since the resource link is a subentry of the context entry described in Installing Web Applications (page 102) and Deploying Web Applications (page 103), you add this entry to Tomcat's configuration in the same ways that you add the context entry: by passing the name of a configuration file containing the entry to the `config` attribute of the `Ant install` task or by copying the configuration file named *mywebapp*`.xml` that contains the context entry to <*JWSDP_HOME*>`/webapps`.

If you are deploying the application using the `Ant deploy` task, you must package a configuration file named `context.xml` containing the context entry in the `META-INF` directory of the WAR.

The examples discussed in chapters 14, 15, 16, and 17 show how to deploy applications using the `Ant deploy` task mechanism.

Further Information

For further information on Web applications and Tomcat see:

- The Java Servlet 2.3 Specification, for details on configuring Web applications.
- The reference documentation on Tomcat distributed with the Java WSDP at <*JWSDP_HOME*>`/docs/tomcat/index.html`.

5

Java API for XML Processing

Eric Armstrong

THE Java API for XML Processing (JAXP) is for processing XML data using applications written in the Java programming language. JAXP leverages the parser standards SAX (Simple API for XML Parsing) and DOM (Document Object Model) so that you can choose to parse your data as a stream of events or to build an object representation of it. JAXP also supports the XSLT (XML Stylesheet Language Transformations) standard, giving you control over the presentation of the data and enabling you to convert the data to other XML documents or to other formats, such as HTML. JAXP also provides namespace support, allowing you to work with DTDs that might otherwise have naming conflicts.

Designed to be flexible, JAXP allows you to use any XML-compliant parser from within your application. It does this with what is called a pluggability layer, which allows you to plug in an implementation of the SAX or DOM APIs. The pluggability layer also allows you to plug in an XSL processor, letting you control how your XML data is displayed.

The JAXP APIs

The main JAXP APIs are defined in the `javax.xml.parsers` package. That package contains two vendor-neutral factory classes: `SAXParserFactory` and

`DocumentBuilderFactory` that give you a `SAXParser` and a `DocumentBuilder`, respectively. The `DocumentBuilder`, in turn, creates DOM-compliant `Document` object.

The factory APIs give you the ability to plug in an XML implementation offered by another vendor without changing your source code. The implementation you get depends on the setting of the `javax.xml.parsers.SAXParserFactory` and `javax.xml.parsers.DocumentBuilderFactory` system properties. The default values (unless overridden at runtime) point to the JWSDP implementation.

The remainder of this section shows how the different JAXP APIs work when you write an application.

An Overview of the Packages

The SAX and DOM APIs are defined by XML-DEV group and by the W3C, respectively. The libraries that define those APIs are:

`javax.xml.parsers`
: The JAXP APIs, which provide a common interface for different vendors' SAX and DOM parsers.

`org.w3c.dom`
: Defines the `Document` class (a DOM), as well as classes for all of the components of a DOM.

`org.xml.sax`
: Defines the basic SAX APIs.

`javax.xml.transform`
: Defines the XSLT APIs that let you transform XML into other forms.

The "Simple API" for XML (SAX) is the event-driven, serial-access mechanism that does element-by-element processing. The API for this level reads and writes XML to a data repository or the Web. For server-side and high-performance apps, you will want to fully understand this level. But for many applications, a minimal understanding will suffice.

The DOM API is generally an easier API to use. It provides a relatively familiar tree structure of objects. You can use the DOM API to manipulate the hierarchy of application objects it encapsulates. The DOM API is ideal for interactive applications because the entire object model is present in memory, where it can be accessed and manipulated by the user.

On the other hand, constructing the DOM requires reading the entire XML structure and holding the object tree in memory, so it is much more CPU and memory intensive. For that reason, the SAX API will tend to be preferred for server-side applications and data filters that do not require an in-memory representation of the data.

Finally, the XSLT APIs defined in `javax.xml.transform` let you write XML data to a file or convert it into other forms. And, as you'll see in the XSLT section, of this tutorial, you can even use it in conjunction with the SAX APIs to convert legacy data to XML.

The Simple API for XML (SAX) APIs

The basic outline of the SAX parsing APIs are shown at right. To start the process, an instance of the `SAXParserFactory` class is used to generate an instance of the parser.

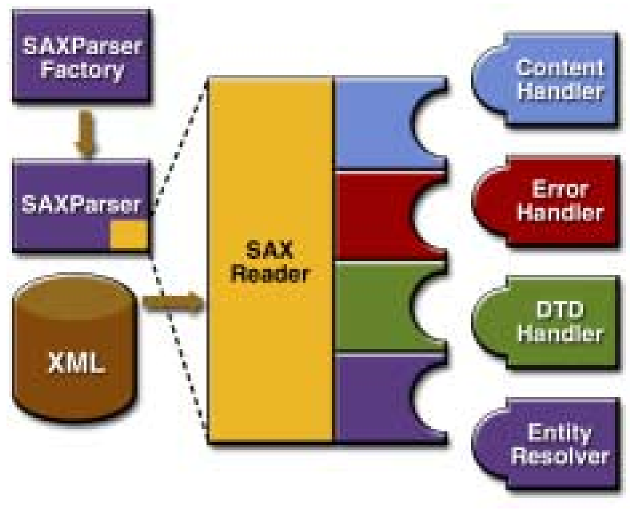


Figure 5–1 SAX APIs

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 the interfaces `ContentHandler`, `ErrorHandler`, `DTDHandler`, and `EntityResolver`.

Here is a summary of the key SAX APIs:

`SAXParserFactory`
: A `SAXParserFactory` object creates an instance of the parser determined by the system property, `javax.xml.parsers.SAXParserFactory`.

`SAXParser`
: The `SAXParser` interface defines several kinds of `parse()` methods. In general, you pass an XML data source and a `DefaultHandler` object to the parser, which processes the XML and invokes the appropriate methods in the handler object.

`SAXReader`
: The `SAXParser` wraps a `SAXReader`. Typically, you don't care about that, but every once in a while you need to get hold of it using `SAXParser`'s `getXMLReader()`, so you can configure it. It is the `SAXReader` which carries on the conversation with the SAX event handlers you define.

`DefaultHandler`
: Not shown in the diagram, a `DefaultHandler` implements the `ContentHandler`, `ErrorHandler`, `DTDHandler`, and `EntityResolver` interfaces (with null methods), so you can override only the ones you're interested in.

`ContentHandler`
: Methods like `startDocument`, `endDocument`, `startElement`, and `endElement` are invoked when an XML tag is recognized. This interface also defines methods `characters` and `processingInstruction`, which are invoked when the parser encounters the text in an XML element or an inline processing instruction, respectively.

`ErrorHandler`
: Methods `error`, `fatalError`, and `warning` are invoked in response to various parsing errors. The default error handler throws an exception for fatal errors and ignores other errors (including validation errors). That's one reason you need to know something about the SAX parser, even if you are using the DOM. Sometimes, the application may be able to recover from a validation error. Other times, it may need to generate an exception. To ensure the correct handling, you'll need to supply your own error handler to the parser.

`DTDHandler`
: Defines methods you will generally never be called upon to use. Used when processing a DTD to recognize and act on declarations for an *unparsed entity.*

`EntityResolver`
: The `resolveEntity` method is invoked when the parser must identify data identified by a URI. In most cases, a URI is simply a URL, which specifies the location of a document, but in some cases the document may be identified by a URN—a *public identifier*, or name, that is unique in the Web space.

The public identifier may be specified in addition to the URL. The `EntityResolver` can then use the public identifier instead of the URL to find the document, for example to access a local copy of the document if one exists.

A typical application implements most of the `ContentHandler` methods, at a minimum. Since the default implementations of the interfaces ignore all inputs except for fatal errors, a robust implementation may want to implement the `ErrorHandler` methods, as well.

The SAX Packages

The SAX parser is defined in the following packages listed in Table 5–1.

Table 5–1 SAX Packagess

Package	**Description**
`org.xml.sax`	Defines the SAX interfaces. The name `org.xml` is the package prefix that was settled on by the group that defined the SAX API.
`org.xml.sax.ext`	Defines SAX extensions that are used when doing more sophisticated SAX processing, for example, to process a document type definitions (DTD) or to see the detailed syntax for a file.
`org.xml.sax.helpers`	Contains helper classes that make it easier to use SAX—for example, by defining a default handler that has null-methods for all of the interfaces, so you only need to override the ones you actually want to implement.
`javax.xml.parsers`	Defines the `SAXParserFactory` class which returns the `SAXParser`. Also defines exception classes for reporting errors.

The Document Object Model (DOM) APIs

Figure 5–2 shows the JAXP APIs in action:

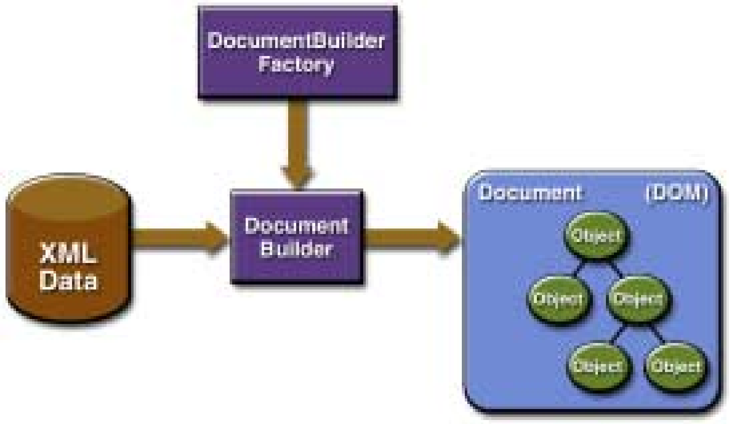


Figure 5–2 DOM APIs

You use the `javax.xml.parsers.DocumentBuilderFactory` class to get a DocumentBuilder instance, and use that to produce a `Document` (a DOM) that conforms to the DOM specification. The builder you get, in fact, is determined by the System property, `javax.xml.parsers.DocumentBuilderFactory`, which selects the factory implementation that is used to produce the builder. (The platform's default value can be overridden from the command line.)

You can also use the `DocumentBuilder newDocument()` method to create an empty `Document` that implements the `org.w3c.dom.Document` interface. Alternatively, you can use one of the builder's parse methods to create a `Document` from existing XML data. The result is a DOM tree like that shown in the diagram.

Note: Although they are called objects, the entries in the DOM tree are actually fairly low-level data structures. For example, under every *element node* (which corresponds to an XML element) there is a *text node* which contains the name of the element tag! This issue will be explored at length in the DOM section of the tutorial, but users who are expecting objects are usually surprised to find that invoking the `text()` method on an element object returns nothing! For a truly object-oriented tree, see the JDOM API at `http://www.jdom.org`.

The DOM Packages

The Document Object Model implementation is defined in the packages listed in Table 5–2.:

Table 5–2 DOM Packages

Package	**Description**
`org.w3c.dom`	Defines the DOM programming interfaces for XML (and, optionally, HTML) documents, as specified by the W3C.
`javax.xml.parsers`	Defines the `DocumentBuilderFactory` class and the `DocumentBuilder` class, which returns an object that implements the W3C Document interface. The factory that is used to create the builder is determined by the `javax.xml.parsers` system property, which can be set from the command line or overridden when invoking the `new Instance` method. This package also defines the `ParserConfigurationException` class for reporting errors.

The XML Stylesheet Language for Transformation (XSLT) APIs

Figure 5–3 shows the XSLT APIs in action.

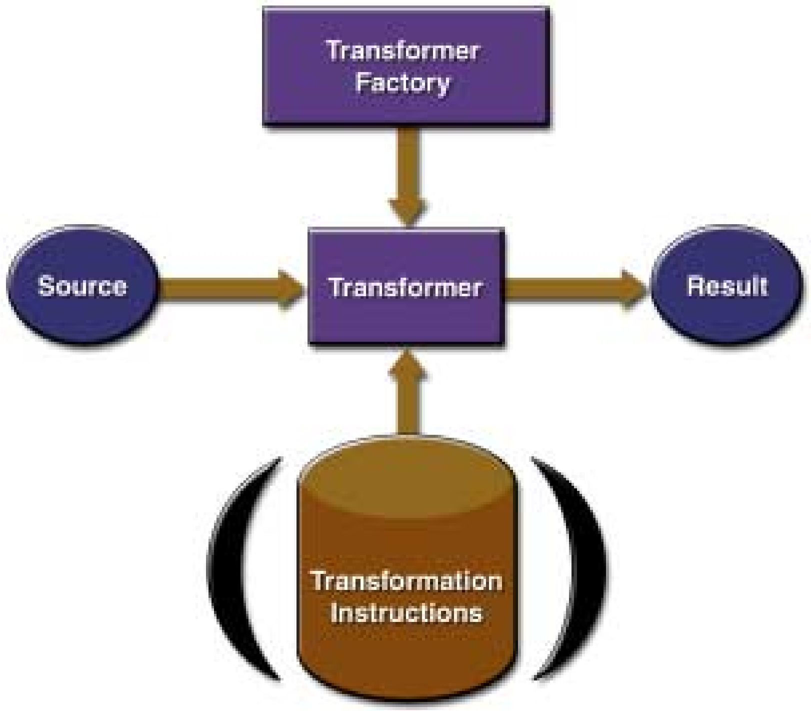


Figure 5–3 XSLT APIs

A `TransformerFactory` object is instantiated, and used to create a `Transformer`. The source object is the input to the transformation process. A source object can be created from SAX reader, from a DOM, or from an input stream.

Similarly, the result object is the result of the transformation process. That object can be a SAX event handler, a DOM, or an output stream.

When the transformer is created, it may be created from a set of transformation instructions, in which case the specified transformations are carried out. If it is created without any specific instructions, then the transformer object simply copies the source to the result.

The XSLT Packages

The XSLT APIs are defined in the following packages:

Table 5–3 XSLT Packages

Package	Description
`javax.xml.transform`	Defines the `TransformerFactory` and `Transformer` classes, which you use to get a object capable of doing transformations. After creating a transformer object, you invoke its `transform()` method, providing it with an input (source) and output (result).
`javax.xml.transform.dom`	Classes to create input (source) and output (result) objects from a DOM.
`javax.xml.transform.sax`	Classes to create input (source) from a SAX parser and output (result) objects from a SAX event handler.
`javax.xml.transform.stream`	Classes to create input (source) and output (result) objects from an I/O stream.

Compiling and Running the Programs

In the Java WSDP, the JAXP libraries are distributed in the directory *`<JWSDP_HOME>`*`/common/lib`. To compile and run the sample programs, you'll first need to install the JAXP libraries in the appropriate location. (The location depends on which version of the JVM you are using.) See the JAXP release notes at *`<JWSDP_HOME>/`*`docs/jaxp/ReleaseNotes.html` for details.

Where Do You Go from Here?

At this point, you have enough information to begin picking your own way through the JAXP libraries. Your next step from here depends on what you want to accomplish. You might want to go to:

The XML Thread

If you want to learn more about XML, spending as little time as possible on the Java APIs. You will see all of the XML sections in the normal course of the tutorial. Follow this thread if you want to bypass the API programming steps:

- Understanding XML (page 41)
- Writing a Simple XML File (page 127)
- Substituting and Inserting Text (page 163)
- Creating a Document Type Definition (DTD) (page 168)
- Defining Attributes and Entities in the DTD (page 177)
- Referencing Binary Entities (page 184)
- Defining Parameter Entities and Conditional Sections (page 193)

Designing an XML Data Structure (page 63)

If you are creating XML data structures for an application and want some tips on how to proceed.

Simple API for XML (page 125)

If the data structures have already been determined, and you are writing a server application or an XML filter that needs to do the fastest possible processing. This section also takes you step by step through the process of constructing an XML document.

Document Object Model (page 211)

If you need to build an object tree from XML data so you can manipulate it in an application, or convert an in-memory tree of objects to XML. This part of the tutorial ends with a section on namespaces.

XML Stylesheet Language for Transformations (page 289)

If you need to transform XML tags into some other form, if you want to generate XML output, or if you want to convert legacy data structures to XML.

6

Simple API for XML

Eric Armstrong

IN this chapter we focus on the Simple API for XML (SAX), an event-driven, serial-access mechanism for accessing XML documents. This is the protocol that most servlets and network-oriented programs will want to use to transmit and receive XML documents, because it's the fastest and least memory-intensive mechanism that is currently available for dealing with XML documents.

The SAX protocol requires a lot more programming than the Document Object Model (DOM). It's an event-driven model (you provide the callback methods, and the parser invokes them as it reads the XML data), which makes it harder to visualize. Finally, you can't "back up" to an earlier part of the document, or rearrange it, any more than you can back up a serial data stream or rearrange characters you have read from that stream.

For those reasons, developers who are writing a user-oriented application that displays an XML document and possibly modifies it will want to use the DOM mechanism described in the next part of the tutorial, Document Object Model (page 211).

However, even if you plan to build with DOM apps exclusively, there are several important reasons for familiarizing yourself with the SAX model:

- Same Error Handling

  When parsing a document for a DOM, the same kinds of exceptions are generated, so the error handling for JAXP SAX and DOM applications are identical.

- Handling Validation Errors

By default, the specifications require that validation errors (which you'll be learning more about in this part of the tutorial) are ignored. If you want to throw an exception in the event of a validation error (and you probably do) then you need to understand how the SAX error handling works.

- Converting Existing Data

  As you'll see in the DOM section of the tutorial, there is a mechanism you can use to convert an existing data set to XML—however, taking advantage of that mechanism requires an understanding of the SAX model.

Note: The examples in this chapter can be found in `<`*`JWSDP_HOME`*`>/docs/tutorial/examples/jaxp/sax/samples`.

When to Use SAX

When it comes to fast, efficient reading of XML data, SAX is hard to beat. It requires little memory, because it does not construct an internal representation (tree structure) of the XML data. Instead, it simply sends data to the application as it is read — your application can then do whatever it wants to do with the data it sees.

In effect, the SAX API acts like a serial I/O stream. You see the data as it streams in, but you can't go back to an earlier position or leap ahead to a different position. In general, it works well when you simply want to read data and have the application act on it.

It is also helpful to understand the SAX event model when you want to convert existing data to XML. As you'll see in Generating XML from an Arbitrary Data Structure (page 312), the key to the conversion process is modifying an existing application to deliver the appropriate SAX events as it reads the data.

But when you need to modify an XML structure — especially when you need to modify it interactively, an in-memory structure like the Document Object Model (DOM) may make more sense.

However, while DOM provides many powerful capabilities for large-scale documents (like books and articles), it also requires a lot of complex coding. (The details of that process are highlighted in When to Use DOM (page 212).)

For simpler applications, that complexity may well be unnecessary. For faster development and simpler applications, one of the object-oriented XML-programming standards may make the most sense, as described in JDOM and dom4j (page 53).

Writing a Simple XML File

Let's start out by writing up a simple version of the kind of XML data you could use for a slide presentation. In this exercise, you'll use your text editor to create the data in order to become comfortable with the basic format of an XML file. You'll be using this file and extending it in later exercises.

Creating the File

Using a standard text editor, create a file called `slideSample.xml`.

Note: Here is a version of it that already exists: `slideSample01.xml`. (The browsable version is `slideSample01-xml.html`.) You can use this version to compare your work, or just review it as you read this guide.

Writing the Declaration

Next, write the declaration, which identifies the file as an XML document. The declaration starts with the characters "<?", which is the standard XML identifier for a *processing instruction*. (You'll see other processing instructions later on in this tutorial.)

```
<?xml version='1.0' encoding='utf-8'?>
```

This line identifies the document as an XML document that conforms to version 1.0 of the XML specification, and says that it uses the 8-bit Unicode character-encoding scheme. (For information on encoding schemes, see Java Encoding Schemes (page 851).)

Since the document has not been specified as "standalone", the parser assumes that it may contain references to other documents. To see how to specify a document as "standalone", see The XML Prolog (page 44).

Adding a Comment

Comments are ignored by XML parsers. You never see them in fact, unless you activate special settings in the parser. You'll see how to do that later on in the tutorial, when we discuss Handling Lexical Events (page 200). For now, add the text highlighted below to put a comment into the file.

```
<?xml version='1.0' encoding='utf-8'?>

<!-- A SAMPLE set of slides -->
```

Defining the Root Element

After the declaration, every XML file defines exactly one element, known as the root element. Any other elements in the file are contained within that element. Enter the text highlighted below to define the root element for this file, `slideshow`:

```
<?xml version='1.0' encoding='utf-8'?>

<!-- A SAMPLE set of slides -->

<slideshow>

</slideshow>
```

Note: XML element names are case-sensitive. The end-tag must exactly match the start-tag.

Adding Attributes to an Element

A slide presentation has a number of associated data items, none of which require any structure. So it is natural to define them as attributes of the `slide-show` element. Add the text highlighted below to set up some attributes:

```
...
  <slideshow
    title="Sample Slide Show"
    date="Date of publication"
    author="Yours Truly"
    >
  </slideshow>
```

When you create a name for a tag or an attribute, you can use hyphens (“`-`”), underscores (“`_`”), colons (“`:`”), and periods (“`.`”) in addition to characters and numbers. Unlike HTML, values for XML attributes are always in quotation marks, and multiple attributes are never separated by commas.

Note: Colons should be used with care or avoided altogether, because they are used when defining the namespace for an XML document.

Adding Nested Elements

XML allows for hierarchically structured data, which means that an element can contain other elements. Add the text highlighted below to define a slide element and a title element contained within it:

```
<slideshow
  ...
  >

   <!-- TITLE SLIDE -->
  <slide type="all">
    <title>Wake up to WonderWidgets!</title>
  </slide>

</slideshow>
```

Here you have also added a `type` attribute to the slide. The idea of this attribute is that slides could be earmarked for a mostly technical or mostly executive audience with `type="tech"` or `type="exec"`, or identified as suitable for both with `type="all"`.

More importantly, though, this example illustrates the difference between things that are more usefully defined as elements (the `title` element) and things that are more suitable as attributes (the `type` attribute). The visibility heuristic is primarily at work here. The title is something the audience will see. So it is an element. The type, on the other hand, is something that never gets presented, so it is an attribute. Another way to think about that distinction is that an element is a container, like a bottle. The type is a characteristic of the *container* (is it tall or short, wide or narrow). The title is a characteristic of the *contents* (water, milk, or tea). These are not hard and fast rules, of course, but they can help when you design your own XML structures.

Adding HTML-Style Text

Since XML lets you define any tags you want, it makes sense to define a set of tags that look like HTML. The XHTML standard does exactly that, in fact. You'll see more about that towards the end of the SAX tutorial. For now, type the text highlighted below to define a slide with a couple of list item entries that use an HTML-style `<em>` tag for emphasis (usually rendered as italicized text):

```
  ...
  <!-- TITLE SLIDE -->
  <slide type="all">
     <title>Wake up to WonderWidgets!</title>
  </slide>

  <!-- OVERVIEW -->
  <slide type="all">
     <title>Overview</title>
        <item>Why <em>WonderWidgets</em> are great</item>
        <item>Who <em>buys</em> WonderWidgets</item>
  </slide>

</slideshow>
```

We'll see later that defining a *title* element conflicts with the XHTML element that uses the same name. We'll discuss the mechanism that produces the conflict (the DTD) and several possible solutions when we cover Parsing the Parameterized DTD (page 197).

Adding an Empty Element

One major difference between HTML and XML, though, is that all XML must be *well-formed* — which means that every tag must have an ending tag or be an empty tag. You're getting pretty comfortable with ending tags, by now. Add the text highlighted below to define an empty list item element with no contents:

```
  ...
  <!-- OVERVIEW -->
  <slide type="all">
    <title>Overview</title>
    <item>Why <em>WonderWidgets</em> are great</item>
    <item/>
    <item>Who <em>buys</em> WonderWidgets</item>
  </slide>

</slideshow>
```

Note that any element can be empty element. All it takes is ending the tag with "`/>`" instead of "`>`". You could do the same thing by entering `<item></item>`, which is equivalent.

Note: Another factor that makes an XML file *well-formed* is proper nesting. So `<b><i>some_text</i></b>` is well-formed, because the `<i>...</i>` sequence is completely nested within the `<b>..</b>` tag. This sequence, on the other hand, is not well-formed: `<b><i>some_text</b></i>`.

The Finished Product

Here is the completed version of the XML file:

```
<?xml version='1.0' encoding='utf-8'?>

<!--  A SAMPLE set of slides  -->

<slideshow
  title="Sample Slide Show"
  date="Date of publication"
  author="Yours Truly"
  >

  <!-- TITLE SLIDE -->
```

```
      <slide type="all">
         <title>Wake up to WonderWidgets!</title>
      </slide>

      <!-- OVERVIEW -->
      <slide type="all">
         <title>Overview</title>
         <item>Why <em>WonderWidgets</em> are great</item>
         <item/>
         <item>Who <em>buys</em> WonderWidgets</item>
      </slide
   </slideshow>
```

Now that you've created a file to work with, you're ready to write a program to echo it using the SAX parser. You'll do that in the next section.

Echoing an XML File with the SAX Parser

In real life, you are going to have little need to echo an XML file with a SAX parser. Usually, you'll want to process the data in some way in order to do something useful with it. (If you want to echo it, it's easier to build a DOM tree and use that for output.) But echoing an XML structure is a great way to see the SAX parser in action, and it can be useful for debugging.

In this exercise, you'll echo SAX parser events to `System.out`. Consider it the "Hello World" version of an XML-processing program. It shows you how to use the SAX parser to get at the data, and then echoes it to show you what you've got.

Note: The code discussed in this section is in `Echo01.java`. The file it operates on is `slideSample01.xml`. (The browsable version is `slideSample01-xml.html`.)

Creating the Skeleton

Start by creating a file named `Echo.java` and enter the skeleton for the application:

```
public class Echo
{
  public static void main(String argv[])
  {

   }

}
```

Since we're going to run it standalone, we need a main method. And we need command-line arguments so we can tell the application which file to echo.

Importing Classes

Next, add the import statements for the classes the application will use:

```
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;

public class Echo
{
  ...
```

The classes in `java.io`, of course, are needed to do output. The `org.xml.sax` package defines all the interfaces we use for the SAX parser. The `SAXParserFactory` class creates the instance we use. It throws a `ParserConfigurationException` if it is unable to produce a parser that matches the specified configuration of options. (You'll see more about the configuration options later.) The `SAXParser` is what the factory returns for parsing, and the `DefaultHandler` defines the class that will handle the SAX events that the parser generates.

Setting up for I/O

The first order of business is to process the command line argument, get the name of the file to echo, and set up the output stream. Add the text highlighted below to take care of those tasks and do a bit of additional housekeeping:

```
public static void main(String argv[])

{
  if (argv.length != 1) {
    System.err.println("Usage: cmd filename");
    System.exit(1);
  }
  try {
    // Set up output stream
    out = new OutputStreamWriter(System.out, "UTF8");
  }
  catch (Throwable t) {
       t.printStackTrace();
  }
  System.exit(0);
}

static private Writer out;
```

When we create the output stream writer, we are selecting the UTF-8 character encoding. We could also have chosen US-ASCII, or UTF-16, which the Java platform also supports. For more information on these character sets, see Java Encoding Schemes (page 851).

Implementing the ContentHandler Interface

The most important interface for our current purposes is the `ContentHandler` interface. That interface requires a number of methods that the SAX parser invokes in response to different parsing events. The major event handling methods are: `startDocument`, `endDocument`, `startElement`, `endElement`, and `characters`.

The easiest way to implement that interface is to extend the `DefaultHandler` class, defined in the `org.xml.sax.helpers` package. That class provides do-nothing methods for all of the `ContentHandler` events. Enter the code highlighted below to extend that class:

```
public class Echo extends DefaultHandler
{
  ...
}
```

Note: `DefaultHandler` also defines do-nothing methods for the other major events, defined in the `DTDHandler`, `EntityResolver`, and `ErrorHandler` interfaces. You'll learn more about those methods as we go along.

Each of these methods is required by the interface to throw a `SAXException`. An exception thrown here is sent back to the parser, which sends it on to the code that invoked the parser. In the current program, that means it winds up back at the `Throwable` exception handler at the bottom of the `main` method.

When a start tag or end tag is encountered, the name of the tag is passed as a `String` to the `startElement` or `endElement` method, as appropriate. When a start tag is encountered, any attributes it defines are also passed in an `Attributes` list. Characters found within the element are passed as an array of characters, along with the number of characters (`length`) and an offset into the array that points to the first character.

Setting up the Parser

Now (at last) you're ready to set up the parser. Add the text highlighted below to set it up and get it started:

```
public static void main(String argv[])
{
  if (argv.length != 1) {
    System.err.println("Usage: cmd filename");
    System.exit(1);
  }

       // Use an instance of ourselves as the SAX event handler
  DefaultHandler handler = new Echo();

        // Use the default (non-validating) parser
  SAXParserFactory factory = SAXParserFactory.newInstance();
  try {
    // Set up output stream
    out = new OutputStreamWriter(System.out, "UTF8");

     // Parse the input
    SAXParser saxParser = factory.newSAXParser();
    saxParser.parse( new File(argv[0]), handler );

       } catch (Throwable t) {
    t.printStackTrace();
  }
  System.exit(0);
}
```

With these lines of code, you created a `SAXParserFactory` instance, as determined by the setting of the `javax.xml.parsers.SAXParserFactory` system property. You then got a parser from the factory and gave the parser an instance of this class to handle the parsing events, telling it which input file to process.

Note: The `javax.xml.parsers.SAXParser` class is a wrapper that defines a number of convenience methods. It wraps the (somewhat-less friendly) `org.xml.sax.Parser` object. If needed, you can obtain that parser using the `SAXParser`'s `getParser()` method.

For now, you are simply catching any exception that the parser might throw. You'll learn more about error processing in a later section of the tutorial, Handling Errors with the Nonvalidating Parser (page 155).

Writing the Output

The `ContentHandler` methods throw `SAXExceptions` but not `IOExceptions`, which can occur while writing. The `SAXException` can wrap another exception, though, so it makes sense to do the output in a method that takes care of the exception-handling details. Add the code highlighted below to define an `emit` method that does that:

```
static private Writer out;

private void emit(String s)
throws SAXException
{
  try {
     out.write(s);
     out.flush();
  } catch (IOException e) {
     throw new SAXException("I/O error", e);
  }
}
...
```

When emit is called, any I/O error is wrapped in `SAXException` along with a message that identifies it. That exception is then thrown back to the SAX parser. You'll learn more about SAX exceptions later on. For now, keep in mind that `emit` is a small method that handles the string output. (You'll see it called a lot in the code ahead.)

Spacing the Output

Here is another bit of infrastructure we need before doing some real processing. Add the code highlighted below to define a `nl()` method that writes the kind of line-ending character used by the current system:

```
private void emit(String s)
  ...
}

private void nl()
throws SAXException
{
  String lineEnd = System.getProperty("line.separator");
  try {
         out.write(lineEnd);
      } catch (IOException e) {
         throw new SAXException("I/O error", e);
      }
   }
```

Note: Although it seems like a bit of a nuisance, you will be invoking `nl()` many times in the code ahead. Defining it now will simplify the code later on. It also provides a place to indent the output when we get to that section of the tutorial.

Handling Content Events

Finally, let's write some code that actually processes the `ContentHandler` events.

Document Events

Add the code highlighted below to handle the start-document and end-document events:

```
static private Writer out;

public void startDocument()
throws SAXException
{
   emit("<?xml version='1.0' encoding='UTF-8'?>");
   nl();
}

public void endDocument()
throws SAXException
{
   try {
      nl();
      out.flush();
   } catch (IOException e) {
      throw new SAXException("I/O error", e);
   }
}

private void echoText()
...
```

Here, you are echoing an XML declaration when the parser encounters the start of the document. Since you set up the `OutputStreamWriter` using the UTF-8 encoding, you include that specification as part of the declaration.

Note: However, the IO classes don't understand the hyphenated encoding names, so you specified "UTF8" rather than "UTF-8".

At the end of the document, you simply put out a final newline and flush the output stream. Not much going on there.

Element Events

Now for the interesting stuff. Add the code highlighted below to process the start-element and end-element events:

```
public void startElement(String namespaceURI,
          String sName, // simple name
          String qName, // qualified name
          Attributes attrs)
throws SAXException
{
  String eName = sName; // element name
  if ("".equals(eName)) eName = qName; // not namespaceAware
  emit("<"+eName);
  if (attrs != null) {
     for (int i = 0; i < attrs.getLength(); i++) {
       String aName = attrs.getLocalName(i); // Attr name
       if ("".equals(aName)) aName = attrs.getQName(i);
       emit(" ");
       emit(aName+"=\""+attrs.getValue(i)+"\"");
     }
  }
  emit(“>”);
}

public void endElement(String namespaceURI,
          String sName, // simple name
          String qName  // qualified name
          )
throws SAXException
{
    String eName = sName; // element name
    if ("".equals(eName)) eName = qName; // not namespaceAware
    emit("<"+eName+">");
  }

  private void emit(String s)
  ...
```

With this code, you echoed the element tags, including any attributes defined in the start tag. Note that when the `startElement`() method is invoked, the simple name (“local name”) for elements and attributes could turn out to be the empty string, if namespace processing was not enabled. The code handles that case by using the qualified name whenever the simple name is the empty string.

Character Events

To finish handling the content events, you need to handle the characters that the parser delivers to your application.

Parsers are not required to return any particular number of characters at one time. A parser can return anything from a single character at a time up to several thousand, and still be standard-conforming implementation. So, if your application needs to process the characters it sees, it is wise to accumulate the characters in a buffer, and operate on them only when you are sure they have all been found.

Add the line highlighted below to define the text buffer:

```
  public class Echo01 extends DefaultHandler
  {
    StringBuffer textBuffer;

    public static void main(String argv[])
    {

  ...
```

Then add the code highlighted below to accumulate the characters the parser delivers in the buffer:

```
public void endElement(...)
throws SAXException
{
  ...
}

public void characters(char buf[], int offset, int len)
throws SAXException
{
  String s = new String(buf, offset, len);
  if (textBuffer == null) {
    textBuffer = new StringBuffer(s);
  } else {
    textBuffer.append(s);
  }
}

private void emit(String s)
...
```

Next, add this method highlighted below to send the contents of the buffer to the output stream.

```
public void characters(char buf[], int offset, int len)
throws SAXException
{
  ...
}

private void echoText()
throws SAXException
{
  if (textBuffer == null) return;
  String s = ""+textBuffer
  emit(s);
  textBuffer = null;
}

private void emit(String s)
...
```

When this method is called twice in a row (which will happens at times, as we'll see next), the buffer will be null. So in that case, the method simply returns. When the buffer is non-null, however, it's contents are sent to the output stream.

Finally, add the code highlighted below to echo the contents of the buffer whenever an element starts or ends:

```
public void startElement(...)
throws SAXException
{
  echoText();
  String eName = sName; // element name
  ...
}

public void endElement(...)
throws SAXException
{
  echoText();
  String eName = sName; // element name
  ...
}
```

You're done accumulating text when an element ends, of course. So you echo it at that point, which clears the buffer before the next element starts.

But you also want to echo the accumulated text when an element starts! That's necessary for document-style data, which can contain XML elements that are intermixed with text. For example, in this document fragment:

```
<para>This paragraph contains <bold>important</bold>
ideas.</para>
```

The initial text, "This paragraph contains" is terminated by the start of the `<bold>` element. The text, "important" is terminated by the end tag, `</bold>`, and the final text, "ideas.", is terminated by the end tag, `</para>`.

Note: Most of the time, though, the accumulated text will be echoed when an `endElement()` event occurs. When a `startElement()` event occurs after that, the buffer will be empty. The first line in the `echoText()` method checks for that case, and simply returns.

Congratulations! At this point you have written a complete SAX parser application. The next step is to compile and run it.

Note: To be strictly accurate, the character handler should scan the buffer for ampersand characters ('`&'`);and left-angle bracket characters ('<') and replace them with the strings "`&amp;`" or "`&lt;`", as appropriate. You'll find out more about that kind of processing when we discuss entity references in Substituting and Inserting Text (page 163).

Compiling and Running the Program

In the Java WSDP, the JAXP libraries are distributed in the directory *<JWSDP_HOME>*`/common/lib`. To compile the program you created, you'll first need to install the JAXP JAR files in the appropriate location. (The names of the JAR files depend on which version of JAXP you are using, and their location depends of which version of the Java platform you are using. See the Java XML release notes at *<JWSDP_HOME>/*`docs/jaxp/ReleaseNotes.html` for the latest details.)

Note: ***Since JAXP 1.1 is built into version 1.4 of the Java 2 platform, you can also execute the majority of the JAXP tutorial (SAX, DOM, and XSLT) sections, without doing any special installation of the JAR files. However, to make use of the added features in JAXP — XML Schema and the XSLTC compiling translator — you will need to install JAXP 1.2, as described in the release notes.***

For versions 1.2 and 1.3 of the Java 2 platform, you can execute the following commands to compile and run the program:

```
javac -classpath jaxp-jar-files Echo.java
java -cp jaxp-jar-files Echo slideSample.xml
```

Alternatively, you could place the JAR files in the platform extensions directory and use the simpler commands:

```
javac Echo.java
java Echo slideSample.xml
```

For version 1.4 of the Java 2 platform, you must identify the JAR files as newer versions of the "endorsed standards" that are built into the Java 2 platform. To do that, put the JAR files in the endorsed standards directory, `jre/lib/endorsed`. (You copy all of the JAR files, except for `jaxp-api.jar`. You ignore that one because the JAXP APIs are already built into the 1.4 platform.)

You can then compile and run the program with these commands:

```
javac Echo.java
java Echo slideSample.xml
```

Note: You could also elect to set the `java.endorsed.dirs` system property on the command line so that it points to a directory containing the necessary JAR files, using an command-line option like this: `-D`”`java.endorsed.dirs=`*`somePath`*”.

Checking the Output

Here is part of the program’s output, showing some of its weird spacing:

```
...
<slideshow title="Sample Slide Show" date="Date of publication"
author="Yours Truly">


  <slide type="all">
    <title>Wake up to WonderWidgets!</title>
  </slide>
  ...
```

Note: The program’s output is contained in `Echo01-01.txt`. (The browsable version is `Echo01-01.html`.)

Looking at this output, a number of questions arise. Namely, where is the excess vertical whitespace coming from? And why is it that the elements are indented properly, when the code isn’t doing it? We’ll answer those questions in a moment. First, though, there are a few points to note about the output:

- The comment defined at the top of the file

  ```
  <!-- A SAMPLE set of slides -->
  ```

  does not appear in the listing. Comments are ignored, unless you implement a `LexicalHandler`. You’ll see more about that later on in this tutorial.
- Element attributes are listed all together on a single line. If your window isn’t really wide, you won’t see them all.

- The single-tag empty element you defined (`<item/>`) is treated exactly the same as a two-tag empty element (`<item></item>`). It is, for all intents and purposes, identical. (It's just easier to type and consumes less space.)

Identifying the Events

This version of the echo program might be useful for displaying an XML file, but it's not telling you much about what's going on in the parser. The next step is to modify the program so that you see where the spaces and vertical lines are coming from.

Note: The code discussed in this section is in `Echo02.java`. The output it produces is shown in `Echo02-01.txt`. (The browsable version is `Echo02-01.html`)

Make the changes highlighted below to identify the events as they occur:

```
public void startDocument()
throws SAXException
{
  nl();
  nl();
  emit("START DOCUMENT");
  nl();
  emit("<?xml version='1.0' encoding='UTF-8'?>");
  ~~nl();~~
}

public void endDocument()
throws SAXException
{
  nl();
  emit("END DOCUMENT");
  try {
  ...
}

public void startElement(...)
throws SAXException
{
  echoText();
  nl();
  emit("ELEMENT: ");
  String eName = sName; // element name
  if ("".equals(eName)) eName = qName; // not namespaceAware
  emit("<"+eName);
  if (attrs != null) {
    for (int i = 0; i < attrs.getLength(); i++) {
      String aName = attrs.getLocalName(i); // Attr name
      if ("".equals(aName)) aName = attrs.getQName(i);
      ~~emit(" ");~~
      ~~emit(aName+"=\""+attrs.getValue(i)+"\"");~~
      nl();
      emit("   ATTR: ");
      emit(aName);
      emit("\t\"");
      emit(attrs.getValue(i));
      emit("\"");
    }
  }
  if (attrs.getLength() > 0) nl();
  emit(">");
}

public void endElement(...)
throws SAXException
{
  echoText();
  nl();
  emit("END_ELM: ");
  String eName = sName; // element name
  if ("".equals(eName)) eName = qName; // not namespaceAware
  emit("<"+eName+">");
}

...

private void echoText()
throws SAXException
{
  if (textBuffer == null) return;
  nl();
  emit("CHARS: |");
  String s = ""+textBuffer
  emit(s);
  emit("|");
  textBuffer = null;
}
```

Compile and run this version of the program to produce a more informative output listing. The attributes are now shown one per line, which is nice. But, more importantly, output lines like this one:

```
    CHARS: |

 |
```

show that both the indentation space and the newlines that separate the attributes come from the data that the parser passes to the `characters()` method.

Note: The XML specification requires all input line separators to be normalized to a single newline. The newline character is specified as in Java, C, and UNIX systems, but goes by the alias “linefeed” in Windows systems.

Compressing the Output

To make the output more readable, modify the program so that it only outputs characters containing something other than whitespace.

Note: The code discussed in this section is in `Echo03.java`.

Make the changes shown below to suppress output of characters that are all whitespace:

```
public void echoText()
throws SAXException
{
  nl();
  ~~emit("CHARS: |");~~
  emit("CHARS:   ");
  String s = ""+textBuffer;
  if (!s.trim().equals("")) emit(s);
  ~~emit("|");~~
}
```

Next, add the code highlighted below to echo each set of characters delivered by the parser:

```
public void characters(char buf[], int offset, int len)
throws SAXException
{
  if (textBuffer != null) {
    echoText();
    textBuffer = null;
  }
  String s = new String(buf, offset, len);
  ...
}
```

If you run the program now, you will see that you have eliminated the indentation as well, because the indent space is part of the whitespace that precedes the start of an element. Add the code highlighted below to manage the indentation:

```
static private Writer out;

private String indentString = "    "; // Amount to indent
private int indentLevel = 0;

...

public void startElement(...)
throws SAXException
{
  indentLevel++;
  nl();
  emit("ELEMENT: ");
  ...
}

public void endElement(...)
throws SAXException
{
  nl();
  emit("END_ELM: ");
  emit("</"+sName+">");
  indentLevel--;
}
...
private void nl()
throws SAXException
{
  ...
```

```
      try {
        out.write(lineEnd);
        for (int i=0; i < indentLevel; i++)
          out.write(indentString);
      } catch (IOException e) {
      ...
    }
```

This code sets up an indent string, keeps track of the current indent level, and outputs the indent string whenever the `nl` method is called. If you set the indent string to "", the output will be un-indented (Try it. You'll see why it's worth the work to add the indentation.)

You'll be happy to know that you have reached the end of the "mechanical" code you have to add to the Echo program. From here on, you'll be doing things that give you more insight into how the parser works. The steps you've taken so far, though, have given you a lot of insight into how the parser sees the XML data it processes. It's also given you a helpful debugging tool you can use to see what the parser sees.

Inspecting the Output

There is part of the output from this version of the program:

```
ELEMENT: <slideshow
...
>
CHARS:
CHARS:
  ELEMENT: <slide
  ...
  END_ELM: </slide>
CHARS:
CHARS:
```

Note: The complete output is `Echo03-01.txt`. (The browsable version is `Echo03-01.html`)

Note that the `characters` method was invoked twice in a row. Inspecting the source file `slideSample01.xml` shows that there is a comment before the first slide. The first call to `characters` comes before that comment. The second call comes after. (Later on, you'll see how to be notified when the parser encounters a comment, although in most cases you won't need such notifications.)

Note, too, that the `characters` method is invoked after the first slide element, as well as before. When you are thinking in terms of hierarchically structured data, that seems odd. After all, you intended for the `slideshow` element to contain `slide` elements, not text. Later on, you'll see how to restrict the `slideshow` element using a DTD. When you do that, the `characters` method will no longer be invoked.

In the absence of a DTD, though, the parser must assume that any element it sees contains text like that in the first item element of the overview slide:

```
<item>Why <em>WonderWidgets</em> are great</item>
```

Here, the hierarchical structure looks like this:

```
ELEMENT:  <item>
CHARS:    Why
  ELEMENT:  <em>
  CHARS:    WonderWidgets
  END_ELM:  </em>
CHARS:    are great
END_ELM:  </item>
```

Documents and Data

In this example, it's clear that there are characters intermixed with the hierarchical structure of the elements. The fact that text can surround elements (or be prevented from doing so with a DTD or schema) helps to explain why you sometimes hear talk about "XML data" and other times hear about "XML documents". XML comfortably handles both structured data and text documents that include markup. The only difference between the two is whether or not text is allowed between the elements.

Note: In an upcoming section of this tutorial, you will work with the `ignorableWhitespace` method in the `ContentHandler` interface. This method can only be invoked when a DTD is present. If a DTD specifies that `slideshow` does not contain text, then all of the whitespace surrounding the `slide` elements is by definition ignorable. On the other hand, if `slideshow` can contain text (which must be assumed to be true in the absence of a DTD), then the parser must assume that spaces and lines it sees between the `slide` elements are significant parts of the document.

Adding Additional Event Handlers

Besides `ignorableWhitespace`, there are two other `ContentHandler` methods that can find uses in even simple applications: `setDocumentLocator` and `processingInstruction`. In this section of the tutorial, you'll implement those two event handlers.

Identifying the Document's Location

A *locator* is an object that contains the information necessary to find the document. The `Locator` class encapsulates a system ID (URL) or a public identifier (URN), or both. You would need that information if you wanted to find something relative to the current document—in the same way, for example, that an HTML browser processes an `href="anotherFile"` attribute in an anchor tag—the browser uses the location of the current document to find `anotherFile`.

You could also use the locator to print out good diagnostic messages. In addition to the document's location and public identifier, the locator contains methods that give the column and line number of the most recently-processed event. The `setDocumentLocator` method is called only once at the beginning of the parse, though. To get the current line or column number, you would save the locator when `setDocumentLocator` is invoked and then use it in the other event-handling methods.

Note: The code discussed in this section is in `Echo04.java`. Its output is in `Echo04-01.txt`. (The browsable version is `Echo04-01.html`.)

Start by removing the extra character-echoing code you added for the last example:

```
public void characters(char buf[], int offset, int len)
throws SAXException
{
  ~~if (textBuffer != null) {~~
    ~~echoText();~~
    ~~textBuffer = null;~~
```

```
      ~~}~~
      String s = new String(buf, offset, len);
      ...
   }
```

Next. add the method highlighted below to the Echo program to get the document locator and use it to echo the document's system ID.

```
   ...
   private String indentString = "    "; // Amount to indent
   private int indentLevel = 0;

   public void setDocumentLocator(Locator l)
   {
      try {
         out.write("LOCATOR");
         out.write("SYS ID: " + l.getSystemId() );
         out.flush();
      } catch (IOException e) {
         // Ignore errors
      }
   }

   public void startDocument()
   ...
```

Notes:

- This method, in contrast to every other `ContentHandler` method, does not return a `SAXException`. So, rather than using `emit` for output, this code writes directly to `System.out`. (This method is generally expected to simply save the `Locator` for later use, rather than do the kind of processing that generates an exception, as here.)
- The spelling of these methods is "`Id`", not "`ID`". So you have `getSystemId` and `getPublicId`.

When you compile and run the program on `slideSample01.xml`, here is the significant part of the output:

```
   LOCATOR
   SYS ID: file:<path>/../samples/slideSample01.xml

   START DOCUMENT
   <?xml version='1.0' encoding='UTF-8'?>
   ...
```

Here, it is apparent that `setDocumentLocator` is called before startDocument. That can make a difference if you do any initialization in the event handling code.

Handling Processing Instructions

It sometimes makes sense to code application-specific processing instructions in the XML data. In this exercise, you'll add a processing instruction to your `slideSample.xml` file and then modify the `Echo` program to display it.

Note: The code discussed in this section is in `Echo05.java`. The file it operates on is `slideSample02.xml`. The output is in `Echo05-02.txt`. (The browsable versions are `slideSample02-xml.html` and `Echo05-02.html`.)

As you saw in Understanding XML (page 41), the format for a processing instruction is `<?target data?>`, where "target" is the target application that is expected to do the processing, and "data" is the instruction or information for it to process. Add the text highlighted below to add a processing instruction for a mythical slide presentation program that will query the user to find out which slides to display (technical, executive-level, or all):

```
<slideshow
  ...
  >

  <!-- PROCESSING INSTRUCTION -->
  <?my.presentation.Program QUERY="exec, tech, all"?>

  <!-- TITLE SLIDE -->
```

Notes:

- The "data" portion of the processing instruction can contain spaces, or may even be null. But there cannot be any space between the initial `<?` and the target identifier.
- The data begins after the first space.
- Fully qualifying the target with the complete Web-unique package prefix makes sense, so as to preclude any conflict with other programs that might process the same data.

- For readability, it seems like a good idea to include a colon (:) after the name of the application, like this:

  ```
  <?my.presentation.Program: QUERY="..."?>
  ```

  The colon makes the target name into a kind of “label” that identifies the intended recipient of the instruction. However, while the w3c spec allows “:” in a target name, some versions of IE5 consider it an error. For this tutorial, then, we avoid using a colon in the target name.

Now that you have a processing instruction to work with, add the code highlighted below to the Echo app:

```
public void characters(char buf[], int offset, int len)
...
}

public void processingInstruction(String target, String data)
throws SAXException
{
  nl();
  emit("PROCESS: ");
  emit("<?"+target+" "+data+"?>");
}

private void echoText()
...
```

When your edits are complete, compile and run the program. The relevant part of the output should look like this:

```
ELEMENT: <slideshow
  ...
>
PROCESS: <?my.presentation.Program QUERY="exec, tech, all"?>
CHARS:
...
```

Summary

With the minor exception of `ignorableWhitespace`, you have used most of the `ContentHandler` methods that you need to handle the most commonly useful SAX events. You’ll see `ignorableWhitespace` a little later on. Next, though, you’ll get deeper insight into how you handle errors in the SAX parsing process.

Handling Errors with the Nonvalidating Parser

This version of the Echo program uses the nonvalidating parser. So it can't tell if the XML document contains the right tags, or if those tags are in the right sequence. In other words, it can't tell you if the document is valid. It can, however, tell whether or not the document is well-formed.

In this section of the tutorial, you'll modify the slideshow file to generate different kinds of errors and see how the parser handles them. You'll also find out which error conditions are ignored, by default, and see how to handle them.

Introducing an Error

The parser can generate one of three kinds of errors: fatal error, error, and warning. In this exercise, you'll make a simple modification to the XML file to introduce a fatal error. Then you'll see how it's handled in the Echo app.

Note: The XML structure you'll create in this exercise is in `slideSampleBad1.xml`. The output is in `Echo05-Bad1.txt`. (The browsable versions are `slideSampleBad1-xml.html` and `Echo05-Bad1.html`.)

One easy way to introduce a fatal error is to remove the final "`/`" from the empty `item` element to create a tag that does not have a corresponding end tag. That constitutes a fatal error, because all XML documents must, by definition, be well formed. Do the following:

1. Copy `slideSample.xml` to `badSample.xml`.
2. Edit `badSample.xml` and remove the character shown below:

```
...
<!-- OVERVIEW -->
<slide type="all">
  <title>Overview</title>
  <item>Why <em>WonderWidgets</em> are great</item>
  <item~~/~~>
  <item>Who <em>buys</em> WonderWidgets</item>
</slide>
...
```

to produce:

```
...
<item>Why <em>WonderWidgets</em> are great</item>
<item>
<item>Who <em>buys</em> WonderWidgets</item>
...
```

3. Run the Echo program on the new file.

The output now gives you an error message that looks like this (after formatting for readability):

```
org.xml.sax.SAXParseException:
  The element type "item" must be terminated by the
  matching end-tag “</item>”.
...
at org.apache.xerces.parsers.AbstractSAXParser...
...
at Echo.main(...)
```

Note: The message above was generated by the JAXP 1.2 libraries. If you are using a different parser, the error message is likely to be somewhat different.

When a fatal error occurs, the parser is unable to continue. So, if the application does not generate an exception (which you'll see how to do a moment), then the default error-event handler generates one. The stack trace is generated by the `Throwable` exception handler in your main method:

```
    ...
} catch (Throwable t) {
  t.printStackTrace();
}
```

That stack trace is not too useful, though. Next, you'll see how to generate better diagnostics when an error occurs.

Handling a SAXParseException

When the error was encountered, the parser generated a `SAXParseException`—a subclass of `SAXException` that identifies the file and location where the error occurred.

Note: The code you'll create in this exercise is in `Echo06.java`. The output is in `Echo06-Bad1.txt`. (The browsable version is `Echo06-Bad1.html`.)

Add the code highlighted below to generate a better diagnostic message when the exception occurs:

```
...
} catch (SAXParseException spe) {
  // Error generated by the parser
  System.out.println("\n** Parsing error"
     + ", line " + spe.getLineNumber()
     + ", uri " + spe.getSystemId());
  System.out.println("   " + spe.getMessage() );

} catch (Throwable t) {
  t.printStackTrace();
}
```

Running the program now generates an error message which is a bit more helpful, like this:

```
** Parsing error, line 22, uri file:<path>/slideSampleBad1.xml
  The element type "item" must be ...
```

Note: The text of the error message depends on the parser used. This message was generated using JAXP 1.2.

Note: Catching all throwables like this is not generally a great idea for production applications. We're doing it now so we can build up to full error handling gradually. In addition, it acts as a catch-all for null pointer exceptions that can be thrown when the parser is passed a null value.

Handling a SAXException

A more general `SAXException` instance may sometimes be generated by the parser, but it more frequently occurs when an error originates in one of application's event handling methods. For example, the signature of the `startDocument` method in the ContentHandler interface is defined as returning a SAXException:

```
public void startDocument() throws SAXException
```

All of the ContentHandler methods (except for setDocumentLocator) have that signature declaration.

A SAXException can be constructed using a message, another exception, or both. So, for example, when Echo.startDocument outputs a string using the emit method, any I/O exception that occurs is wrapped in a SAXException and sent back to the parser:

```
private void emit(String s)
throws SAXException
{
  try {
    out.write(s);
    out.flush();
  } catch (IOException e) {
    throw new SAXException("I/O error", e);
  }
}
```

Note: If you saved the Locator object when setDocumentLocator was invoked, you could use it to generate a SAXParseException, identifying the document and location, instead of generating a SAXException.

When the parser delivers the exception back to the code that invoked the parser, it makes sense to use the original exception to generate the stack trace. Add the code highlighted below to do that:

```
   ...
} catch (SAXParseException err) {
  System.out.println("\n** Parsing error"
    + ", line " + err.getLineNumber()
    + ", uri " + err.getSystemId());
  System.out.println("   " + err.getMessage());

} catch (SAXException sxe) {
  // Error generated by this application
  // (or a parser-initialization error)
  Exception  x = sxe;
  if (sxe.getException() != null)
```

```
      x = sxe.getException();
    x.printStackTrace();

  } catch (Throwable t) {
    t.printStackTrace();
  }
```

This code tests to see if the `SAXException` is wrapping another exception. If so, it generates a stack trace originating from where that exception occurred to make it easier to pinpoint the code responsible for the error. If the exception contains only a message, the code prints the stack trace starting from the location where the exception was generated.

Improving the SAXParseException Handler

Since the `SAXParseException` can also wrap another exception, add the code highlighted below to use the contained exception for the stack trace:

```
      ...
  } catch (SAXParseException err) {
    System.out.println("\n** Parsing error"
      + ", line " + err.getLineNumber()
      + ", uri " + err.getSystemId());
    System.out.println("   " + err.getMessage());

    // Use the contained exception, if any
    Exception  x = spe;
    if (spe.getException() != null)
      x = spe.getException();
    x.printStackTrace();

  } catch (SAXException sxe) {
    // Error generated by this application
    // (or a parser-initialization error)
    Exceptionx = sxe;
    if (sxe.getException() != null)
      x = sxe.getException();
    x.printStackTrace();

  } catch (Throwable t) {
    t.printStackTrace();
  }
```

The program is now ready to handle any SAX parsing exceptions it sees. You've seen that the parser generates exceptions for fatal errors. But for nonfatal errors and warnings, exceptions are never generated by the default error handler, and no messages are displayed. In a moment, you'll learn more about errors and warnings and find out how to supply an error handler to process them.

Handling a ParserConfigurationException

Finally, recall that the `SAXParserFactory` class could throw an exception if it were for unable to create a parser. Such an error might occur if the factory could not find the class needed to create the parser (class not found error), was not permitted to access it (illegal access exception), or could not instantiate it (instantiation error).

Add the code highlighted below to handle such errors:

```
} catch (SAXException sxe) {
   Exceptionx = sxe;
   if (sxe.getException() != null)
      x = sxe.getException();
   x.printStackTrace();

} catch (ParserConfigurationException pce) {
   // Parser with specified options can't be built
   pce.printStackTrace();

} catch (Throwable t) {
   t.printStackTrace();
```

Admittedly, there are quite a few error handlers here. But at least now you know the kinds of exceptions that can occur.

Note: A `javax.xml.parsers.FactoryConfigurationError` could also be thrown if the factory class specified by the system property cannot be found or instantiated. That is a non-trappable error, since the program is not expected to be able to recover from it.

Handling an IOException

Finally, while we're at it, let's add a handler for `IOExceptions`:

```
} catch (ParserConfigurationException pce) {
  // Parser with specified options can't be built
  pce.printStackTrace();

} catch (IOException ioe) {
  // I/O error
  ioe.printStackTrace();
}

} catch (Throwable t) {
  ...
```

We'll leave the handler for `Throwables` to catch null pointer errors, but note that at this point it is doing the same thing as the `IOException` handler. Here, we're merely illustrating the kinds of exceptions that *can* occur, in case there are some that your application could recover from.

Handling NonFatal Errors

A *nonfatal* error occurs when an XML document fails a validity constraint. If the parser finds that the document is not valid, then an error event is generated. Such errors are generated by a validating parser, given a DTD or schema, when a document has an invalid tag, or a tag is found where it is not allowed, or (in the case of a schema) if the element contains invalid data.

You won't actually dealing with validation issues until later in this tutorial. But since we're on the subject of error handling, you'll write the error-handling code now.

The most important principle to understand about non-fatal errors is that they are *ignored*, by default.

But if a validation error occurs in a document, you probably don't want to continue processing it. You probably want to treat such errors as fatal. In the code you write next, you'll set up the error handler to do just that.

Note: The code for the program you'll create in this exercise is in `Echo07.java`.

To take over error handling, you override the `DefaultHandler` methods that handle fatal errors, nonfatal errors, and warnings as part of the `ErrorHandler` interface. The SAX parser delivers a `SAXParseException` to each of these methods, so generating an exception when an error occurs is as simple as throwing it back.

Add the code highlighted below to override the handler for errors:

```
public void processingInstruction(String target, String data)
throws SAXException
{
  ...
}

// treat validation errors as fatal
public void error(SAXParseException e)
throws SAXParseException
{
  throw e;
}
```

Note: It can be instructive to examine the error-handling methods defined in `org.xml.sax.helpers.DefaultHandler`. You'll see that the `error()` and `warning()` methods do nothing, while `fatalError()` throws an exception. Of course, you could always override the `fatalError()` method to throw a different exception. But if your code *doesn't* throw an exception when a fatal error occurs, then the SAX parser will — the XML specification requires it.

Handling Warnings

Warnings, too, are ignored by default. Warnings are informative, and require a DTD. For example, if an element is defined twice in a DTD, a warning is generated—it's not illegal, and it doesn't cause problems, but it's something you might like to know about since it might not have been intentional.

Add the code highlighted below to generate a message when a warning occurs:

```
// treat validation errors as fatal
public void error(SAXParseException e)
throws SAXParseException
{
  throw e;
}

// dump warnings too
public void warning(SAXParseException err)
throws SAXParseException
{
  System.out.println("** Warning"
    + ", line " + err.getLineNumber()
    + ", uri " + err.getSystemId());
  System.out.println("   " + err.getMessage());
}
```

Since there is no good way to generate a warning without a DTD or schema, you won't be seeing any just yet. But when one does occur, you're ready!

Substituting and Inserting Text

The next thing we want to do with the parser is to customize it a bit, so you can see how to get information it usually ignores. But before we can do that, you're going to need to learn a few more important XML concepts. In this section, you'll learn about:

- Handling Special Characters ("<", "&", and so on)
- Handling Text with XML-style syntax

Handling Special Characters

In XML, an entity is an XML structure (or plain text) that has a name. Referencing the entity by name causes it to be inserted into the document in place of the entity reference. To create an entity reference, the entity name is surrounded by an ampersand and a semicolon, like this:

```
&entityName;
```

Later, when you learn how to write a DTD, you'll see that you can define your own entities, so that `&yourEntityName;` expands to all the text you defined for that entity. For now, though, we'll focus on the predefined entities and character references that don't require any special definitions.

Predefined Entities

An entity reference like `&amp;` contains a name (in this case, "amp") between the start and end delimiters. The text it refers to (&) is substituted for the name, like a macro in a C or C++ program. Table 6–1 shows the predefined entities for special characters.

Table 6–1 Predefined Entities

Character	Reference
&	&amp;
<	&lt;
>	&gt;
"	&quot;
'	&apos;

Character References

A character reference like `&#147;` contains a hash mark (`#`) followed by a number. The number is the Unicode value for a single character, such as 65 for the letter "A", 147 for the left-curly quote, or 148 for the right-curly quote. In this case, the "name" of the entity is the hash mark followed by the digits that identify the character.

Note: XML expects values to be specified in decimal. However, the Unicode charts at http://www.unicode.org/charts/ specify values in hexadecimal! So you'll need to do a conversion to get the right value to insert into your XML data set.

Using an Entity Reference in an XML Document

Suppose you wanted to insert a line like this in your XML document:

```
Market Size < predicted
```

The problem with putting that line into an XML file directly is that when the parser sees the left-angle bracket (<), it starts looking for a tag name, which throws off the parse. To get around that problem, you put `&lt;` in the file, instead of "<".

Note: The results of the modifications below are contained in `slideSample03.xml`. The results of processing it are shown in `Echo07-03.txt`. (The browsable versions are `slideSample03-xml.html` and `Echo07-03.html`.)

If you are following the programming tutorial, add the text highlighted below to your `slideSample.xml` file:

```
  <!-- OVERVIEW -->
  <slide type="all">
    <title>Overview</title>
    ...
  </slide>

  <slide type="exec">
    <title>Financial Forecast</title>
    <item>Market Size &lt; predicted</item>
    <item>Anticipated Penetration</item>
    <item>Expected Revenues</item>
    <item>Profit Margin </item>
  </slide>

</slideshow>
```

When you run the Echo program on your XML file, you see the following output:

```
ELEMENT:  <item>
CHARS:    Market Size < predicted
END_ELM:  </item>
```

The parser converted the reference into the entity it represents, and passed the entity to the application.

Handling Text with XML-Style Syntax

When you are handling large blocks of XML or HTML that include many of the special characters, it would be inconvenient to replace each of them with the appropriate entity reference. For those situations, you can use a CDATA section.

Note: The results of the modifications below are contained in slideSample04.xml. The results of processing it are shown in Echo07-04.txt. (The browsable versions are slideSample04-xml.html and Echo07-04.html.)

A CDATA section works like `<pre>...</pre>` in HTML, only more so—all whitespace in a CDATA section is significant, and characters in it are not interpreted as XML. A CDATA section starts with `<![CDATA[` and ends with `]]>`. Add the text highlighted below to your slideSample.xml file to define a CDATA section for a fictitious technical slide:

```
     ...
    <slide type="tech">
      <title>How it Works</title>
      <item>First we fozzle the frobmorten</item>
      <item>Then we framboze the staten</item>
      <item>Finally, we frenzle the fuznaten</item>
      <item><![CDATA[Diagram:
        frobmorten <--------------- fuznaten
          |   <3>^
          | <1>|  <1> = fozzle
          V    |  <2> = framboze
          Staten--------------------+<3> = frenzle
             <2>
      ]]></item>
    </slide>
  </slideshow>
```

When you run the Echo program on the new file, you see the following output:

```
    ELEMENT: <item>
    CHARS:   Diagram:

  frobmorten <--------------fuznaten
       |       <3>    ^
```

```
        | <1>           |   <1> = fozzle
       V                |   <2> = framboze
     staten-----------------------+   <3> = frenzle
               <2>

   END_ELM: </item>
```

You can see here that the text in the `CDATA` section arrived as it was written. Since the parser didn't treat the angle brackets as XML, they didn't generate the fatal errors they would otherwise cause. (Because, if the angle brackets weren't in a CDATA section, the document would not be well-formed.)

Handling CDATA and Other Characters

The existence of CDATA makes the proper echoing of XML a bit tricky. If the text to be output is *not* in a CDATA section, then any angle brackets, ampersands, and other special characters in the text should be replaced with the appropriate entity reference. (Replacing left angle brackets and ampersands is most important, other characters will be interpreted properly without misleading the parser.)

But if the output text *is* in a CDATA section, then the substitutions should not occur, to produce text like that in the example above. In a simple program like our Echo application, it's not a big deal. But many XML-filtering applications will want to keep track of whether the text appears in a CDATA section, in order to treat special characters properly.

One other area to watch for is attributes. The text of an attribute value could also contain angle brackets and semicolons that need to be replaced by entity references. (Attribute text can never be in a CDATA section, though, so there is never any question about doing that substitution.)

Later in this tutorial, you will see how to use a `LexicalHandler` to find out whether or not you are processing a CDATA section. Next, though, you will see how to define a DTD.

Creating a Document Type Definition (DTD)

After the XML declaration, the document prolog can include a DTD, which lets you specify the kinds of tags that can be included in your XML document. In addition to telling a validating parser which tags are valid, and in what arrangements, a DTD tells both validating and nonvalidating parsers where text is expected, which lets the parser determine whether the whitespace it sees is significant or ignorable.

Basic DTD Definitions

When you were parsing the slide show, for example, you saw that the `characters` method was invoked multiple times before and after comments and slide elements. In those cases, the whitespace consisted of the line endings and indentation surrounding the markup. The goal was to make the XML document readable—the whitespace was not in any way part of the document contents. To begin learning about DTD definitions, let's start by telling the parser where whitespace is ignorable.

Note: The DTD defined in this section is contained in `slideshow1a.dtd`. (The browsable version is `slideshow1a-dtd.html`.)

Start by creating a file named `slideshow.dtd`. Enter an XML declaration and a comment to identify the file, as shown below:

```
<?xml version='1.0' encoding='utf-8'?>

<!--
  DTD for a simple "slide show".
-->
```

Next, add the text highlighted below to specify that a `slideshow` element contains `slide` elements and nothing else:

```
<!-- DTD for a simple "slide show". -->

<!ELEMENT slideshow (slide+)>
```

As you can see, the DTD tag starts with `<!` followed by the tag name (`ELEMENT`). After the tag name comes the name of the element that is being defined (`slideshow`) and, in parentheses, one or more items that indicate the valid contents for that element. In this case, the notation says that a `slideshow` consists of one or more `slide` elements.

Without the plus sign, the definition would be saying that a `slideshow` consists of a single `slide` element. The qualifiers you can add to an element definition are listed in Table 6–2.

Table 6–2 DTD Element Qualifiers

Qualifier	Name	Meaning
?	Question Mark	Optional (zero or one)
*	Asterisk	Zero or more
+	Plus Sign	One or more

You can include multiple elements inside the parentheses in a comma separated list, and use a qualifier on each element to indicate how many instances of that element may occur. The comma-separated list tells which elements are valid and the order they can occur in.

You can also nest parentheses to group multiple items. For an example, after defining an `image` element (coming up shortly), you could declare that every `image` element must be paired with a `title` element in a slide by specifying `((image, title)+)`. Here, the plus sign applies to the `image/title` pair to indicate that one or more pairs of the specified items can occur.

Defining Text and Nested Elements

Now that you have told the parser something about where *not* to expect text, let's see how to tell it where text *can* occur. Add the text highlighted below to define the `slide`, `title`, `item`, and `list` elements:

```
<!ELEMENT slideshow (slide+)>
<!ELEMENT slide (title, item*)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT item (#PCDATA | item)* >
```

The first line you added says that a slide consists of a `title` followed by zero or more `item` elements. Nothing new there. The next line says that a title consists entirely of *parsed character data* (`PCDATA`). That's known as "text" in most parts of the country, but in XML-speak it's called "parsed character data". (That distinguishes it from `CDATA` sections, which contain character data that is not parsed.) The `"#"` that precedes `PCDATA` indicates that what follows is a special word, rather than an element name.

The last line introduces the vertical bar (`|`), which indicates an *or* condition. In this case, either `PCDATA` or an `item` can occur. The asterisk at the end says that either one can occur zero or more times in succession. The result of this specification is known as a *mixed-content model*, because any number of `item` elements can be interspersed with the text. Such models must always be defined with `#PCDATA` specified first, some number of alternate items divided by vertical bars (`|`), and an asterisk (`*`) at the end.

Limitations of DTDs

It would be nice if we could specify that an `item` contains either text, or text followed by one or more list items. But that kind of specification turns out to be hard to achieve in a DTD. For example, you might be tempted to define an `item` like this:

```
<!ELEMENT item (#PCDATA | (#PCDATA, item+)) >
```

That would certainly be accurate, but as soon as the parser sees #PCDATA and the vertical bar, it requires the remaining definition to conform to the mixed-content model. This specification doesn't, so you get can error that says: `Illegal mixed content model for 'item'. Found &#x28; ...`, where the hex character 28 is the angle bracket the ends the definition.

Trying to double-define the item element doesn't work, either. A specification like this:

```
<!ELEMENT item (#PCDATA) >
<!ELEMENT item (#PCDATA, item+) >
```

produces a "duplicate definition" warning when the validating parser runs. The second definition is, in fact, ignored. So it seems that defining a mixed content model (which allows `item` elements to be interspersed in text) is about as good as we can do.

In addition to the limitations of the mixed content model mentioned above, there is no way to further qualify the kind of text that can occur where `PCDATA` has been specified. Should it contain only numbers? Should be in a date format, or possibly a monetary format? There is no way to say in the context of a DTD.

Finally, note that the DTD offers no sense of hierarchy. The definition for the `title` element applies equally to a `slide` title and to an `item` title. When we expand the DTD to allow HTML-style markup in addition to plain text, it would make sense to restrict the size of an `item` title compared to a `slide` title, for example. But the only way to do that would be to give one of them a different name, such as "`item-title`". The bottom line is that the lack of hierarchy in the DTD forces you to introduce a "hyphenation hierarchy" (or its equivalent) in your namespace. All of these limitations are fundamental motivations behind the development of schema-specification standards.

Special Element Values in the DTD

Rather than specifying a parenthesized list of elements, the element definition could use one of two special values: `ANY` or `EMPTY`. The `ANY` specification says that the element may contain any other defined element, or `PCDATA`. Such a specification is usually used for the root element of a general-purpose XML document such as you might create with a word processor. Textual elements could occur in any order in such a document, so specifying `ANY` makes sense.

The `EMPTY` specification says that the element contains no contents. So the DTD for e-mail messages that let you "flag" the message with `<flag/>` might have a line like this in the DTD:

```
<!ELEMENT flag EMPTY>
```

Referencing the DTD

In this case, the DTD definition is in a separate file from the XML document. That means you have to reference it from the XML document, which makes the DTD file part of the external subset of the full Document Type Definition (DTD) for the XML file. As you'll see later on, you can also include parts of the DTD within the document. Such definitions constitute the local subset of the DTD.

Note: The XML written in this section is contained in `slideSample05.xml`. (The browsable version is `slideSample05-xml.html`.)

To reference the DTD file you just created, add the line highlighted below to your `slideSample.xml` file:

```
<!--  A SAMPLE set of slides  -->

<!DOCTYPE slideshow SYSTEM "slideshow.dtd">

<slideshow
```

Again, the DTD tag starts with `"<!"`. In this case, the tag name, `DOCTYPE`, says that the document is a `slideshow`, which means that the document consists of the `slideshow` element and everything within it:

```
<slideshow>
...
</slideshow>
```

This tag defines the `slideshow` element as the root element for the document. An XML document must have exactly one root element. This is where that element is specified. In other words, this tag identifies the document *content* as a `slideshow`.

The `DOCTYPE` tag occurs after the XML declaration and before the root element. The `SYSTEM` identifier specifies the location of the DTD file. Since it does not start with a prefix like `http:/` or `file:/`, the path is relative to the location of the XML document. Remember the `setDocumentLocator` method? The parser is using that information to find the DTD file, just as your application would to find a file relative to the XML document. A `PUBLIC` identifier could also be used to specify the DTD file using a unique name—but the parser would have to be able to resolve it

The `DOCTYPE` specification could also contain DTD definitions within the XML document, rather than referring to an external DTD file. Such definitions would be contained in square brackets, like this:

```
<!DOCTYPE slideshow SYSTEM "slideshow1.dtd" [
  ...local subset definitions here...
]>
```

You'll take advantage of that facility later on to define some entities that can be used in the document.

DTD's Effect on the Nonvalidating Parser

In the last section, you defined a rudimentary document type and used it in your XML file. In this section, you'll use the Echo program to see how the data appears to the SAX parser when the DTD is included.

Note: The output shown in this section is contained in `Echo07-05.txt`. (The browsable version is `Echo07-05.html`.)

Running the Echo program on your latest version of `slideSample.xml` shows that many of the superfluous calls to the `characters` method have now disappeared.

Where before you saw:

```
  ...
>
PROCESS: ...
CHARS:
  ELEMENT:  <slide
    ATTR: ...
  >
      ELEMENT:  <title>
      CHARS:    Wake up to ...
      END_ELM:  </title>
  END_ELM:  </slide>
CHARS:
  ELEMENT:  <slide
    ATTR: ...
  >
  ...
```

Now you see:

```
  ...
>
PROCESS: ...
  ELEMENT:  <slide
  ATTR: ...
>
    ELEMENT:  <title>
    CHARS:    Wake up to ...
    END_ELM:  </title>
END_ELM:  </slide>
ELEMENT:  <slide
  ATTR: ...
>
...
```

It is evident here that the whitespace characters which were formerly being echoed around the `slide` elements are no longer being delivered by the parser, because the DTD declares that `slideshow` consists solely of `slide` elements:

```
<!ELEMENT slideshow (slide+)>
```

Tracking Ignorable Whitespace

Now that the DTD is present, the parser is no longer calling the `characters` method with whitespace that it knows to be irrelevant. From the standpoint of an application that is only interested in processing the XML data, that is great. The application is never bothered with whitespace that exists purely to make the XML file readable.

On the other hand, if you were writing an application that was filtering an XML data file, and you wanted to output an equally readable version of the file, then that whitespace would no longer be irrelevant—it would be essential. To get those characters, you need to add the `ignorableWhitespace` method to your application. You'll do that next.

Note: The code written in this section is contained in `Echo08.java`. The output is in `Echo08-05.txt`. (The browsable version is `Echo08-05.html`.)

To process the (generally) ignorable whitespace that the parser is seeing, add the code highlighted below to implement the `ignorableWhitespace` event handler in your version of the Echo program:

```
public void characters (char buf[], int offset, int len)
...
}

public void ignorableWhitespace char buf[], int offset, int Len)
throws SAXException
{
  nl();
  emit("IGNORABLE");
}

public void processingInstruction(String target, String data)
...
```

This code simply generates a message to let you know that ignorable whitespace was seen.

Note: Again, not all parsers are created equal. The SAX specification does not require this method to be invoked. The Java XML implementation does so whenever the DTD makes it possible.

When you run the Echo application now, your output looks like this:

```
ELEMENT: <slideshow
  ATTR: ...
>
IGNORABLE
IGNORABLE
PROCESS: ...
IGNORABLE
IGNORABLE
  ELEMENT: <slide
     ATTR: ...
  >
  IGNORABLE
     ELEMENT: <title>
     CHARS:   Wake up to ...
     END_ELM: </title>
  IGNORABLE
  END_ELM: </slide>
IGNORABLE
```

```
IGNORABLE
  ELEMENT: <slide
    ATTR: ...
  >
  ...
```

Here, it is apparent that the ignorableWhitespace is being invoked before and after comments and slide elements, where characters was being invoked before there was a DTD.

Cleanup

Now that you have seen ignorable whitespace echoed, remove that code from your version of the Echo program—you won't be needing it any more in the exercises ahead.

Note: That change has been made in `Echo09.java`.

Documents and Data

Earlier, you learned that one reason you hear about XML *documents*, on the one hand, and XML *data*, on the other, is that XML handles both comfortably, depending on whether text is or is not allowed between elements in the structure.

In the sample file you have been working with, the `slideshow` element is an example of a *data element*—it contains only subelements with no intervening text. The `item` element, on the other hand, might be termed a *document element*, because it is defined to include both text and subelements.

As you work through this tutorial, you will see how to expand the definition of the title element to include HTML-style markup, which will turn it into a document element as well.

Empty Elements, Revisited

Now that you understand how certain instances of whitespace can be ignorable, it is time revise the definition of an "empty" element. That definition can now be expanded to include

```
<foo>   </foo>
```

where there is whitespace between the tags and the DTD defines that whitespace as ignorable.

Defining Attributes and Entities in the DTD

The DTD you've defined so far is fine for use with the nonvalidating parser. It tells where text is expected and where it isn't, which is all the nonvalidating parser is going to pay attention to. But for use with the validating parser, the DTD needs to specify the valid attributes for the different elements. You'll do that in this section, after which you'll define one internal entity and one external entity that you can reference in your XML file.

Defining Attributes in the DTD

Let's start by defining the attributes for the elements in the slide presentation.

Note: The XML written in this section is contained in `slideshow1b.dtd`. (The browsable version is `slideshow1b-dtd.html`.)

Add the text highlighted below to define the attributes for the `slideshow` element:

```
<!ELEMENT slideshow (slide+)>
<!ATTLIST slideshow
     title    CDATA    #REQUIRED
     date     CDATA    #IMPLIED
     author   CDATA    "unknown"
>
<!ELEMENT slide (title, item*)>
```

The DTD tag `ATTLIST` begins the series of attribute definitions. The name that follows `ATTLIST` specifies the element for which the attributes are being defined. In this case, the element is the `slideshow` element. (Note once again the lack of hierarchy in DTD specifications.)

Each attribute is defined by a series of three space-separated values. Commas and other separators are not allowed, so formatting the definitions as shown above is helpful for readability. The first element in each line is the name of the attribute: `title`, `date`, or `author`, in this case. The second element indicates the type of the data: `CDATA` is character data—unparsed data, once again, in which a left-angle bracket (<) will never be construed as part of an XML tag. Table 6–3 presents the valid choices for the attribute type.

Table 6–3 Attribute Types

Attribute Type	**Specifies...**
`(value1 | value2 | ...)`	A list of values separated by vertical bars. (Example below)
`CDATA`	“Unparsed character data”. (For normal people, a text string.)
`ID`	A name that no other ID attribute shares.
`IDREF`	A reference to an ID defined elsewhere in the document.
`IDREFS`	A space-separated list containing one or more ID references.
`ENTITY`	The name of an entity defined in the DTD.
`ENTITIES`	A space-separated list of entities.
`NMTOKEN`	A valid XML name composed of letters, numbers, hyphens, underscores, and colons.
`NMTOKENS`	A space-separated list of names.
`NOTATION`	The name of a DTD-specified notation, which describes a non-XML data format, such as those used for image files.*

*This is a rapidly obsolescing specification which will be discussed in greater length towards the end of this section.

When the attribute type consists of a parenthesized list of choices separated by vertical bars, the attribute must use one of the specified values. For an example, add the text highlighted below to the DTD:

```
<!ELEMENT slide (title, item*)>
<!ATTLIST slide
     type   (tech | exec | all) #IMPLIED
>
<!ELEMENT title (#PCDATA)>
<!ELEMENT item (#PCDATA | item)* >
```

This specification says that the `slide` element's `type` attribute must be given as `type="tech"`, `type="exec"`, or `type="all"`. No other values are acceptable. (DTD-aware XML editors can use such specifications to present a pop-up list of choices.)

The last entry in the attribute specification determines the attributes default value, if any, and tells whether or not the attribute is required. Table 6–4 shows the possible choices.

Table 6–4 Attribute-Specification Parameters

Specification	**Specifies...**
`#REQUIRED`	The attribute value must be specified in the document.
`#IMPLIED`	The value need not be specified in the document. If it isn't, the application will have a default value it uses.
"defaultValue"	The default value to use, if a value is not specified in the document.
`#FIXED` "fixedValue"	The value to use. If the document specifies any value at all, it must be the same.

Defining Entities in the DTD

So far, you've seen predefined entities like `&amp;` and you've seen that an attribute can reference an entity. It's time now for you to learn how to define entities of your own.

Note: The XML defined here is contained in `slideSample06.xml`. The output is shown in `Echo09-06.txt`. (The browsable versions are `slideSample06-xml.html` and `Echo09-06.html`.)

Add the text highlighted below to the `DOCTYPE` tag in your XML file:

```
<!DOCTYPE slideshow SYSTEM "slideshow.dtd" [
  <!ENTITY product  "WonderWidget">
  <!ENTITY products "WonderWidgets">
]>
```

The `ENTITY` tag name says that you are defining an entity. Next comes the name of the entity and its definition. In this case, you are defining an entity named “product” that will take the place of the product name. Later when the product name changes (as it most certainly will), you will only have to change the name one place, and all your slides will reflect the new value.

The last part is the substitution string that replaces the entity name whenever it is referenced in the XML document. The substitution string is defined in quotes, which are not included when the text is inserted into the document.

Just for good measure, we defined two versions, one singular and one plural, so that when the marketing mavens come up with “Wally” for a product name, you will be prepared to enter the plural as “Wallies” and have it substituted correctly.

Note: Truth be told, this is the kind of thing that really belongs in an external DTD. That way, all your documents can reference the new name when it changes. But, hey, this is an example...

Now that you have the entities defined, the next step is to reference them in the slide show. Make the changes highlighted below to do that:

```
<slideshow
  title="~~WonderWidget~~**&product;** Slide Show"
  ...

  <!-- TITLE SLIDE -->
  <slide type="all">
    <title>Wake up to ~~WonderWidgets~~**&products;**!</title>
  </slide>

   <!-- OVERVIEW -->
  <slide type="all">
    <title>Overview</title>
    <item>Why <em>~~WonderWidgets~~**&products;**</em> are
great</item>
    <item/>
    <item>Who <em>buys</em> ~~WonderWidgets~~**&products;**</item>
  </slide>
```

The points to notice here are that entities you define are referenced with the same syntax (`&entityName;`) that you use for predefined entities, and that the entity can be referenced in an attribute value as well as in an element’s contents.

Echoing the Entity References

When you run the Echo program on this version of the file, here is the kind of thing you see:

```
ELEMENT:  <title>
CHARS:    Wake up to **WonderWidgets**!
END_ELM:  </title>
```

Note that the product name has been substituted for the entity reference.

Additional Useful Entities

Here are several other examples for entity definitions that you might find useful when you write an XML document:

```
<!ENTITY ldquo  "&#147;"> <!-- Left Double Quote -->
<!ENTITY rdquo  "&#148;"> <!-- Right Double Quote -->
<!ENTITY trade  "&#153;"> <!-- Trademark Symbol (TM) -->
<!ENTITY rtrade "&#174;"> <!-- Registered Trademark (R) -->
<!ENTITY copyr  "&#169;"> <!-- Copyright Symbol -->
```

Referencing External Entities

You can also use the `SYSTEM` or `PUBLIC` identifier to name an entity that is defined in an external file. You'll do that now.

Note: The XML defined here is contained in `slideSample07.xml` and in `copyright.xml`. The output is shown in `Echo09-07.txt`. (The browsable versions are `slideSample07-xml.html`, `copyright-xml.html` and `Echo09-07.html`.)

To reference an external entity, add the text highlighted below to the `DOCTYPE` statement in your XML file:

```
<!DOCTYPE slideshow SYSTEM "slideshow.dtd" [
  <!ENTITY product  "WonderWidget">
  <!ENTITY products "WonderWidgets">
  <!ENTITY copyright SYSTEM "copyright.xml">
]>
```

This definition references a copyright message contained in a file named `copyright.xml`. Create that file and put some interesting text in it, perhaps something like this:

```
  <!--  A SAMPLE copyright  -->

This is the standard copyright message that our lawyers
make us put everywhere so we don't have to shell out a
million bucks every time someone spills hot coffee in their
lap...
```

Finally, add the text highlighted below to your `slideSample.xml` file to reference the external entity:

```
<!-- TITLE SLIDE -->
  ...
</slide>

<!-- COPYRIGHT SLIDE -->
<slide type="all">
  <item>&copyright;</item>
</slide>
```

You could also use an external entity declaration to access a servlet that produces the current date using a definition something like this:

```
<!ENTITY currentDate SYSTEM
  "http://www.example.com/servlet/CurrentDate?fmt=dd-MMM-
yyyy">
```

You would then reference that entity the same as any other entity:

```
  Today's date is &currentDate;.
```

Echoing the External Entity

When you run the Echo program on your latest version of the slide presentation, here is what you see:

```
...
END_ELM: </slide>
ELEMENT: <slide
  ATTR: type "all"
>
  ELEMENT: <item>
  CHARS:
This is the standard copyright message that our lawyers
make us put everywhere so we don't have to shell out a
million bucks every time someone spills hot coffee in their
lap...
  END_ELM: </item>
END_ELM: </slide>
...
```

Note that the newline which follows the comment in the file is echoed as a character, but that the comment itself is ignored. That is the reason that the copyright message appears to start on the next line after the `CHARS:` label, instead of immediately after the label—the first character echoed is actually the newline that follows the comment.

Summarizing Entities

An entity that is referenced in the document content, whether internal or external, is termed a general entity. An entity that contains DTD specifications that are referenced from within the DTD is termed a parameter entity. (More on that later.)

An entity which contains XML (text and markup), and which is therefore parsed, is known as a *parsed entity*. An entity which contains binary data (like images) is known as an *unparsed entity*. (By its very nature, it must be external.) We'll be discussing references to unparsed entities in the next section of this tutorial.

Referencing Binary Entities

This section contains no programming exercises. Instead, it discusses the options for referencing binary files like image files and multimedia data files.

Using a MIME Data Type

There are two ways to go about referencing an unparsed entity like a binary image file. One is to use the DTD's `NOTATION`-specification mechanism. However, that mechanism is a complex, non-intuitive holdover that mostly exists for compatibility with SGML documents. We will have occasion to discuss it in a bit more depth when we look at the `DTDHandler` API, but suffice it for now to say that the combination of the recently defined XML namespaces standard, in conjunction with the MIME data types defined for electronic messaging attachments, together provide a much more useful, understandable, and extensible mechanism for referencing unparsed external entities.

Note: The XML described here is in `slideshow1b.dtd`. We won't actually be echoing any images. That's beyond the scope of this tutorial's Echo program. This section is simply for understanding how such references can be made. It assumes that the application which will be processing the XML data knows how to handle such references.

To set up the slideshow to use image files, add the text highlighted below to your `slideshow.dtd` file:

```
<!ELEMENT slide (image?, title, item*)>
<!ATTLIST slide
     type   (tech | exec | all) #IMPLIED
>
<!ELEMENT title (#PCDATA)>
<!ELEMENT item (#PCDATA | item)* >
<!ELEMENT image EMPTY>
<!ATTLIST image
     alt    CDATA    #IMPLIED
     src    CDATA    #REQUIRED
     type   CDATA    "image/gif"
>
```

These modifications declare `image` as an optional element in a `slide`, define it as empty element, and define the attributes it requires. The `image` tag is patterned after the HTML 4.0 tag, `img`, with the addition of an image-type specifier, `type`. (The `img` tag is defined in the HTML 4.0 Specification.)

The `image` tag's attributes are defined by the `ATTLIST` entry. The `alt` attribute, which defines alternate text to display in case the image can't be found, accepts character data (`CDATA`). It has an "implied" value, which means that it is optional, and that the program processing the data knows enough to substitute something like "Image not found". On the other hand, the `src` attribute, which names the image to display, is required.

The `type` attribute is intended for the specification of a MIME data type, as defined at `ftp://ftp.isi.edu/in-notes/iana/assignments/media-types/`. It has a default value: `image/gif`.

Note: It is understood here that the character data (`CDATA`) used for the type attribute will be one of the MIME data types. The two most common formats are: `image/gif`, and `image/jpeg`. Given that fact, it might be nice to specify an attribute list here, using something like:

type ("image/gif", "image/jpeg")

That won't work, however, because attribute lists are restricted to name tokens. The forward slash isn't part of the valid set of name-token characters, so this declaration fails. Besides that, creating an attribute list in the DTD would limit the valid MIME types to those defined today. Leaving it as `CDATA` leaves things more open ended, so that the declaration will continue to be valid as additional types are defined.

In the document, a reference to an image named "intro-pic" might look something like this:

```
<image src="image/intro-pic.gif", alt="Intro Pic",
type="image/gif" />
```

The Alternative: Using Entity References

Using a MIME data type as an attribute of an element is a mechanism that is flexible and expandable. To create an external `ENTITY` reference using the notation mechanism, you need DTD `NOTATION` elements for jpeg and gif data. Those can of course be obtained from some central repository. But then you need to define a different `ENTITY` element for each image you intend to reference! In other words, adding a new image to your document always requires both a new entity definition in the DTD and a reference to it in the document. Given the anticipated ubiquity of the HTML 4.0 specification, the newer standard is to use the MIME data types and a declaration like `image`, which assumes the application knows how to process such elements.

Choosing your Parser Implementation

If no other factory class is specified, the default `SAXParserFactory` class is used. To use a different manufacturer's parser, you can change the value of the environment variable that points to it. You can do that from the command line, like this:

```
java -Djavax.xml.parsers.SAXParserFactory=yourFactoryHere ...
```

The factory name you specify must be a fully qualified class name (all package prefixes included). For more information, see the documentation in the `newInstance()` method of the `SAXParserFactory` class.

Using the Validating Parser

By now, you have done a lot of experimenting with the nonvalidating parser. It's time to have a look at the validating parser and find out what happens when you use it to parse the sample presentation.

Two things to understand about the validating parser at the outset are:

- A schema or Document Type Definition (DTD) is required.
- Since the schema/DTD is present, the `ignorableWhitespace` method is invoked whenever possible.

Configuring the Factory

The first step is modify the Echo program so that it uses the validating parser instead of the nonvalidating parser.

Note: The code in this section is contained in `Echo10.java`.

To use the validating parser, make the changes highlighted below:

```
public static void main(String argv[])
{
  if (argv.length != 1) {
    ...
  }
  ~~// Use the default (non-validating) parser~~
  // Use the validating parser
  SAXParserFactory factory = SAXParserFactory.newInstance();
  factory.setValidating(true);
  try {
    ...
```

Here, you configured the factory so that it will produce a validating parser when `newSAXParser` is invoked. You can also configure it to return a namespace-aware parser using `setNamespaceAware(true)`. The JWSDP implementation supports any combination of configuration options. (If a combination is not supported by any particular implementation, it is required to generate a factory configuration error.)

Validating with XML Schema

Although a full treatment of XML Schema is beyond the scope of this tutorial, this section will show you the steps you need to take to validate an XML document using an existing schema written in the XML Schema language. (To learn more about XML Schema, you can review the online tutorial, *XML Schema Part 0: Primer*, at http://www.w3.org/TR/xmlschema-0/. You can also examine the sample programs that are part of the JAXP download. They use a simple XML Schema definition to validate personnel data stored in an XML file.)

Note: There are multiple schema-definition languages, including RELAX NG, Schematron, and the W3C "XML Schema" standard. (Even a DTD qualifies as a "schema", although it is the only one that does not use XML syntax to describe schema constraints.) However, "XML Schema" presents us with a terminology challenge. While the phrase "XML Schema schema" would be precise, we'll use the phrase "XML Schema definition" to avoid the appearance of redundancy.

To be notified of validation errors in an XML document, the parser factory must be configured to create a validating parser, as shown in the previous section. In addition,

1. The appropriate properties must be set on the SAX parser.
2. The appropriate error handler must be set.
3. The document must be associated with a schema.

Setting the SAX Parser Properties

It's helpful to start by defining the constants you'll use when setting the properties:

```
static final String JAXP_SCHEMA_LANGUAGE =
     "http://java.sun.com/xml/jaxp/properties/schemaLanguage";

static final String W3C_XML_SCHEMA =
     "http://www.w3.org/2001/XMLSchema";
```

Next, you need to configure the parser factory to generate a parser that is namespace-aware parser, as well as validating:

```
...
  SAXParserFactory factory = SAXParserFactory.newInstance();
  factory.setNamespaceAware(true);
  factory.setValidating(true);
```

You'll learn more about namespaces in Using Namespaces (page 277). For now, understand that schema validation is a namespace-oriented process. Since JAXP-compliant parsers are not namespace-aware by default, it is necessary to set the property for schema validation to work.

The last step is to configure the parser to tell it which schema language to use. Here, you will use the constants you defined earlier to specify the W3C's XML Schema language:

```
saxParser.setProperty(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
```

In the process, however, there is an extra error to handle. You'll take a look at that error next.

Setting up the Appropriate Error Handling

In addition to the error handling you've already learned about, there is one error that can occur when you are configuring the parser for schema-based validation. If the parser is not 1.2 compliant, and therefore does not support XML Schema, it could throw a `SAXNotRecognizedException`.

To handle that case, you wrap the `setProperty()` statement in a `try/catch` block, as shown in the code highlighted below.

```
...
SAXParser saxParser = factory.newSAXParser();
try {
  saxParser.setProperty(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
}
catch (SAXNotRecognizedException x) {
  // Happens if the parser does not support JAXP 1.2
  ...
}
...
```

Associating a Document with A Schema

Now that the program is ready to validate the data using an XML Schema definition, it is only necessary to ensure that the XML document is associated with one. There are two ways to do that:

- With a schema declaration in the XML document.
- By specifying the schema to use in the application.

Note: When the application specifies the schema to use, it overrides any schema declaration in the document.

To specify the schema definition in the document, you would create XML like this:

```
<documentRoot
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation='YourSchemaDefinition.xsd'
>
  ...
```

The first attribute defines the XML NameSpace (xmlns) prefix, “xsi”, where “xsi” stands for “XML Schema Instance”. The second line specifies the schema to use for elements in the document that do *not* have a namespace prefix — that is, for the elements you typically define in any simple, uncomplicated XML document.

Note: You’ll be learning about namespaces in Using Namespaces (page 277). For now, think of these attributes as the “magic incantation” you use to validate a simple XML file that doesn’t use them. Once you’ve learned more about namespaces, you’ll see how to use XML Schema to validate complex documents that use them. Those ideas are discussed in Validating with Multiple Namespaces (page 283).

You can also specify the schema file in the application, using code like this:

```
static final String JAXP_SCHEMA_SOURCE =
     "http://java.sun.com/xml/jaxp/properties/schemaSource";

...
```

```
SAXParser saxParser = spf.newSAXParser();
...
saxParser.setProperty(JAXP_SCHEMA_SOURCE,
    new File(schemaSource));
```

Now that you know how to make use of an XML Schema definition, we'll turn our attention to the kinds of errors you can see when the application is validating its incoming data. To that, you'll use a Document Type Definition (DTD) as you experiment with validation.

Experimenting with Validation Errors

To see what happens when the XML document does not specify a DTD, remove the `DOCTYPE` statement from the XML file and run the Echo program on it.

Note: The output shown here is contained in `Echo10-01.txt`. (The browsable version is `Echo10-01.html`.)

The result you see looks like this:

```
<?xml version='1.0' encoding='UTF-8'?>
** Parsing error, line 9, uri .../slideSample01.xml
  Document root element "slideshow", must match DOCTYPE root
"null"
```

Note: The message above was generated by the JAXP 1.2 libraries. If you are using a different parser, the error message is likely to be somewhat different.

This message says that the root element of the document must match the element specified in the `DOCTYPE` declaration. That declaration specifies the document's DTD. Since you don't have one yet, it's value is "null". In other words, the message is saying that you are trying to validate the document, but no DTD has been declared, because no `DOCTYPE` declaration is present.

So now you know that a DTD is a requirement for a valid document. That makes sense. What happens when you run the parser on your current version of the slide presentation, with the DTD specified?

Note: The output shown here, produced from slideSample07.xml is contained in Echo10-07.txt. (The browsable version is Echo10-07.html.)

This time, the parser gives a different error message:

```
  ** Parsing error, line 29, uri file:...
  The content of element type "slide" must match
"(image?,title,item*)
```

Note: The message above was generated by the JAXP 1.2 libraries. If you are using a different parser, the error message is likely to be somewhat different.

This message says that the element found at line 29 (<item>) does not match the definition of the <slide> element in the DTD. The error occurs because the definition says that the slide element requires a title. That element is not optional, and the copyright slide does not have one. To fix the problem, add the question mark highlighted below to make title an optional element:

```
<!ELEMENT slide (image?, title?, item*)>
```

Now what happens when you run the program?

Note: You could also remove the copyright slide, which produces the same result shown below, as reflected in Echo10-06.txt. (The browsable version is Echo10-06.html.)

The answer is that everything runs fine until the parser runs into the <em> tag contained in the overview slide. Since that tag was not defined in the DTD, the attempt to validate the document fails. The output looks like this:

```
  ...
  ELEMENT: <title>
  CHARS:   Overview
  END_ELM: </title>
  ELEMENT: <item>
  CHARS:   Why ** Parsing error, line 28, uri: ...
Element "em" must be declared.
org.xml.sax.SAXParseException: ...
...
```

Note: The message above was generated by the JAXP 1.2 libraries. If you are using a different parser, the error message is likely to be somewhat different.

The error message identifies the part of the DTD that caused validation to fail. In this case it is the line that defines an `item` element as `(#PCDATA | item)`.

Exercise: Make a copy of the file and remove all occurrences of `<em>` from it. Can the file be validated now? (In the next section, you'll learn how to define parameter entries so that we can use XHTML in the elements we are defining as part of the slide presentation.)

Error Handling in the Validating Parser

It is important to recognize that the only reason an exception is thrown when the file fails validation is as a result of the error-handling code you entered in the early stages of this tutorial. That code is reproduced below:

```
public void error(SAXParseException e)
throws SAXParseException
{
  throw e;
}
```

If that exception is not thrown, the validation errors are simply ignored.

Exercise: Try commenting out the line that throws the exception. What happens when you run the parser now?

In general, a SAX parsing *error* is a validation error, although we have seen that it can also be generated if the file specifies a version of XML that the parser is not prepared to handle. The thing to remember is that your application will not generate a validation exception unless you supply an error handler like the one above.

Defining Parameter Entities and Conditional Sections

Just as a general entity lets you reuse XML data in multiple places, a parameter entity lets you reuse parts of a DTD in multiple places. In this section of the tutorial, you'll see how to define and use parameter entities. You'll also see how to use parameter entities with conditional sections in a DTD.

Creating and Referencing a Parameter Entity

Recall that the existing version of the slide presentation could not be validated because the document used `<em>` tags, and those are not part of the DTD. In general, we'd like to use a whole variety of HTML-style tags in the text of a slide, not just one or two, so it makes more sense to use an existing DTD for XHTML than it does to define all the tags we might ever need. A parameter entity is intended for exactly that kind of purpose.

Note: The DTD specifications shown here are contained in `slideshow2.dtd`. The XML file that references it is `slideSample08.xml`. (The browsable versions are `slideshow2-dtd.html` and `slideSample08-xml.html`.)

Open your DTD file for the slide presentation and add the text highlighted below to define a parameter entity that references an external DTD file:

```
<!ELEMENT slide (image?, title?, item*)>
<!ATTLIST slide
        ...
>

<!ENTITY % xhtml SYSTEM "xhtml.dtd">
%xhtml;

<!ELEMENT title ...
```

Here, you used an `<!ENTITY>` tag to define a parameter entity, just as for a general entity, but using a somewhat different syntax. You included a percent sign (%) before the entity name when you defined the entity, and you used the percent sign instead of an ampersand when you referenced it.

Also, note that there are always two steps for using a parameter entity. The first is to define the entity name. The second is to reference the entity name, which actually does the work of including the external definitions in the current DTD. Since the URI for an external entity could contain slashes (/) or other characters that are not valid in an XML name, the definition step allows a valid XML name to be associated with an actual document. (This same technique is used in the definition of namespaces, and anywhere else that XML constructs need to reference external documents.)

Notes:

- The DTD file referenced by this definition is `xhtml.dtd`. You can either copy that file to your system or modify the `SYSTEM` identifier in the `<!ENTITY>` tag to point to the correct URL.
- This file is a small subset of the XHTML specification, loosely modeled after the Modularized XHTML draft, which aims at breaking up the DTD for XHTML into bite-sized chunks, which can then be combined to create different XHTML subsets for different purposes. When work on the modularized XHTML draft has been completed, this version of the DTD should be replaced with something better. For now, this version will suffice for our purposes.

The whole point of using an XHTML-based DTD was to gain access to an entity it defines that covers HTML-style tags like `<em>` and `<b>`. Looking through `xhtml.dtd` reveals the following entity, which does exactly what we want:

```
<!ENTITY % inline "#PCDATA|em|b|a|img|br">
```

This entity is a simpler version of those defined in the Modularized XHTML draft. It defines the HTML-style tags we are most likely to want to use -- emphasis, bold, and break, plus a couple of others for images and anchors that we may or may not use in a slide presentation. To use the `inline` entity, make the changes highlighted below in your DTD file:

```
<!ELEMENT title (#~~PCDATA~~ %inline;)*>
<!ELEMENT item (#~~PCDATA~~ %inline; | item)* >
```

These changes replaced the simple `#PCDATA` item with the `inline` entity. It is important to notice that `#PCDATA` is first in the `inline` entity, and that inline is first wherever we use it. That is required by XML's definition of a mixed-content model. To be in accord with that model, you also had to add an asterisk at the end of the `title` definition. (In the next two sections, you'll see that our definition of the `title` element actually conflicts with a version defined in `xhtml.dtd`, and see different ways to resolve the problem.)

Note: The Modularized XHTML DTD defines both `inline` and `Inline` entities, and does so somewhat differently. Rather than specifying `#PCDATA|em|b|a|img|Br`, their definitions are more like `(#PCDATA|em|b|a|img|Br)*`. Using one of those definitions, therefore, looks more like this:

```
<!ELEMENT title %Inline; >
```

Conditional Sections

Before we proceed with the next programming exercise, it is worth mentioning the use of parameter entities to control *conditional sections*. Although you cannot conditionalize the content of an XML document, you can define conditional sections in a DTD that become part of the DTD only if you specify `include`. If you specify `ignore`, on the other hand, then the conditional section is not included.

Suppose, for example, that you wanted to use slightly different versions of a DTD, depending on whether you were treating the document as an XML document or as a SGML document. You could do that with DTD definitions like the following:

```
someExternal.dtd:
  <![ INCLUDE [
     ... XML-only definitions
  ]]>
  <![ IGNORE [
     ... SGML-only definitions
  ]]>
  ... common definitions
```

The conditional sections are introduced by "`<![`", followed by the `INCLUDE` or `IGNORE` keyword and another "`[`". After that comes the contents of the conditional section, followed by the terminator: "`]]>`". In this case, the XML definitions are included, and the SGML definitions are excluded. That's fine for XML documents, but you can't use the DTD for SGML documents. You could change the keywords, of course, but that only reverses the problem.

The solution is to use references to parameter entities in place of the `INCLUDE` and `IGNORE` keywords:

```
someExternal.dtd:
  <![ %XML; [
     ... XML-only definitions
  ]]>
```

```
<![ %SGML; [
   ... SGML-only definitions
]]>
... common definitions
```

Then each document that uses the DTD can set up the appropriate entity definitions:

```
<!DOCTYPE foo SYSTEM "someExternal.dtd" [
  <!ENTITY % XML  "INCLUDE" >
  <!ENTITY % SGML "IGNORE" >
]>
<foo>
  ...
</foo>
```

This procedure puts each document in control of the DTD. It also replaces the `INCLUDE` and `IGNORE` keywords with variable names that more accurately reflect the purpose of the conditional section, producing a more readable, self-documenting version of the DTD.

Parsing the Parameterized DTD

This section uses the Echo program to see what happens when you reference `xhtml.dtd` in `slideshow.dtd`. It also covers the kinds of warnings that are generated by the SAX parser when a DTD is present.

Note: The output described in this section is contained in `Echo10-08.txt`. (The browsable version is `Echo10-08.html`.)

When you try to echo the slide presentation, you find that it now contains a new error. The relevant part of the output is shown here (formatted for readability):

```
<?xml version='1.0' encoding='UTF-8'?>
** Parsing error, line 22, uri: .../slideshow.dtd
Element type "title" must not be declared more than once.
```

Note: The message above was generated by the JAXP 1.2 libraries. If you are using a different parser, the error message is likely to be somewhat different.

It seems that xhtml.dtd defines a title element which is entirely different from the title element defined in the slideshow DTD. Because there is no hierarchy in the DTD, these two definitions conflict.

Note: The Modularized XHTML DTD also defines a title element that is intended to be the document title, so we can't avoid the conflict by changing xhtml.dtd—the problem would only come back to haunt us later.

You could also use XML namespaces to resolve the conflict, or use one of the more hierarchical schema proposals described in Schema Standards (page 56). For now, though, let's simply rename the title element in slideshow.dtd.

Note: The XML shown here is contained in slideshow3.dtd and slideSample09.xml, which references copyright.xml and xhtml.dtd. The results of processing are shown in Echo10-09.txt. (The browsable versions are slideshow3-dtd.html, slideSample09-xml.html, copyright-xml.html, xhtml-dtd.html, and Echo10-09.html.)

To keep the two title elements separate, we'll resort to a "hyphenation hierarchy". Make the changes highlighted below to change the name of the title element in slideshow.dtd to slide-title:

```
<!ELEMENT slide (image?, slide-title?, item*)>
<!ATTLIST slide
      type   (tech | exec | all) #IMPLIED
>

<!-- Defines the %inline; declaration -->
<!ENTITY % xhtml SYSTEM "xhtml.dtd">
%xhtml;

<!ELEMENT slide-title (%inline;)*>
```

The next step is to modify the XML file to use the new element name. To do that, make the changes highlighted below:

```
...
<slide type="all">
<slide-title>Wake up to ... </slide-title>
</slide>

...

<!-- OVERVIEW -->
<slide type="all">
<slide-title>Overview</slide-title>
<item>...
```

Now run the Echo program on this version of the slide presentation. It should run to completion and display output like that shown in `Echo10-09`.

Congratulations! You have now read a fully validated XML document. The changes you made had the effect of putting your DTD's `title` element into a slideshow "namespace" that you artificially constructed by hyphenating the name. Now the `title` element in the "slideshow namespace" (`slide-title`, really) no longer conflicts with the `title` element in `xhtml.dtd`. In the next section of the tutorial, you'll see how to do that without renaming the definition. To finish off this section, we'll take a look at the kinds of warnings that the validating parser can produce when processing the DTD.

DTD Warnings

As mentioned earlier in this tutorial, warnings are generated only when the SAX parser is processing a DTD. Some warnings are generated only by the validating parser. The nonvalidating parser's main goal is operate as rapidly as possible, but it too generates some warnings. (The explanations that follow tell which does what.)

The XML specification suggests that warnings should be generated as result of:

- Providing additional declarations for entities, attributes, or notations.

  (Such declarations are ignored. Only the first is used. Also, note that duplicate definitions of *elements* always produce a fatal error when validating, as you saw earlier.)

- Referencing an undeclared element type.

(A validity error occurs only if the undeclared type is actually used in the XML document. A warning results when the undeclared element is referenced in the DTD.)

- Declaring attributes for undeclared element types.

The Java XML SAX parser also emits warnings in other cases, such as:

- No <!DOCTYPE ...> when validating.
- Referencing an undefined parameter entity when not validating.

  (When validating, an error results. Although nonvalidating parsers are not required to read parameter entities, the Java XML parser does so. Since it is not a requirement, the Java XML parser generates a warning, rather than an error.)
- Certain cases where the character-encoding declaration does not look right.

At this point, you have digested many XML concepts, including DTDs, external entities. You have also learned your way around the SAX parser. The remainder of the SAX tutorial covers advanced topics that you will only need to understand if you are writing SAX-based applications. If your primary goal is to write DOM-based applications, you can skip ahead to Document Object Model (page 211).

Handling Lexical Events

You saw earlier that if you are writing text out as XML, you need to know if you are in a `CDATA` section. If you are, then angle brackets (<) and ampersands (&) should be output unchanged. But if you're not in a `CDATA` section, they should be replaced by the predefined entities `&lt;` and `&amp;`. But how do you know if you're processing a `CDATA` section?

Then again, if you are filtering XML in some way, you would want to pass comments along. Normally the parser ignores comments. How can you get comments so that you can echo them?

Finally, there are the parsed entity definitions. If an XML-filtering app sees `&myEntity;` it needs to echo the same string—not the text that is inserted in its place. How do you go about doing that?

This section of the tutorial answers those questions. It shows you how to use `org.xml.sax.ext.LexicalHandler` to identify comments, `CDATA` sections, and references to parsed entities.

Comments, `CDATA` tags, and references to parsed entities constitute *lexical* information—that is, information that concerns the text of the XML itself, rather than the XML's information content. Most applications, of course, are concerned only with the *content* of an XML document. Such apps will not use the `LexicalEventListener` API. But apps that output XML text will find it invaluable.

Note: Lexical event handling is a optional parser feature. Parser implementations are not required to support it. (The JWSDP implementation does so.) This discussion assumes that the parser you are using does so, as well.

How the LexicalHandler Works

To be informed when the SAX parser sees lexical information, you configure the `XmlReader` that underlies the parser with a `LexicalHandler`. The `LexicalHandler` interface defines these even-handling methods:

`comment(String comment)`
: Passes comments to the application.

`startCDATA(), endCDATA()`
: Tells when a `CDATA` section is starting and ending, which tells your application what kind of characters to expect the next time `characters()` is called.

`startEntity(String name), endEntity(String name)`
: Gives the name of a parsed entity.

`startDTD(String name, String publicId, String systemId), endDTD()`
: Tells when a DTD is being processed, and identifies it.

Working with a LexicalHandler

In the remainder of this section, you'll convert the Echo app into a lexical handler and play with its features.

Note: The code shown in this section is in `Echo11.java`. The output is shown in `Echo11-09.txt`. (The browsable version is `Echo11-09.html`.)

To start, add the code highlighted below to implement the `LexicalHandler` interface and add the appropriate methods.

```
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.ext.LexicalHandler;
...
public class Echo extends HandlerBase
  implements LexicalHandler
{
  public static void main(String argv[])
     {
        ...
        // Use an instance of ourselves as the SAX event handler
        ~~DefaultHandler handler = new Echo();~~
        Echo handler = new Echo();
        ...
```

At this point, the `Echo` class extends one class and implements an additional interface. You changed the class of the handler variable accordingly, so you can use the same instance as either a `DefaultHandler` or a `LexicalHandler`, as appropriate.

Next, add the code highlighted below to get the `XMLReader` that the parser delegates to, and configure it to send lexical events to your lexical handler:

```
public static void main(String argv[])
{
  ...
  try {
     ...
     // Parse the input
     SAXParser saxParser = factory.newSAXParser();
     XMLReader xmlReader = saxParser.getXMLReader();
     xmlReader.setProperty(
        "http://xml.org/sax/properties/lexical-handler",
        handler
        );
     saxParser.parse( new File(argv[0]), handler);
  } catch (SAXParseException spe) {
     ...
```

Here, you configured the `XMLReader` using the `setProperty()` method defined in the `XMLReader` class. The property name, defined as part of the SAX standard, is the URL, `http://xml.org/sax/properties/lexical-handler`.

Finally, add the code highlighted below to define the appropriate methods that implement the interface.

```
public void warning(SAXParseException err)
  ...
}

public void comment(char[] ch, int start, int length)throws SAX-
Exception
{
}

public void startCDATA()
throws SAXException
{
}

pubic void endCDATA()
throws SAXException
{
}

public void startEntity(String name)
throws SAXException
{
}

public void endEntity(String name)
throws SAXException
{
}

public void startDTD(
  String name, String publicId, String systemId)
throws SAXException
{
}

public void endDTD()
throws SAXException
{
}

private void echoText()
  ...
```

You have now turned the Echo class into a lexical handler. In the next section, you'll start experimenting with lexical events.

Echoing Comments

The next step is to do something with one of the new methods. Add the code highlighted below to echo comments in the XML file:

```
public void comment(char[] ch, int start, int length)
  throws SAXException
{
  String text = new String(ch, start, length);
  nl();
  emit("COMMENT: "+text);
}
```

When you compile the Echo program and run it on your XML file, the result looks something like this:

```
COMMENT:   A SAMPLE set of slides
COMMENT:  FOR WALLY / WALLIES
COMMENT:
  DTD for a simple "slide show".

COMMENT:  Defines the %inline; declaration
COMMENT:  ...
```

The line endings in the comments are passed as part of the comment string, once again normalized to newlines. You can also see that comments in the DTD are echoed along with comments from the file. (That can pose problems when you want to echo only comments that are in the data file. To get around that problem, you can use the startDTD and endDTD methods.)

Echoing Other Lexical Information

To finish up this section, you'll exercise the remaining LexicalHandler methods.

Note: The code shown in this section is in Echo12.java. The file it operates on is slideSample10.xml. (The browsable version is slideSample10-xml.html.) The results of processing are in Echo12-10.

Make the changes highlighted below to remove the comment echo (you don't need that any more) and echo the other events, along with any characters that have been accumulated when an event occurs:

```
public void comment(char[] ch, int start, int length)
throws SAXException
{
  ~~String text = new String(ch, start, length);~~
  ~~nl();~~
  ~~emit("COMMENT: "+text);~~
}

public void startCDATA()
throws SAXException
{
  echoText();
  nl();
  emit("START CDATA SECTION");
}

public void endCDATA()
throws SAXException
{
  echoText();
  nl();
  emit("END CDATA SECTION");
}

public void startEntity(String name)
throws SAXException
{
  echoText();
  nl();
  emit("START ENTITY: "+name);
}

public void endEntity(String name)
throws SAXException
{
  echoText();
  nl();
  emit("END ENTITY: "+name);
}

public void startDTD(String name, String publicId, String
systemId)
throws SAXException
```

```
{
  nl();
  emit("START DTD: "+name
    +"          publicId=" + publicId
    +"          systemId=" + systemId);
}

public void endDTD()
throws SAXException
{
  nl();
  emit("END DTD");
}
```

Here is what you see when the DTD is processed:

```
START DTD: slideshow
       publicId=null
       systemId=file:/..../samples/slideshow3.dtd
START ENTITY: ...
...
END DTD
```

Note: To see events that occur while the DTD is being processed, use `org.xml.sax.ext.DeclHandler`.

Here is some of the additional output you see when the internally defined `products` entity is processed with the latest version of the program:

```
START ENTITY: products
CHARS:   WonderWidgets
END ENTITY: products
```

And here is the additional output you see as a result of processing the external copyright entity:

```
  START ENTITY: copyright
  CHARS:
This is the standard copyright message that our lawyers
make us put everywhere so we don't have to shell out a
million bucks every time someone spills hot coffee in their
lap...

  END ENTITY: copyright
```

Finally, you get output that shows when the `CDATA` section was processed:

```
   START CDATA SECTION
   CHARS:   Diagram:

frobmorten <--------------fuznaten
     |       <3>      ^
     | <1>            |   <1> = fozzle
    V                 |   <2> = framboze
  staten----------------------+   <3> = frenzle
             <2>


   END CDATA SECTION
```

In summary, the `LexicalHandler` gives you the event-notifications you need to produce an accurate reflection of the original XML text.

Note: To accurately echo the input, you would modify the characters() method to echo the text it sees in the appropriate fashion, depending on whether or not the program was in CDATA mode.

Using the DTDHandler and EntityResolver

In this section of the tutorial, we'll carry on a short discussion of the two remaining SAX event handlers: `DTDHandler` and `EntityResolver`. The `DTDHandler` is invoked when the DTD encounters an unparsed entity or a notation declaration. The `EntityResolver` comes into play when a URN (public ID) must be resolved to a URL (system ID).

The DTDHandler API

In the section Referencing Binary Entities (page 184) you saw a method for referencing a file that contains binary data, like an image file, using MIME data types. That is the simplest, most extensible mechanism to use. For compatibility with older SGML-style data, though, it is also possible to define an unparsed entity.

The NDATA keyword defines an unparsed entity, like this:

```
<!ENTITY myEntity SYSTEM "..URL.." NDATA gif>
```

The NDATA keyword says that the data in this entity is not parsable XML data, but is instead data that uses some other notation. In this case, the notation is named “gif”. The DTD must then include a declaration for that notation, which would look something like this:

```
<!NOTATION gif SYSTEM "..URL..">
```

When the parser sees an unparsed entity or a notation declaration, it does nothing with the information except to pass it along to the application using the DTDHandler interface. That interface defines two methods:

```
notationDecl(String name, String publicId, String systemId)

unparsedEntityDecl(String name, String publicId,
  String systemId, String notationName)
```

The notationDecl method is passed the name of the notation and either the public or system identifier, or both, depending on which is declared in the DTD. The unparsedEntityDecl method is passed the name of the entity, the appropriate identifiers, and the name of the notation it uses.

Note: The DTDHandler interface is implemented by the DefaultHandler class.

Notations can also be used in attribute declarations. For example, the following declaration requires notations for the GIF and PNG image-file formats:

```
<!ENTITY image EMPTY>
<!ATTLIST image
     ...
     type  NOTATION  (gif | png) "gif"
>
```

Here, the type is declared as being either gif, or png. The default, if neither is specified, is gif.

Whether the notation reference is used to describe an unparsed entity or an attribute, it is up to the application to do the appropriate processing. The parser knows nothing at all about the semantics of the notations. It only passes on the declarations.

The EntityResolver API

The `EntityResolver` API lets you convert a public ID (URN) into a system ID (URL). Your application may need to do that, for example, to convert something like `href="urn:/someName"` into “`http://someURL`”.

The `EntityResolver` interface defines a single method:

```
resolveEntity(String publicId, String systemId)
```

This method returns an `InputSource` object, which can be used to access the entity’s contents. Converting an URL into an `InputSource` is easy enough. But the URL that is passed as the system ID will be the location of the original document which is, as likely as not, somewhere out on the Web. To access a local copy, if there is one, you must maintain a catalog somewhere on the system that maps names (public IDs) into local URLs.

Further Information

For further information on the Simple API for XML processing (SAX) standard, see:

- The SAX standard page: `http://www.saxproject.org/`

For more information on schema-based validation mechanisms, see:

- The W3C standard validation mechanism, XML Schema: `http://www.w3c.org/XML/Schema`
- RELAX NG’s regular-expression based validation mechanism: `http://www.oasis-open.org/committees/relax-ng/`
- Schematron’s assertion-based validation mechansim: `http://www.ascc.net/xml/resource/schematron/schematron.html`

7

Document Object Model

Eric Armstrong

IN the SAX chapter, you wrote an XML file that contains slides for a presentation. You then used the SAX API to echo the XML to your display.

In this chapter, you'll use the Document Object Model (DOM) to build a small SlideShow application. You'll start by constructing a DOM and inspecting it, then see how to write a DOM as an XML structure, display it in a GUI, and manipulate the tree structure.

A Document Object Model is a garden-variety tree structure, where each node contains one of the components from an XML structure. The two most common types of nodes are *element nodes* and *text nodes*. Using DOM functions lets you create nodes, remove nodes, change their contents, and traverse the node hierarchy.

In this chapter, you'll parse an existing XML file to construct a DOM, display and inspect the DOM hierarchy, convert the DOM into a display-friendly `JTree`, and explore the syntax of namespaces. You'll also create a DOM from scratch, and see how to use some of the implementation-specific features in Sun's JAXP implementation to convert an existing data set to XML.

First though, we'll make sure that DOM is the most appropriate choice for your application. We'll do that in the next section, When to Use DOM.

Note: The examples in this chapter can be found in `<`*`JWSDP_HOME`*`>/docs/tutorial/examples/jaxp/dom/samples`.

When to Use DOM

The Document Object Model (DOM) is a standard that is, above all, designed for *documents* (for example, articles and books). In addition, the JAXP 1.2 implementation supports XML Schema, which may be an important consideration for any given application.

On the other hand, if you are dealing with simple *data* structures, and if XML Schema isn’t a big part of your plans, then you may find that one of the more object-oriented standards like JDOM and dom4j (page 53) is better suited for your purpose.

From the start, DOM was intended to be language neutral. Because it was designed for use with languages like C or Perl, DOM does not take advantage of Java's object-oriented features. That fact, in addition to the document/data distinction, also helps to account for the ways in which processing a DOM differs from processing a JDOM or dom4j structure.

In this section, we'll examine the differences between the models underlying those standards to give help you choose the one that is most appropriate for your application.

Documents vs. Data

The major point of departure between the document model used in DOM and the data model used in JDOM or dom4j lies in:

- The kind of node that exists in the hierarchy.
- The capacity for “mixed-content”.

It is the difference in what constitutes a “node” in the data hierarchy that primarily accounts for the differences in programming with these two models. However, it is the capacity for mixed-content which, more than anything else, accounts for the difference in how the standards define a “node”. So we'll start by examining DOM's “mixed-content model”.

Mixed Content Model

Recall from the discussion of Document-Driven Programming (DDP) (page 49) that text and elements can be freely intermixed in a DOM hierarchy. That kind of structure is dubbed “mixed content” in the DOM model.

Mixed content occurs frequently in documents. For example, to represent this structure:

```
<sentence>This is an <bold>important</bold> idea.</sentence>
```

The hierarchy of DOM nodes would look something like this, where each line represents one node:

```
ELEMENT: sentence
  +  TEXT: This is an
  +  ELEMENT: bold
     + TEXT: important
  + TEXT: idea.
```

Note that the sentence element contains text, followed by a subelement, followed by additional text. It is that intermixing of text and elements that defines the “mixed-content model”.

Kinds of Nodes

In order to provide the capacity for mixed content, DOM nodes are inherently very simple. In the example above, for instance, the “content” of the first element (it’s *value*) simply identifies the kind of node it is.

First time users of a DOM are usually thrown by this fact. After navigating to the <sentence> node, they ask for the node's “content”, and expect to get something useful. Instead, all they get is the name of the element, “sentence”.

Note: The DOM Node API defines `nodeValue()`, `node.nodeType()`, and `nodeName()` methods. For the first element node, `nodeName()` returns “sentence”, while `nodeValue()` returns null. For the first text node, `nodeName()` returns “#text”, and `nodeValue()` returns “This is an “. The important point is that the *value* of an element is not the same as its *content*.

Instead, obtaining the content you care about when processing a DOM means inspecting the list of subelements the node contains, ignoring those you aren't interested in, and processing the ones you do care about.

For example, in the example above, what does it mean if you ask for the “text” of the sentence? Any of the following could be reasonable, depending on your application:

- This is an
- This is an idea.
- This is an important idea.
- This is an <bold>important</bold> idea.

A Simpler Model

With DOM, you are free to create the semantics you need. However, you are also required to do the processing necessary to implement those semantics. Standards like JDOM and dom4j, on the other hand, make it a lot easier to do simple things, because each node in the hierarchy is an object.

Although JDOM and dom4j make allowances for elements with mixed content, they are not primarily designed for such situations. Instead, they are targeted for applications where the XML structure contains data.

As described in Traditional Data Processing (page 49), the elements in a data structure typically contain either text or other elements, but not both. For example, here is some XML that represents a simple address book:

```
<addressbook>
  <entry>
     <name>Fred</name>
     <email>fred@home</email>
  </entry>
     ...
</addressbook>
```

Note: For very simple XML data structures like this one, you could also use the regular expression package (java.util.regex) built into version 1.4 of the Java platform.

In JDOM and dom4j, once you navigate to an element that contains text, you invoke a method like `text()` to get it's content. When processing a DOM, though, you would have to inspect the list of subelements to “put together” the text of the node, as you saw earlier -- even if that list only contained one item (a `TEXT` node).

So for simple data structures like the address book above, you could save yourself a bit of work by using JDOM or dom4j. It may make sense to use one of those models even when the data is technically “mixed”, but when there is always one (and only one) segment of text for a given node.

Here is an example of that kind of structure, which would also be easily processed in JDOM or dom4j:

```
<addressbook>
  <entry>Fred
    <email>fred@home</email>
  </entry>
   ...
</addressbook>
```

Here, each entry has a bit of identifying text, followed by other elements. With this structure, the program could navigate to an entry, invoke `text()` to find out who it belongs to, and process the `<email>` sub element if it is at the correct node.

Increasing the Complexity

But to get a full understanding of the kind of processing you need to do when searching or manipulating a DOM, it is important to know the kinds of nodes that a DOM can conceivably contain.

Here is an example that tries to bring the point home. It is a representation of this data:

```
<sentence>
  The &projectName; <![CDATA[<i>project</i>]]> is
  <?editor: red><bold>important</bold><?editor: normal>.
</sentence>
```

This sentence contains an *entity reference* — a pointer to an “entity” which is defined elsewhere. In this case, the entity contains the name of the project. The example also contains a `CDATA` section (uninterpreted data, like `<pre>` data in HTML), as well as *processing instructions* (`<?...?>`) that in this case tell the editor to which color to use when rendering the text.

Here is the DOM structure for that data. It’s fairly representative of the kind of structure that a robust application should be prepared to handle:

```
+ ELEMENT: sentence
  + TEXT: The
  + ENTITY REF: projectName
    + COMMENT: The latest name we're using
    + TEXT: Eagle
  + CDATA: <i>project</i>
  + TEXT: is
  + PI: editor: red
  + ELEMENT: bold
    + TEXT: important
  + PI: editor: normal
```

This example depicts the kinds of nodes that may occur in a DOM. Although your application may be able to ignore most of them most of the time, a truly robust implementation needs to recognize and deal with each of them.

Similarly, the process of navigating to a node involves processing subelements, ignoring the ones you don't care about and inspecting the ones you do care about, until you find the node you are interested in.

Often, in such cases, you are interested in finding a node that contains specific text. For example, in The DOM API (page 10) you saw an example where you wanted to find a `<coffee>` node whose `<name>` element contains the text, “Mocha Java”. To carry out that search, the program needed to work through the list of `<coffee>` elements and, for each one: a) get the `<name>` element under it and, b) examine the `TEXT` node under that element.

That example made some simplifying assumptions, however. It assumed that processing instructions, comments, `CDATA` nodes, and entity references would not exist in the data structure. Many simple applications can get away with such assumptions. Truly robust applications, on the other hand, need to be prepared to deal with the all kinds of valid XML data.

(A “simple” application will work only so long as the input data contains the simplified XML structures it expects. But there are no validation mechanisms to ensure that more complex structures will not exist. After all, XML was specifically designed to allow them.)

To be more robust, the sample code described in The DOM API (page 10), would have to do these things:

1. When searching for the `<name>` element:
   a. Ignore comments, attributes, and processing instructions.
   b. Allow for the possibility that the `<coffee>` subelements do not occur in the expected order.
   c. Skip over `TEXT` nodes that contain ignorable whitespace, if not validating.
2. When extracting text for a node:
   a. Extract text from `CDATA` nodes as well as text nodes.
   b. Ignore comments, attributes, and processing instructions when gathering the text.
   c. If an entity reference node or another element node is encountered, recurse. (That is, apply the text-extraction procedure to all subnodes.)

Note: The JAXP 1.2 parser does not insert entity reference nodes into the DOM. Instead, it inserts a `TEXT` node containing the contents of the reference. The JAXP 1.1 parser which is built into the 1.4 platform, on the other hand, does insert entity reference nodes. So a robust implementation which is parser-independent needs to be prepared to handle entity reference nodes.

Many applications, of course, won't have to worry about such things, because the kind of data they see will be strictly controlled. But if the data can come from a variety of external sources, then the application will probably need to take these possibilities into account.

The code you need to carry out these functions is given near the end of the DOM tutorial in Searching for Nodes (page 274) and Obtaining Node Content (page 275). Right now, the goal is simply to determine whether DOM is suitable for your application.

Choosing Your Model

As you can see, when you are using DOM, even a simple operation like getting the text from a node can take a bit of programming. So if your programs will be handling simple data structures, JDOM, dom4j, or even the 1.4 regular expression package (`java.util.regex`) may be more appropriate for your needs.

For full-fledged documents and complex applications, on the other hand, DOM gives you a lot of flexibility. And if you need to use XML Schema, then once again DOM is the way to go for now, at least.

If you will be processing both documents *and* data in the applications you develop, then DOM may still be your best choice. After all, once you have written the code to examine and process a DOM structure, it is fairly easy to customize it for a specific purpose. So choosing to do everything in DOM means you'll only have to deal with one set of APIs, rather than two.

Plus, the DOM standard *is* a standard. It is robust and complete, and it has many implementations. That is a significant decision-making factor for many large installations — particularly for production applications, to prevent doing large rewrites in the event of an API change.

Finally, even though the text in an address book may not permit bold, italics, colors, and font sizes today, someday you may want to handle things. Since DOM will handle virtually anything you throw at it, choosing DOM makes it easier to “future-proof” your application.

Reading XML Data into a DOM

In this section of the tutorial, you’ll construct a Document Object Model (DOM) by reading in an existing XML file. In the following sections, you’ll see how to display the XML in a Swing tree component and practice manipulating the DOM.

Note: In the next part of the tutorial, XML Stylesheet Language for Transformations (page 289), you’ll see how to write out a DOM as an XML file. (You’ll also see how to convert an existing data file into XML with relative ease.)

Creating the Program

The Document Object Model (DOM) provides APIs that let you create nodes, modify them, delete and rearrange them. So it is relatively easy to create a DOM, as you’ll see in later in section 5 of this tutorial, Creating and Manipulating a DOM (page 268).

Before you try to create a DOM, however, it is helpful to understand how a DOM is structured. This series of exercises will make DOM internals visible by displaying them in a Swing `JTree`.

Create the Skeleton

Now that you've had a quick overview of how to create a DOM, let's build a simple program to read an XML document into a DOM then write it back out again.

Note: The code discussed in this section is in `DomEcho01.java`. The file it operates on is `slideSample01.xml`. (The browsable version is `slideSample01-xml.html`.)

Start with a normal basic logic for an app, and check to make sure that an argument has been supplied on the command line:

```
public class DomEcho {
  public static void main(String argv[])
  {
     if (argv.length != 1) {
        System.err.println(
             "Usage: java DomEcho filename");
        System.exit(1);
     }
  }// main
}// DomEcho
```

Import the Required Classes

In this section, you're going to see all the classes individually named. That's so you can see where each class comes from when you want to reference the API documentation. In your own apps, you may well want to replace import statements like those below with the shorter form: `javax.xml.parsers.*`.

Add these lines to import the JAXP APIs you'll be using:

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

Add these lines for the exceptions that can be thrown when the XML document is parsed:

```
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
```

Add these lines to read the sample XML file and identify errors:

```
import java.io.File;
import java.io.IOException;
```

Finally, import the W3C definition for a DOM and DOM exceptions:

```
import org.w3c.dom.Document;
import org.w3c.dom.DOMException;
```

Note: A `DOMException` is only thrown when traversing or manipulating a DOM. Errors that occur during parsing are reporting using a different mechanism that is covered below.

Declare the DOM

The `org.w3c.dom.Document` class is the W3C name for a Document Object Model (DOM). Whether you parse an XML document or create one, a Document instance will result. We'll want to reference that object from another method later on in the tutorial, so define it as a global object here:

```
public class DomEcho
{
  static Document document;

  public static void main(String argv[])
  {
```

It needs to be `static`, because you're going to generate its contents from the `main` method in a few minutes.

Handle Errors

Next, put in the error handling logic. This logic is basically the same as the code you saw in Handling Errors with the Nonvalidating Parser (page 155) in the

SAX tutorial, so we won't go into it in detail here. The major point worth noting is that a JAXP-conformant document builder is required to report SAX exceptions when it has trouble parsing the XML document. The DOM parser does not have to actually use a SAX parser internally, but since the SAX standard was already there, it seemed to make sense to use it for reporting errors. As a result, the error-handling code for DOM and SAX applications are very similar:

```
public static void main(String argv[])
{
  if (argv.length != 1) {
     ...
  }

  try {

} catch (SAXParseException spe) {
  // Error generated by the parser
     System.out.println("\n** Parsing error"
        + ", line " + spe.getLineNumber()
        + ", uri " + spe.getSystemId());
     System.out.println("   " + spe.getMessage() );

     // Use the contained exception, if any
     Exception  x = spe;
     if (spe.getException() != null)
        x = spe.getException();
     x.printStackTrace();

  } catch (SAXException sxe) {
     // Error generated during parsing
     Exception  x = sxe;
     if (sxe.getException() != null)
        x = sxe.getException();
     x.printStackTrace();

   } catch (ParserConfigurationException pce) {
     // Parser with specified options can't be built
     pce.printStackTrace();

   } catch (IOException ioe) {
     // I/O error
     ioe.printStackTrace();
  }

}// main
```

Instantiate the Factory

Next, add the code highlighted below to obtain an instance of a factory that can give us a document builder:

```
public static void main(String argv[])
{
  if (argv.length != 1) {
     ...
  }
  DocumentBuilderFactory factory =
     DocumentBuilderFactory.newInstance();
  try {
```

Get a Parser and Parse the File

Now, add the code highlighted below to get a instance of a builder, and use it to parse the specified file:

```
try {
  DocumentBuilder builder = factory.newDocumentBuilder();
  document = builder.parse( new File(argv[0]) );
} catch (SAXParseException spe) {
```

Save This File!

By now, you should be getting the idea that every JAXP application starts pretty much the same way. You're right! Save this version of the file as a template. You'll use it later on as the basis for an XSLT transformation application.

Run the Program

Throughout most of the DOM tutorial, you'll be using the sample slideshows you saw in the SAX section. In particular, you'll use `slideSample01.xml`, a simple XML file with nothing much in it, and `slideSample10.xml`, a more complex example that includes a DTD, processing instructions, entity references, and a CDATA section.

For instructions on how to compile and run your program, see Compiling and Running the Program from the SAX tutorial. Substitute "DomEcho" for "Echo" as the name of the program, and you're ready to roll.

For now, just run the program on `slideSample01.xml`. If it ran without error, you have successfully parsed an XML document and constructed a DOM. Congratulations!

Note: You'll have to take my word for it, for the moment, because at this point you don't have any way to display the results. But that feature is coming shortly...

Additional Information

Now that you have successfully read in a DOM, there are one or two more things you need to know in order to use `DocumentBuilder` effectively. Namely, you need to know about:

- Configuring the Factory
- Handling Validation Errors

Configuring the Factory

By default, the factory returns a nonvalidating parser that knows nothing about namespaces. To get a validating parser, and/or one that understands namespaces, you configure the factory to set either or both of those options using the command(s) highlighted below:

```
public static void main(String argv[])
{
  if (argv.length != 1) {
     ...
  }
  DocumentBuilderFactory factory =
     DocumentBuilderFactory.newInstance();
  factory.setValidating(true);
  factory.setNamespaceAware(true);
  try {
     ...
```

Note: JAXP-conformant parsers are not required to support all combinations of those options, even though the reference parser does. If you specify an invalid combination of options, the factory generates a `ParserConfigurationException` when you attempt to obtain a parser instance.

You'll be learning more about how to use namespaces in the last section of the DOM tutorial, Using Namespaces (page 277). To complete this section, though, you'll want to learn something about...

Handling Validation Errors

Remember when you were wading through the SAX tutorial, and all you really wanted to do was construct a DOM? Well, here's when that information begins to pay off.

Recall that the default response to a validation error, as dictated by the SAX standard, is to do nothing. The JAXP standard requires throwing SAX exceptions, so you use exactly the same error handling mechanisms as you used for a SAX application. In particular, you need to use the `DocumentBuilder`'s `setErrorHandler` method to supply it with an object that implements the SAX `ErrorHandler` interface.

Note: `DocumentBuilder` also has a `setEntityResolver` method you can use

The code below uses an anonymous inner class to define that `ErrorHandler`. The highlighted code is the part that makes sure validation errors generate an exception.

```
builder.setErrorHandler(
  new org.xml.sax.ErrorHandler() {
     // ignore fatal errors (an exception is guaranteed)
     public void fatalError(SAXParseException exception)
     throws SAXException {
     }
     // treat validation errors as fatal
     public void error(SAXParseException e)
     throws SAXParseException
     {
        throw e;
     }

      // dump warnings too
     public void warning(SAXParseException err)
     throws SAXParseException
     {
        System.out.println("** Warning"
           + ", line " + err.getLineNumber()
           + ", uri " + err.getSystemId());
```

```
            System.out.println("   " + err.getMessage());
        }

    );
```

This code uses an anonymous inner class to generate an instance of an object that implements the `ErrorHandler` interface. Since it has no class name, it's "anonymous". You can think of it as an "ErrorHandler" instance, although technically it's a no-name instance that implements the specified interface. The code is substantially the same as that described in Handling Errors with the Nonvalidating Parser (page 155). For a more complete background on validation issues, refer to Using the Validating Parser (page 187).

Looking Ahead

In the next section, you'll display the DOM structure in a JTree and begin to explore its structure. For example, you'll see how entity references and CDATA sections appear in the DOM. And perhaps most importantly, you'll see how text nodes (which contain the actual data) reside *under* element nodes in a DOM.

Displaying a DOM Hierarchy

To create a Document Object Hierarchy (DOM) or manipulate one, it helps to have a clear idea of how the nodes in a DOM are structured. In this section of the tutorial, you'll expose the internal structure of a DOM.

Echoing Tree Nodes

What you need at this point is a way to expose the nodes in a DOM so you can see what it contains. To do that, you'll convert a DOM into a `JTreeModel` and display the full DOM in a `JTree`. It's going to take a bit of work, but the end result will be a diagnostic tool you can use in the future, as well as something you can use to learn about DOM structure now.

Convert DomEcho to a GUI App

Since the DOM is a tree, and the Swing `JTree` component is all about displaying trees, it makes sense to stuff the DOM into a `JTree`, so you can look at it. The first step in that process is to hack up the `DomEcho` program so it becomes a GUI application.

Note: The code discussed in this section is in `DomEcho02.java`.

Add Import Statements

Start by importing the GUI components you're going to need to set up the application and display a `JTree`:

```
// GUI components and layouts
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTree;
```

Later on in the DOM tutorial, we'll tailor the DOM display to generate a user-friendly version of the `JTree` display. When the user selects an element in that tree, you'll be displaying subelements in an adjacent editor pane. So, while we're doing the setup work here, import the components you need to set up a divided view (`JSplitPane`) and to display the text of the subelements (`JEditorPane`):

```
import javax.swing.JSplitPane;
import javax.swing.JEditorPane;
```

Add a few support classes you're going to need to get this thing off the ground:

```
// GUI support classes
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.Toolkit;
import java.awt.event.WindowEvent;
import java.awt.event.WindowAdapter;
```

Finally, import some classes to make a fancy border:

```
// For creating borders
import javax.swing.border.EmptyBorder;
import javax.swing.border.BevelBorder;
import javax.swing.border.CompoundBorder;
```

(These are optional. You can skip them and the code that depends on them if you want to simplify things.)

Create the GUI Framework

The next step is to convert the application into a GUI application. To do that, the static main method will create an instance of the main class, which will have become a GUI pane.

Start by converting the class into a GUI pane by extending the Swing `JPanel` class:

```
public class DomEcho02 extends JPanel
{
  // Global value so it can be ref'd by the tree-adapter
  static Document document;
  ...
```

While you're there, define a few constants you'll use to control window sizes:

```
public class DomEcho02 extends JPanel
{
  // Global value so it can be ref'd by the tree-adapter
  static Document document;

  static final int windowHeight = 460;
  static final int leftWidth = 300;
  static final int rightWidth = 340;
  static final int windowWidth = leftWidth + rightWidth;
```

Now, in the main method, invoke a method that will create the outer frame that the GUI pane will sit in:

```
public static void main(String argv[])
{
  ...
  DocumentBuilderFactory factory ...
  try {
     DocumentBuilder builder = factory.newDocumentBuilder();
     document = builder.parse( new File(argv[0]) );
     makeFrame();

   } catch (SAXParseException spe) {
     ...
```

Next, you'll need to define the `makeFrame` method itself. It contains the standard code to create a frame, handle the exit condition gracefully, give it an instance of the main panel, size it, locate it on the screen, and make it visible:

```
    ...
} // main

public static void makeFrame()
{
  // Set up a GUI framework
  JFrame frame = new JFrame("DOM Echo");
  frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e)
      {System.exit(0);}
  });

  // Set up the tree, the views, and display it all
  final DomEcho02 echoPanel = new DomEcho02();
  frame.getContentPane().add("Center", echoPanel );
  frame.pack();
  Dimension screenSize =
    Toolkit.getDefaultToolkit().getScreenSize();
  int w = windowWidth + 10;
  int h = windowHeight + 10;
  frame.setLocation(screenSize.width/3 - w/2,
           screenSize.height/2 - h/2);
  frame.setSize(w, h);
  frame.setVisible(true)
} // makeFrame
```

Add the Display Components

The only thing left in the effort to convert the program to a GUI application is to create the class constructor and make it create the panel's contents. Here is the constructor:

```
public class DomEcho02 extends JPanel
{
  ...
  static final int windowWidth = leftWidth + rightWidth;

  public DomEcho02()
  {
  } // Constructor
```

Here, you make use of the border classes you imported earlier to make a regal border (optional):

```
public DomEcho02()
{
  // Make a nice border
  EmptyBorder eb = new EmptyBorder(5,5,5,5);
  BevelBorder bb = new BevelBorder(BevelBorder.LOWERED);
  CompoundBorder cb = new CompoundBorder(eb,bb);
  this.setBorder(new CompoundBorder(cb,eb));

} // Constructor
```

Next, create an empty tree and put it a `JScrollPane` so users can see its contents as it gets large:

```
public DomEcho02(
{
  ...

  // Set up the tree
  JTree tree = new JTree();

  // Build left-side view
  JScrollPane treeView = new JScrollPane(tree);
  treeView.setPreferredSize(
    new Dimension( leftWidth, windowHeight ));

} // Constructor
```

Now create a non-editable `JEditPane` that will eventually hold the contents pointed to by selected `JTree` nodes:

```
public DomEcho02(
{
  ....

  // Build right-side view
  JEditorPane htmlPane = new JEditorPane("text/html","");
  htmlPane.setEditable(false);
  JScrollPane htmlView = new JScrollPane(htmlPane);
  htmlView.setPreferredSize(
    new Dimension( rightWidth, windowHeight ));

}  // Constructor
```

With the left-side `JTree` and the right-side `JEditorPane` constructed, create a `JSplitPane` to hold them:

```
public DomEcho02()
{
  ....

  // Build split-pane view
  JSplitPane splitPane =
    new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
            treeView, htmlView );
  splitPane.setContinuousLayout( true );
  splitPane.setDividerLocation( leftWidth );
  splitPane.setPreferredSize(
    new Dimension( windowWidth + 10, windowHeight+10 ));

}  // Constructor
```

With this code, you set up the `JSplitPane` with a vertical divider. That produces a “horizontal split” between the tree and the editor pane. (More of a horizontal layout, really.) You also set the location of the divider so that the tree got the width it prefers, with the remainder of the window width allocated to the editor pane.

Finally, specify the layout for the panel and add the split pane:

```
public DomEcho02()
{
  ...

  // Add GUI components
  this.setLayout(new BorderLayout());
  this.add("Center", splitPane );

} // Constructor
```

Congratulations! The program is now a GUI application. You can run it now to see what the general layout will look like on screen. For reference, here is the completed constructor:

```
public DomEcho02()
{
  // Make a nice border
  EmptyBorder eb = new EmptyBorder(5,5,5,5);
  BevelBorder bb = new BevelBorder(BevelBorder.LOWERED);
  CompoundBorder CB = new CompoundBorder(eb,bb);
```

```
      this.setBorder(new CompoundBorder(CB,eb));

      // Set up the tree
      JTree tree = new JTree();

      // Build left-side view
      JScrollPane treeView = new JScrollPane(tree);
      treeView.setPreferredSize(
         new Dimension( leftWidth, windowHeight ));

      // Build right-side view
      JEditorPane htmlPane = new JEditorPane("text/html","");
      htmlPane.setEditable(false);
      JScrollPane htmlView = new JScrollPane(htmlPane);
      htmlView.setPreferredSize(
         new Dimension( rightWidth, windowHeight ));

      // Build split-pane view
      JSplitPane splitPane =
         new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                treeView, htmlView )
      splitPane.setContinuousLayout( true );
      splitPane.setDividerLocation( leftWidth );
      splitPane.setPreferredSize(
         new Dimension( windowWidth + 10, windowHeight+10 ));

      // Add GUI components
      this.setLayout(new BorderLayout());
      this.add("Center", splitPane );

   } // Constructor
```

Create Adapters to Display the DOM in a JTree

Now that you have a GUI framework to display a `JTree` in, the next step is get the `JTree` to display the DOM. But a `JTree` wants to display a `TreeModel`. A DOM is a tree, but it's not a `TreeModel`. So you'll need to create an adapter class that makes the DOM look like a `TreeModel` to a `JTree`.

Now, when the `TreeModel` passes nodes to the `JTree`, `JTree` uses the `toString` function of those nodes to get the text to display in the tree. The standard `toString` function isn't going to be very pretty, so you'll need to wrap the DOM nodes in an `AdapterNode` that returns the text we want. What the `TreeModel` gives to the `JTree`, then, will in fact be `AdapterNode` objects that wrap DOM nodes.

Note: The classes that follow are defined as inner classes. If you are coding for the 1.1 platform, you will need to define these class as external classes.

Define the AdapterNode Class

Start by importing the tree, event, and utility classes you're going to need to make this work:

```
// For creating a TreeModel
import javax.swing.tree.*;
import javax.swing.event.*;
import java.util.*;

public class DomEcho extends JPanel
{
```

Moving back down to the end of the program, define a set of strings for the node element types:

```
        ...
} // makeFrame

// An array of names for DOM node-types
// (Array indexes = nodeType() values.)
static final String[] typeName = {
  "none",
  "Element",
  "Attr",
  "Text",
  "CDATA",
  "EntityRef",
  "Entity",
  "ProcInstr",
  "Comment",
  "Document",
  "DocType",
  "DocFragment",
  "Notation",
};
```

} // DomEcho

These are the strings that will be displayed in the `JTree`. The specification of these nodes types can be found in the Document Object Model (DOM) Level 2 Core Specification at `http://www.w3.org/TR/2000/REC-DOM/Level-2-Core-20001113`, under the specification for Node. That table is reproduced below, with the headings modified for clarity, and with the `nodeType`() column added:

Table 7–1 Node Types

Node	**nodeName()**	**nodeValue()**	**attributes**	**nodeType()**
Attr	name of attribute	value of attribute	null	2
CDATASection	#cdata-section	content of the CDATA section	null	4
Comment	#comment	content of the comment	null	8
Document	#document	null	null	9
DocumentFragment	#document-fragment	null	null	11
DocumentType	document type name	null	null	10
Element	tag name	null	NamedNodeMap	1
Entity	entity name	null	null	6
EntityReference	name of entity referenced	null	null	5
Notation	notation name	null	null	12
ProcessingInstruction	target	entire content excluding the target	null	7
Text	#text	content of the text node	null	3

Suggestion:

Print this table and keep it handy. You need it when working with the DOM, because all of these types are intermixed in a DOM tree. So your code is forever asking, “Is this the kind of node I’m interested in?”.

Next, define the `AdapterNode` wrapper for DOM nodes as an inner class:

```
static final String[] typeName = {
   ...
};

public class AdapterNode
{
   org.w3c.dom.Node domNode;

   // Construct an Adapter node from a DOM node
   public AdapterNode(org.w3c.dom.Node node) {
      domNode = node;
   }

   // Return a string that identifies this node
   //   in the tree
   public String toString() {
      String s = typeName[domNode.getNodeType()];
      String nodeName = domNode.getNodeName();
      if (! nodeName.startsWith("#")) {
         s += ": " + nodeName;
      }
      if (domNode.getNodeValue() != null) {
         if (s.startsWith("ProcInstr"))
            s += ", ";
         else
            s += ": ";

         // Trim the value to get rid of NL's
         //   at the front
         String t = domNode.getNodeValue().trim();
         int x = t.indexOf(");
         if (x >= 0) t = t.substring(0, x);
         s += t;
      }
      return s;
   }

} // AdapterNode

} // DomEcho
```

This class declares a variable to hold the DOM node, and requires it to be specified as a constructor argument. It then defines the `toString` operation, which returns the node type from the `String` array, and then adds to that additional information from the node, to further identify it.

As you can see in the table of node types in `org.w3c.dom.Node`, every node has a type, and name, and a value, which may or may not be empty. In those cases where the node name starts with "`#`", that field duplicates the node type, so there is in point in including it. That explains the lines that read:

```
if (! nodeName.startsWith("#")) {
  s += ": " + nodeName;
}
```

The remainder of the `toString` method deserves a couple of notes, as well. For instance, these lines:

```
if (s.startsWith("ProcInstr"))
  s += ", ";
else
  s += ": ";
```

Merely provide a little "syntactic sugar". The type field for a Processing Instructions end with a colon (:) anyway, so those codes keep from doubling the colon.

The other interesting lines are:

```
String t = domNode.getNodeValue().trim();
int x = t.indexOf(");
if (x >= 0) t = t.substring(0, x);
s += t;
```

Those lines trim the value field down to the first newline (linefeed) character in the field. If you leave those lines out, you will see some funny characters (square boxes, typically) in the `JTree`.

Note: Recall that XML stipulates that all line endings are normalized to newlines, regardless of the system the data comes from. That makes programming quite a bit simpler.

Wrapping a `DomNode` and returning the desired string are the `AdapterNode`'s major functions. But since the `TreeModel` adapter will need to answer questions like "How many children does this node have?" and satisfy commands like

"Give me this node's Nth child", it will be helpful to define a few additional utility methods. (The adapter could always access the DOM node and get that information for itself, but this way things are more encapsulated.)

Next, add the code highlighted below to return the index of a specified child, the child that corresponds to a given index, and the count of child nodes:

```
public class AdapterNode
{
  ...
  public String toString() {
    ...
  }

  public int index(AdapterNode child) {
    //System.err.println("Looking for index of " + child);
    int count = childCount();
    for (int i=0; i<count; i++) {
      AdapterNode n = this.child(i);
      if (child == n) return i;
    }
    return -1; // Should never get here.
  }

  public AdapterNode child(int searchIndex) {
    //Note: JTree index is zero-based.
    org.w3c.dom.Node node =
      domNode.getChildNodes().item(searchIndex);
    return new AdapterNode(node);
  }

  public int childCount() {
    return domNode.getChildNodes().getLength();
  }

} // AdapterNode

} // DomEcho
```

Note: During development, it was only after I started writing the `TreeModel` adapter that I realized these were needed, and went back to add them. In just a moment, you'll see why.

Define the TreeModel Adapter

Now, at last, you are ready to write the `TreeModel` adapter. One of the really nice things about the `JTree` model is the relative ease with which you convert an existing tree for display. One of the reasons for that is the clear separation between the displayable view, which `JTree` uses, and the modifiable view, which the application uses. For more on that separation, see Understanding the TreeModel at `http://java.sun.com/products/jfc/tsc/articles/jtree/index.html`. For now, the important point is that to satisfy the `TreeModel` interface we only need to (a) provide methods to access and report on children and (b) register the appropriate `JTree` listener, so it knows to update its view when the underlying model changes.

Add the code highlighted below to create the `TreeModel` adapter and specify the child-processing methods:

```
     ...
  } // AdapterNode

  // This adapter converts the current Document (a DOM) into
  // a JTree model.
  public class DomToTreeModelAdapter implements
  javax.swing.tree.TreeModel
  {
    // Basic TreeModel operations
    public Object  getRoot() {
       //System.err.println("Returning root: " +document);
       return new AdapterNode(document);
    }

    public boolean isLeaf(Object aNode) {
       // Determines whether the icon shows up to the left.
       // Return true for any node with no children
       AdapterNode node = (AdapterNode) aNode;
       if (node.childCount() > 0) return false;
       return true;
    }

    public int     getChildCount(Object parent)
       AdapterNode node = (AdapterNode) parent;
       return node.childCount();
    }

    public Object  getChild(Object parent, int index) {
       AdapterNode node = (AdapterNode) parent;
       return node.child(index);
```

```
      }

      public int    getIndexOfChild(Object parent, Object child) {
         AdapterNode node = (AdapterNode) parent;
         return node.index((AdapterNode) child);
      }

      public void valueForPathChanged(
                TreePath path, Object newValue)
      {
         // Null. We won't be making changes in the GUI
         // If we did, we would ensure the new value was
         // really new and then fire a TreeNodesChanged event.
      }

   } // DomToTreeModelAdapter

   } // DomEcho
```

In this code, the `getRoot` method returns the root node of the DOM, wrapped as an `AdapterNode` object. From here on, all nodes returned by the adapter will be `AdapterNodes` that wrap DOM nodes. By the same token, whenever the `JTree` asks for the child of a given parent, the number of children that parent has, etc., the `JTree` will be passing us an `AdapterNode`. We know that, because we control every node the `JTree` sees, starting with the root node.

`JTree` uses the `isLeaf` method to determine whether or not to display a clickable expand/contract icon to the left of the node, so that method returns true only if the node has children. In this method, we see the cast from the generic object `JTree` sends us to the `AdapterNode` object we know it has to be. *We* know it is sending us an adapter object, but the interface, to be general, defines objects, so we have to do the casts.

The next three methods return the number of children for a given node, the child that lives at a given index, and the index of a given child, respectively. That’s all pretty straightforward.

The last method is invoked when the user changes a value stored in the `JTree`. In this app, we won’t support that. But if we did, the application would have to make the change to the underlying model and then inform any listeners that a change had occurred. (The `JTree` might not be the only listener. In many an application it isn’t, in fact.)

To inform listeners that a change occurred, you'll need the ability to register them. That brings us to the last two methods required to implement the `TreeModel` interface. Add the code highlighted below to define them:

```
public class DomToTreeModelAdapter ...
{
  ...
  public void valueForPathChanged(
     TreePath path, Object newValue)
  {
     ...
  }
  private Vector listenerList = new Vector();
  public void addTreeModelListener(
     TreeModelListener listener ) {
     if ( listener != null
     && ! listenerList.contains(listener) ) {
        listenerList.addElement( listener );
     }
  }

  public void removeTreeModelListener(
     TreeModelListener listener )
  {
     if ( listener != null ) {
        listenerList.removeElement( listener );
     }
  }

} // DomToTreeModelAdapter
```

Since this application won't be making changes to the tree, these methods will go unused, for now. However, they'll be there in the future, when you need them.

Note: This example uses `Vector` so it will work with 1.1 apps. If coding for 1.2 or later, though, I'd use the excellent collections framework instead:

```
  private LinkedList listenerList = new LinkedList();
```

The operations on the `List` are then `add` and `remove`. To iterate over the list, as in the operations below, you would use:

```
Iterator it = listenerList.iterator();
while ( it.hasNext() ) {
  TreeModelListener listener = (TreeModelListener) it.next();
    ...
}
```

Here, too, are some optional methods you won't be using in this application. At this point, though, you have constructed a reasonable template for a TreeModel adapter. In the interests of completeness, you might want to add the code highlighted below. You can then invoke them whenever you need to notify `JTree` listeners of a change:

```
public void removeTreeModelListener(
   TreeModelListener listener)
{
   ...
}

public void fireTreeNodesChanged( TreeModelEvent e ) {
   Enumeration listeners = listenerList.elements();
   while ( listeners.hasMoreElements() ) {
      TreeModelListener listener =
         (TreeModelListener) listeners.nextElement();
      listener.treeNodesChanged( e );
   }
}

public void fireTreeNodesInserted( TreeModelEvent e ) {
   Enumeration listeners = listenerList.elements();
   while ( listeners.hasMoreElements() ) {
      TreeModelListener listener =
         (TreeModelListener) listeners.nextElement();
      listener.treeNodesInserted( e );
   }
}

public void fireTreeNodesRemoved( TreeModelEvent e ) {
   Enumeration listeners = listenerList.elements();
   while ( listeners.hasMoreElements() ) {
      TreeModelListener listener =
         (TreeModelListener) listeners.nextElement();
      listener.treeNodesRemoved( e );
   }
}
```

```
    public void fireTreeStructureChanged( TreeModelEvent e ) {
      Enumeration listeners = listenerList.elements();
      while ( listeners.hasMoreElements() ) {
        TreeModelListener listener =
          (TreeModelListener) listeners.nextElement();
        listener.treeStructureChanged( e );
      }
    }

  } // DomToTreeModelAdapter
```

Note: These methods are taken from the `TreeModelSupport` class described in Understanding the TreeModel. That architecture was produced by Tom Santos and Steve Wilson, and is a lot more elegant than the quick hack going on here. It seemed worthwhile to put them here, though, so they would be immediately at hand when and if they're needed.

Finishing Up

At this point, you are basically done. All you need to do is jump back to the constructor and add the code to construct an adapter and deliver it to the `JTree` as the `TreeModel`:

```
// Set up the tree
JTree tree = new JTree(new DomToTreeModelAdapter());
```

You can now compile and run the code on an XML file. In the next section, you will do that, and explore the DOM structures that result.

Examining the Structure of a DOM

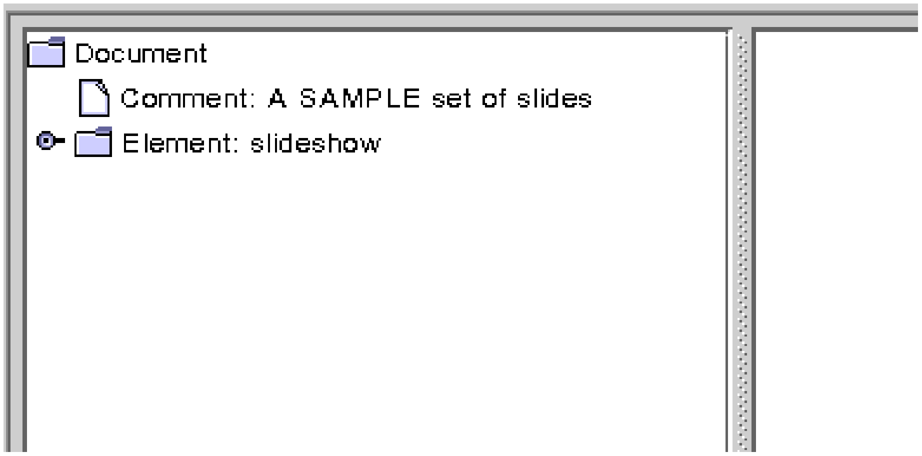
In this section, you'll use the GUI-fied `DomEcho` application you created in the last section to visually examine a DOM. You'll see what nodes make up the DOM, and how they are arranged. With the understanding you acquire, you'll be well prepared to construct and modify Document Object Model structures in the future.

Displaying A Simple Tree

We'll start out by displaying a simple file, so you get an idea of basic DOM structure. Then we'll look at the structure that results when you include some of the more advanced XML elements.

Note: The code used to create the figures in this section is in `DomEcho02.java`. The file displayed is `slideSample01.xml`. (The browsable version is `slideSample01-xml.html`.)

Figure 7–1 shows the tree you see when you run the DomEcho program on the first XML file you created in the DOM tutorial.



```
Document
    Comment: A SAMPLE set of slides
    Element: slideshow
```

Figure 7–1 Document, Comment, and Element Nodes Displayed

Recall that the first bit of text displayed for each node is the element `type`. After that comes the element `name`, if any, and then the element `value`. This view shows three element types: `Document`, `Comment`, and `Element`. There is only `Document` type for the whole tree—that is the root node. The `Comment` node displays the `value` attribute, while the `Element` node displays the element `name`, “slideshow”.

Compare Figure 7–1 with the code in the `AdapterNode`'s `toString` method to see whether the name or value is being displayed for a particular node. If you need to make it more clear, modify the program to indicate which property is being displayed (for example, with N: *name*, V: *value*).

Expanding the slideshow element brings up the display shown in Figure 7–2.

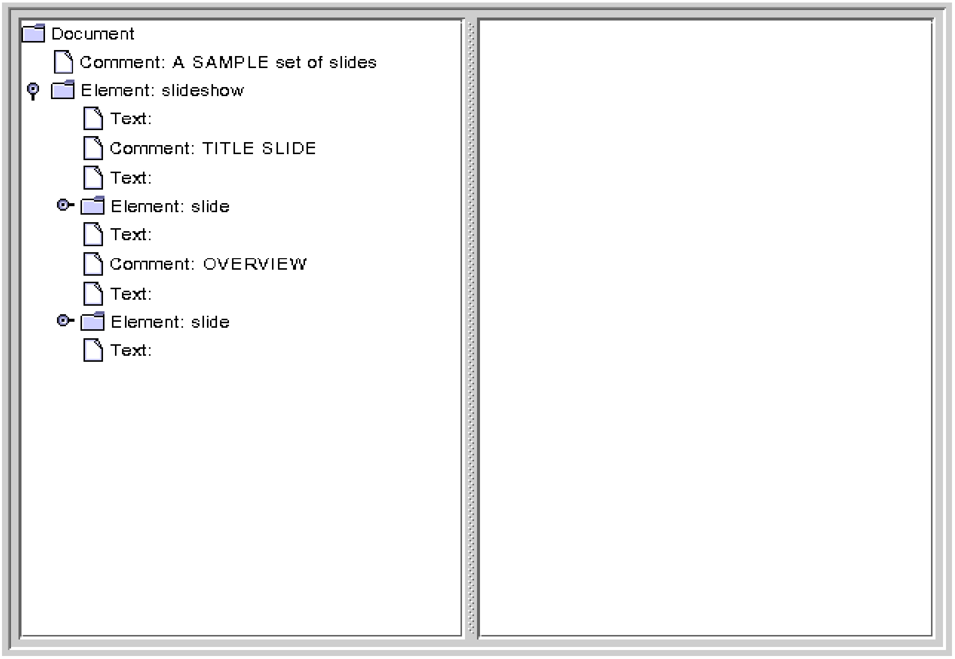


Figure 7–2 Element Node Expanded, No Attribute Nodes Showing

Here, you can see the `Text` nodes and `Comment` nodes that are interspersed between Slide elements. The empty `Text` nodes exist because there is no DTD to tell the parser that no text exists. (Generally, the vast majority of nodes in a DOM tree will be `Element` and `Text` nodes.)

Important!
Text nodes exist *under* element nodes in a DOM, and data is *always* stored in text nodes. Perhaps the most common error in DOM processing is to navigate to an element node and expect it to contain the data that is stored in that element. Not so! Even the simplest element node has a text node under it. For example, given `<size>12</size>`, there is an element node (`size`), *and a text node under it* which contains the actual data (`12`).

Notably absent from this picture are the `Attribute` nodes. An inspection of the table in `org.w3c.dom.Node` shows that there is indeed an Attribute node type. But they are not included as children in the DOM hierarchy. They are instead obtained via the Node interface `getAttributes` method.

Note: The display of the text nodes is the reason for including the lines below in the AdapterNode's `toString` method. If your remove them, you'll see the funny characters (typically square blocks) that are generated by the newline characters that are in the text.

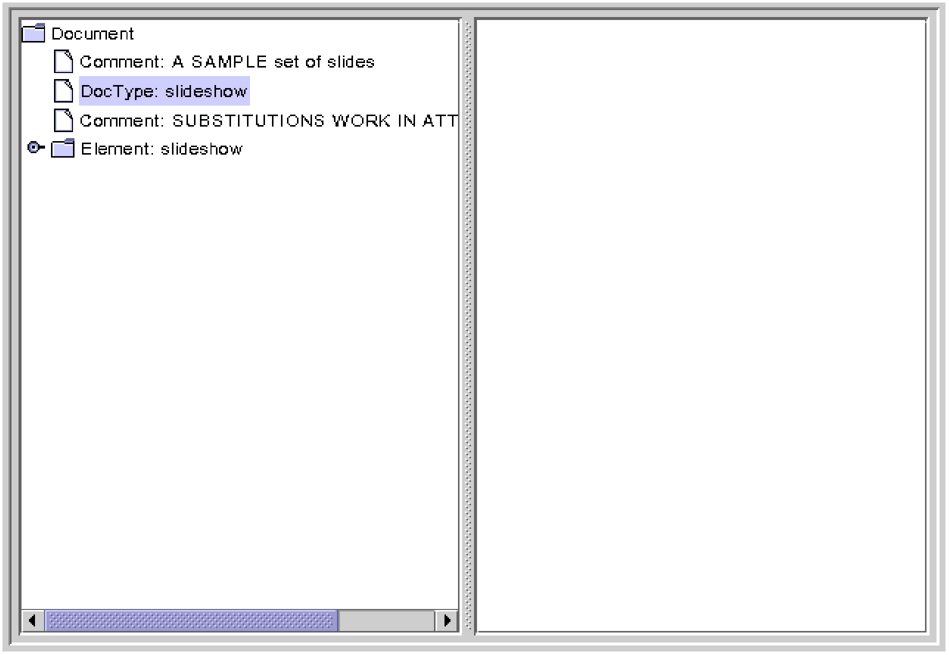
```
String t = domNode.getNodeValue().trim();
int x = t.indexOf(");
if (x >= 0) t = t.substring(0, x);
s += t;
```

Displaying a More Complex Tree

Here, you'll display the example XML file you created at the end of the SAX tutorial, to see how entity references, processing instructions, and CDATA sections appear in the DOM.

Note: The file displayed in this section is `slideSample10.xml`. The `slideSample10.xml` file references `slideshow3.dtd` which, in turn, references `copyright.xml` and a (very simplistic) `xhtml.dtd`. (The browsable versions are `slideSample10-xml.html`, `slideshow3-dtd.html`, `copyright-xml.html`, and `xhtml-dtd.html`.)

Figure 7–3 shows the result of running the `DomEcho` application on `slideSample10.xml`, which includes a `DOCTYPE` entry that identifies the document's DTD.



Document
Comment: A SAMPLE set of slides
DocType: slideshow
Comment: SUBSTITUTIONS WORK IN ATT
Element: slideshow

Figure 7–3 DocType Node Displayed

The `DocType` interface is actually an extension of `w3c.org.dom.Node`. It defines a `getEntities` method that you would use to obtain `Entity` nodes—the nodes that define entities like the `product` entity, which has the value "WonderWidgets". Like `Attribute` nodes, `Entity` nodes do not appear as children of DOM nodes.

When you expand the `slideshow` node, you get the display shown in Figure 7–4.

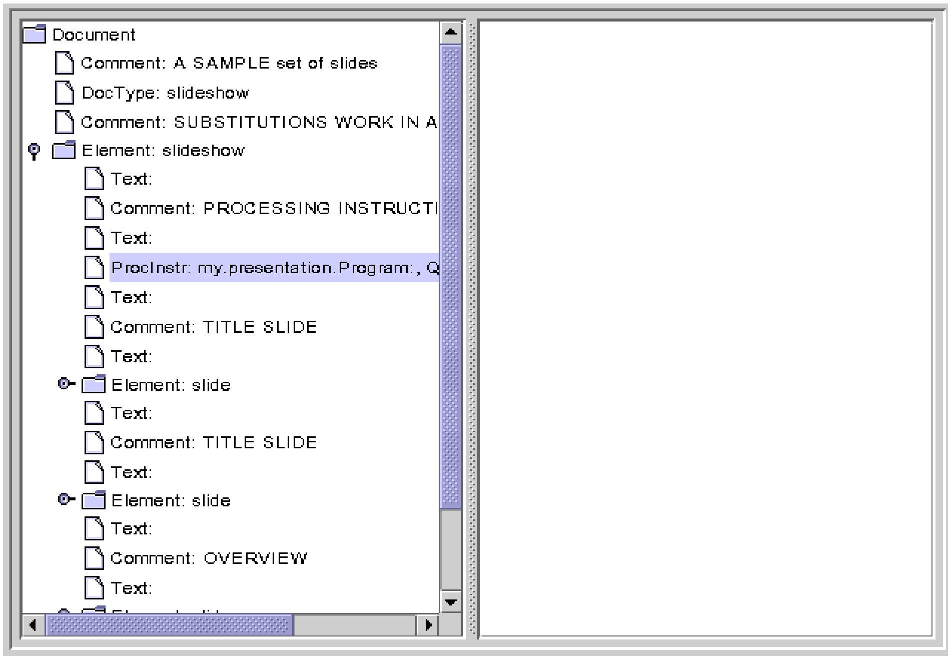
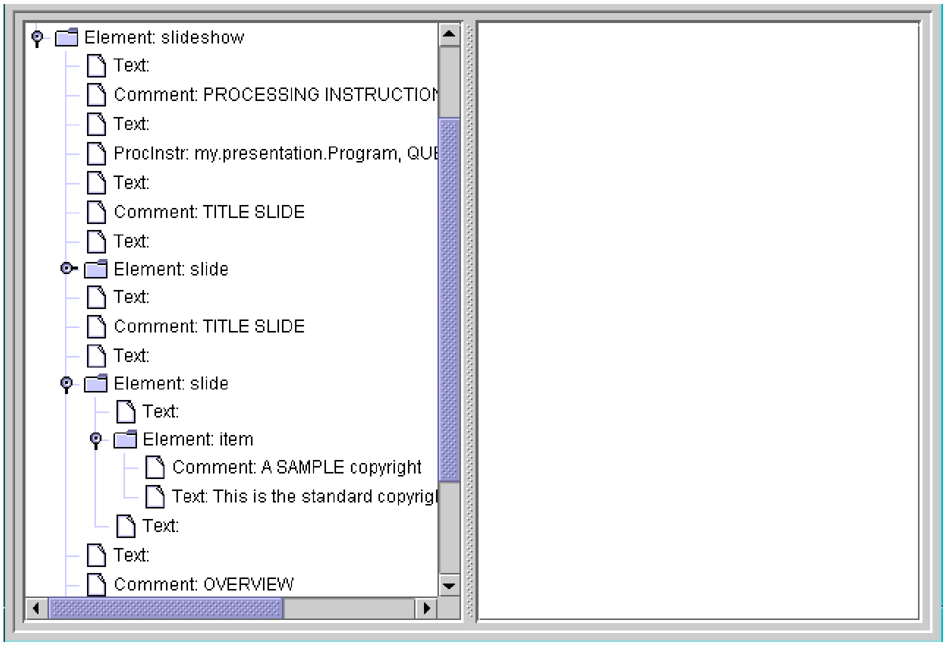

Document
Comment: A SAMPLE set of slides
DocType: slideshow
Comment: SUBSTITUTIONS WORK IN A
Element: slideshow
Text:
Comment: PROCESSING INSTRUCTI
Text:
ProcInstr: my.presentation.Program:, Q
Text:
Comment: TITLE SLIDE
Text:
Element: slide
Text:
Comment: TITLE SLIDE
Text:
Element: slide
Text:
Comment: OVERVIEW
Text:

Figure 7–4 Processing Instruction Node Displayed

Here, the processing instruction node is highlighted, showing that those nodes do appear in the tree. The `name` property contains the target-specification, which identifies the application that the instruction is directed to. The `value` property contains the text of the instruction.

Note that empty text nodes are also shown here, even though the DTD specifies that a `slideshow` can contain `slide` elements only, never text. Logically, then, you might think that these nodes would not appear. (When this file was run through the SAX parser, those elements generated `ignorableWhitespace` events, rather than `character` events.)

Moving down to the second `slide` element and opening the `item` element under it brings up the display shown in Figure 7–5.



```
Element: slideshow
    Text:
    Comment: PROCESSING INSTRUCTION
    Text:
    ProcInstr: my.presentation.Program, QUE
    Text:
    Comment: TITLE SLIDE
    Text:
    Element: slide
    Text:
    Comment: TITLE SLIDE
    Text:
    Element: slide
        Text:
        Element: item
            Comment: A SAMPLE copyright
            Text: This is the standard copyrigl
        Text:
    Text:
    Comment: OVERVIEW
```

Figure 7–5 JAXP 1.2 DOM — Item Text Returned from an Entity Reference

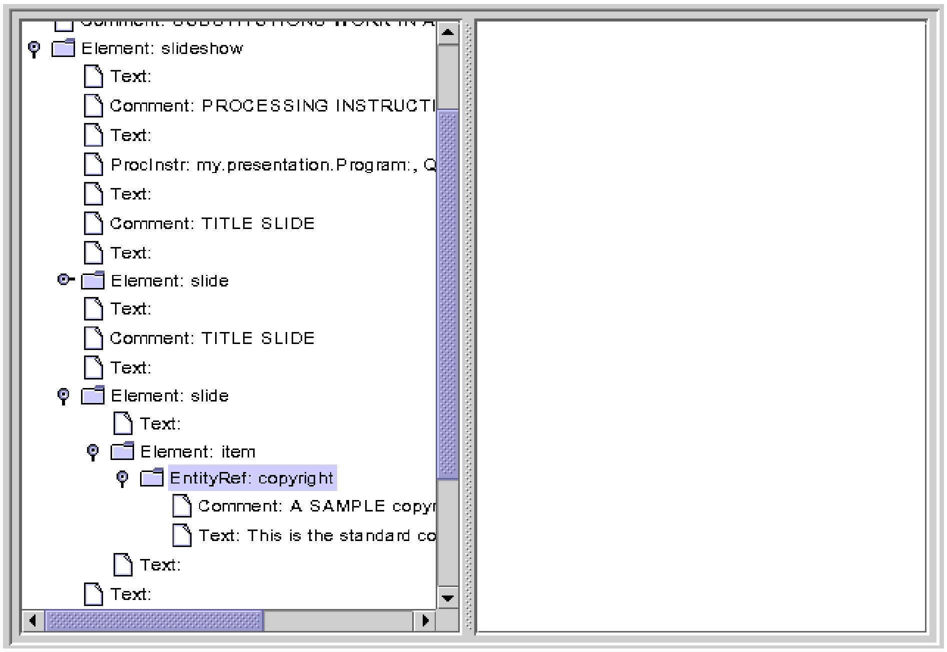
Here, you can see that a text node containing the copyright text was inserted into the DOM, rather than the entity reference which pointed to it.

For most applications, the insertion of the text is exactly what you want. That way, when you're looking for the text under a node, you don't have to worry about an entity references it might contain.

For other applications, though, you may need the ability to reconstruct the original XML. For example, an editor application would need to save the result of user modifications without throwing away entity references in the process.

Various `DocumentBuilderFactory` APIs give you control over the kind of DOM structure that is created. For example, add the highlighted line below to produce the DOM structure shown in Figure 7–6.

```
public static void main(String argv[])
{
  ...
  DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
  factory.setExpandEntityReferences(true);
  ...
```

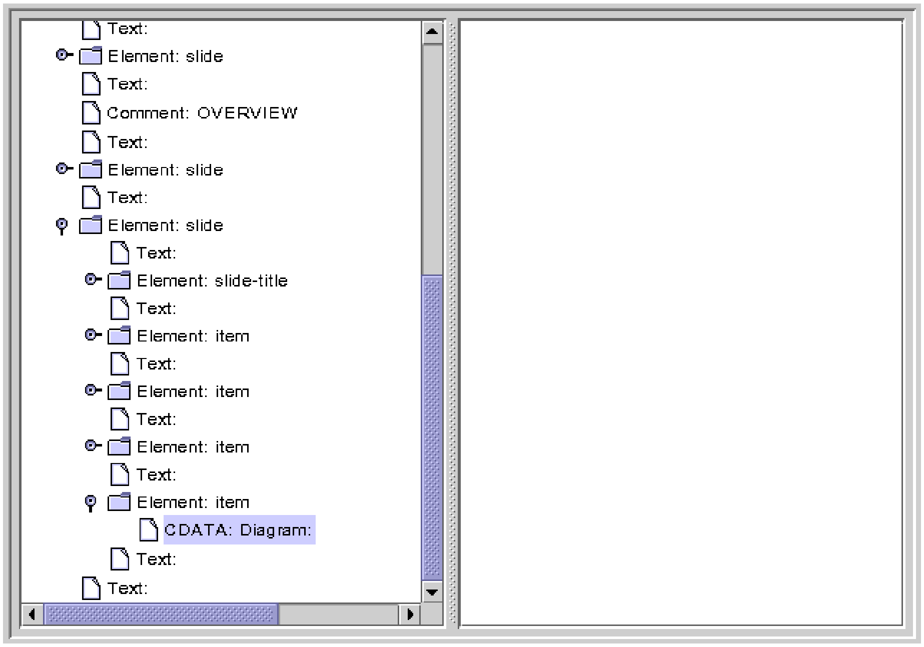


```
  Comment: SUBSTITUTIONS WORK IN A
Element: slideshow
    Text:
    Comment: PROCESSING INSTRUCTI
    Text:
    ProcInstr: my.presentation.Program:, Q
    Text:
    Comment: TITLE SLIDE
    Text:
  Element: slide
    Text:
    Comment: TITLE SLIDE
    Text:
  Element: slide
      Text:
    Element: item
      EntityRef: copyright
          Comment: A SAMPLE copyr
          Text: This is the standard co
      Text:
    Text:
```

Figure 7–6 JAXP 1.1 in 1.4 Platform — Entity Reference Node Displayed

Here, the Entity Reference node is highlighted. Note that the entity reference contains multiple nodes under it. This example shows only comment and a text nodes, but the entity could conceivably contain other element nodes, as well.

Finally, moving down to the last `item` element under the last `slide` brings up the display shown in Figure 7–7.



```
Text:
Element: slide
Text:
Comment: OVERVIEW
Text:
Element: slide
Text:
Element: slide
    Text:
    Element: slide-title
    Text:
    Element: item
    Text:
    Element: item
    Text:
    Element: item
    Text:
    Element: item
        CDATA: Diagram:
    Text:
Text:
```

Figure 7–7 CDATA Node Displayed

Here, the `CDATA` node is highlighted. Note that there are no nodes under it. Since a `CDATA` section is entirely uninterpreted, all of its contents are contained in the node's `value` property.

Summary of Lexical Controls

Lexical information is the information you need to reconstruct the original syntax of an XML document. As we discussed earlier, preserving lexical information is important for editing applications, where you want to save a document that is an accurate reflection of the original -- complete with comments, entity references, and any CDATA sections it may have included at the outset.

A majority of applications, however, are only concerned with the content of the XML structures. They can afford to ignore comments, and they don't care whether data was coded in a CDATA section, as plain text, or whether it included an entity reference. For such applications, a minimum of lexical information is desirable, because it simplifies the number and kind of DOM nodes that the application has to be prepared to examine.

The following `DocumentBuilderFactory` methods give you control over the lexical information you see in the DOM:

- `setCoalescing()`
  To convert CDATA nodes to Text node and append to an adjacent Text node (if any).
- `setExpandEntityReferences()`
  To expand entity reference nodes.
- `setIgnoringComments()`
  To ignore comments.
- `setIgnoringElementContentWhitespace()`

  To ignore ignorable whitespace in element content.

The default values for all of these properties is `false`. Table 7–2 shows the settings you need to preserve all the lexical information necessary to reconstruct the original document, in its original form. It also shows the settings that construct the simplest possible DOM, so the application can focus on the data's semantic content, without having to worry about lexical syntax details.

Table 7–2 Configuring `DocumentBuilderFactory`

API	**Preserve Lexical Info**	**Focus on Content**
`setCoalescing()`	false	true
`setExpandEntityReferences()`	true	false
`setIgnoringComments()`	false	true
`setIgnoringElementContentWhitespace()`	false	true

Finishing Up

At this point, you have seen most of the nodes you will ever encounter in a DOM tree. There are one or two more that we'll mention in the next section, but you now know what you need to know to create or modify a DOM structure. In the next section, you'll see how to convert a DOM into a `JTree` that is suitable for an interactive GUI. Or, if you prefer, you can skip ahead to the 5th section of the DOM tutorial, Creating and Manipulating a DOM (page 268), where you'll learn how to create a DOM from scratch.

Constructing a User-Friendly JTree from a DOM

Now that you know what a DOM looks like internally, you'll be better prepared to modify a DOM or construct one from scratch. Before going on to that, though, this section presents some modifications to the `JTreeModel` that let you produce a more user-friendly version of the `JTree` suitable for use in a GUI.

Compressing the Tree View

Displaying the DOM in tree form is all very well for experimenting and to learn how a DOM works. But it's not the kind of "friendly" display that most users want to see in a `JTree`. However, it turns out that very few modifications are needed to turn the `TreeModel` adapter into something that *will* present a user-friendly display. In this section, you'll make those modifications.

Note: The code discussed in this section is in `DomEcho03.java`. The file it operates on is `slideSample01.xml`. (The browsable version is `slideSample01-xml.html`.)

Make the Operation Selectable

When you modify the adapter, you're going to *compress* the view of the DOM, eliminating all but the nodes you really want to display. Start by defining a boolean variable that controls whether you want the compressed or uncompressed view of the DOM:

```
public class DomEcho extends JPanel
{
  static Document document;
   boolean compress = true;
   static final int windowHeight = 460;
   ...
```

Identify Tree Nodes

The next step is to identify the nodes you want to show up in the tree. To do that, add the code highlighted below:

```
...
import org.w3c.dom.Document;
import org.w3c.dom.DOMException;
import org.w3c.dom.Node;

public class DomEcho extends JPanel
{
  ...

  public static void makeFrame() {
     ...
  }

  // An array of names for DOM node-type
  static final String[] typeName = {
     ...
  };

  static final int ELEMENT_TYPE = Node.ELEMENT_NODE;

  // The list of elements to display in the tree
  static String[] treeElementNames = {
     "slideshow",
     "slide",
     "title",         // For slideshow #1
     "slide-title",   // For slideshow #10
     "item",
  };

  boolean treeElement(String elementName) {
     for (int i=0; i<treeElementNames.length; i++) {
        if ( elementName.equals(treeElementNames[i]) )
           return true;
     }
     return false;
  }
```

With this code, you set up a constant you can use to identify the `ELEMENT` node type, declared the names of the elements you want in the tree, and created a method tells whether or not a given element name is a “tree element”. Since `slideSample01.xml` has `title` elements and `slideSample10.xml` has `slide-`

title elements, you set up the contents of this arrays so it would work with either data file.

Note: The mechanism you are creating here depends on the fact that *structure* nodes like slideshow and slide never contain text, while text usually does appear in *content* nodes like item. Although those "content" nodes may contain subelements in slideShow10.xml, the DTD constrains those subelements to be XHTML nodes. Because they are XHTML nodes (an XML version of HTML that is constrained to be well-formed), the entire substructure under an item node can be combined into a single string and displayed in the htmlPane that makes up the other half of the application window. In the second part of this section, you'll do that concatenation, displaying the text and XHTML as content in the htmlPane.

Although you could simply reference the node types defined in the class, org.w3c.dom.Node, defining the ELEMENT_TYPE constant keeps the code a little more readable. Each node in the DOM has a name, a type, and (potentially) a list of subnodes. The functions that return these values are getNodeName(), getNodeType, and getChildNodes(). Defining our own constants will let us write code like this:

```
Node node = nodeList.item(i);
int type = node.getNodeType();
if (type == ELEMENT_TYPE) {
  ....
```

As a stylistic choice, the extra constants help us keep the reader (and ourselves!) clear about what we're doing. Here, it is fairly clear when we are dealing with a node object, and when we are dealing with a type constant. Otherwise, it would be fairly tempting to code something like, if (node == ELEMENT_NODE), which of course would not work at all.

Control Node Visibility

The next step is to modify the AdapterNode's childCount function so that it only counts "tree element" nodes—nodes which are designated as displayable in the JTree. Make the modifications highlighted below to do that:

```
public class DomEcho extends JPanel
{
  ...
  public class AdapterNode
  {
      ...
      public AdapterNode child(int searchIndex) {
         ...
      }
      public int childCount() {
         if (!compress) {
            // Indent this
            return domNode.getChildNodes().getLength();
         }
         int count = 0;
         for (int i=0;
            i<domNode.getChildNodes().getLength(); i++)
         {
            org.w3c.dom.Node node =
               domNode.getChildNodes().item(i);
            if (node.getNodeType() == ELEMENT_TYPE
            && treeElement( node.getNodeName() ))
            {
               ++count;
            }
         }
         return count;
      }
   } // AdapterNode
```

The only tricky part about this code is checking to make sure the node is an element node before comparing the node. The DocType node makes that necessary, because it has the same name, “slideshow”, as the `slideshow` element.

Control Child Access

Finally, you need to modify the `AdapterNode’s` `child` function to return the Nth item from the list of displayable nodes, rather than the Nth item from all nodes in the list. Add the code highlighted below to do that:

```
public class DomEcho extends JPanel
{
   ...
   public class AdapterNode
   {
      ...
      public int index(AdapterNode child) {
         ...
      }
      public AdapterNode child(int searchIndex) {
      //Note: JTree index is zero-based.
      org.w3c.dom.Node node =
         domNode.getChildNodes()Item(searchIndex);
      if (compress) {
         // Return Nth displayable node
         int elementNodeIndex = 0;
         for (int i=0;
            i<domNode.getChildNodes().getLength(); i++)
         {
            node = domNode.getChildNodes()Item(i);
            if (node.getNodeType() == ELEMENT_TYPE
            && treeElement( node.getNodeName() )
            && elementNodeIndex++ == searchIndex) {
               break;
            }
         }
      }
      return new AdapterNode(node);
   } // child
}  // AdapterNode
```

There's nothing special going on here. It's a slightly modified version the same logic you used when returning the child count.

Check the Results

When you compile and run this version of the application on `slideSample01.xml`, and then expand the nodes in the tree, you see the results shown in Figure 7–8. The only nodes remaining in the tree are the high-level "structure" nodes.

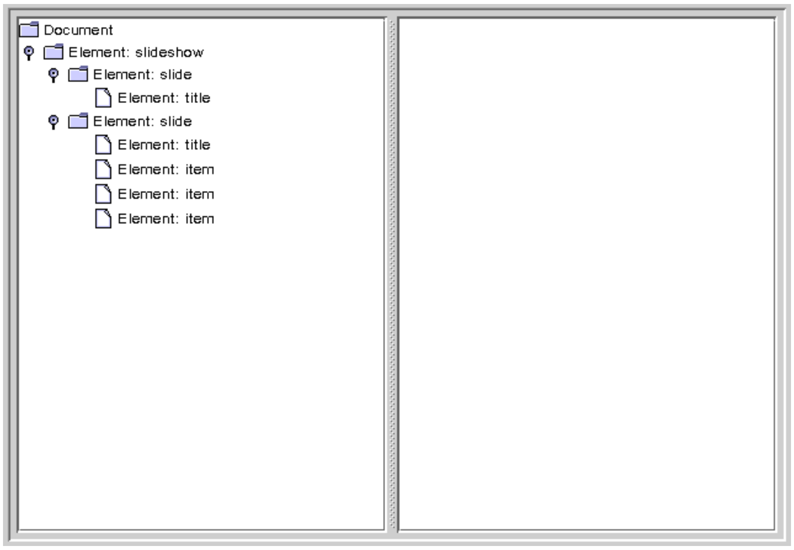


Figure 7–8 Tree View with a Collapsed Hierarchy

Extra Credit

The way the application stands now, the information that tells the application how to compress the tree for display is “hard-coded”. Here are some ways you could consider extending the app:

Use a Command-Line Argument

Whether you compress or don’t compress the tree could be determined by a command line argument, rather than being a hard-coded boolean variable. On the other hand, the list the list of elements that goes into the tree is still hard coded, so maybe that option doesn’t make much sense, unless...

Read the treeElement list from a file

If you read the list of elements to include in the tree from an external file, that would make the whole application command driven. That would be good. But wouldn’t it be really nice to derive that information from the DTD or schema, instead? So you might want to consider...

Automatically Build the List

Watch out, though! As things stand right now, there are no standard DTD parsers! If you use a DTD, then, you’ll need to write your parser to make sense out of its somewhat arcane syntax. You’ll probably have better luck if you use a schema, instead of a DTD. The nice thing about schemas is that use XML syntax, so you can use an XML parser to read the schema the same way you use any other file.

As you analyze the schema, note that the `JTree`-displayable *structure* nodes are those that have no text, while the *content* nodes may contain text and, optionally, XHTML subnodes. That distinction works for this example, and will likely work for a large body of real-world applications. It's pretty easy to construct cases that will create a problem, though, so you'll have to be on the lookout for schema/DTD specifications that embed non-XHTML elements in text-capable nodes, and take the appropriate action.

Acting on Tree Selections

Now that the tree is being displayed properly, the next step is to concatenate the subtrees under selected nodes to display them in the `htmlPane`. While you're at it, you'll use the concatenated text to put node-identifying information back in the `JTree`.

Note: The code discussed in this section is in `DomEcho04.java`.

Identify Node Types

When you concatenate the subnodes under an element, the processing you do is going to depend on the type of node. So the first thing to is to define constants for the remaining node types. Add the code highlighted below to do that:

```
public class DomEcho extends JPanel
{
  ...
  // An array of names for DOM node-types
  static final String[] typeName = {
    ...
  };
  static final int ELEMENT_TYPE =   1;
  static final int ATTR_TYPE =Node.ATTRIBUTE_NODE;
  static final int TEXT_TYPE =Node.TEXT_NODE;
  static final int CDATA_TYPE = Node.CDATA_SECTION_NODE;
  static final int ENTITYREF_TYPE =
Node.ENTITY_REFERENCE_NODE;
  static final int ENTITY_TYPE =Node.ENTITY_NODE;
  static final int PROCINSTR_TYPE =
               Node.PROCESSING_INSTRUCTION_NODE;
    static final int COMMENT_TYPE = Node.COMMENT_NODE;
    static final int DOCUMENT_TYPE =Node.DOCUMENT_NODE;
    static final int DOCTYPE_TYPE =Node.DOCUMENT_TYPE_NODE;
    static final int DOCFRAG_TYPE =Node.DOCUMENT_FRAGMENT_NODE;
    static final int NOTATION_TYPE =Node.NOTATION_NODE;
```

Concatenate Subnodes to Define Element Content

Next, you need to define add the method that concatenates the text and subnodes for an element and returns it as the element's "content". To define the content method, you'll need to add the big chunk of code highlighted below, but this is the last big chunk of code in the DOM tutorial!.

```
public class DomEcho extends JPanel
{
  ...
  public class AdapterNode
  {
    ...
    public String toString() {
    ...
    }
    public String content() {
      String s = "";
      org.w3c.dom.NodeList nodeList =
        domNode.getChildNodes();
      for (int i=0; i<nodeList.getLength(); i++) {
        org.w3c.dom.Node node = nodeList.item(i);
        int type = node.getNodeType();
        AdapterNode adpNode = new AdapterNode(node);
        if (type == ELEMENT_TYPE) {
          if ( treeElement(node.getNodeName()) )
            continue;
          s += "<" + node.getNodeName() + ">";
          s += adpNode.content();
          s += "</" + node.getNodeName() + ">";
        } else if (type == TEXT_TYPE) {
          s += node.getNodeValue();
        } else if (type == ENTITYREF_TYPE) {
          // The content is in the TEXT node under it
          s += adpNode.content();
        } else if (type == CDATA_TYPE) {
          StringBuffer sb = new StringBuffer(
            node.getNodeValue() );
          for (int j=0; j<sb.length(); j++) {
                  if (sb.charAt(j) == '<') {
                     sb.setCharAt(j, '&');
                     sb.insert(j+1, "lt;");
                     j += 3;
                  } else if (sb.charAt(j) == '&') {
                     sb.setCharAt(j, '&');
                     sb.insert(j+1, "amp;");
                     j += 4;
                  }
               }
               s += "<pre>" + sb + "</pre>";
            }
         }
         return s;
      }
      ...
   } // AdapterNode
```

Note: This code collapses EntityRef nodes, as inserted by the JAXP 1.1 parser that ins included in the 1.4 Java platform. With JAXP 1.2, that portion of the code is not necessary because entity references are converted to text nodes by the parser. Other parsers may well insert such nodes, however, so including this code "future proofs" your application, should you use a different parser in the future.

Although this code is not the most efficient that anyone ever wrote, it works and it will do fine for our purposes. In this code, you are recognizing and dealing with the following data types:

Element
For elements with names like the XHTML "em" node, you return the node's content sandwiched between the appropriate `<em>` and `</em>` tags. However, when processing the content for the `slideshow` element, for example, you don't include tags for the `slide` elements it contains so, when returning a node's content, you skip any subelements that are themselves displayed in the tree.

Text
No surprise here. For a text node, you simply return the node's `value`.

Entity Reference
Unlike `CDATA` nodes, Entity References can contain multiple subelements. So the strategy here is to return the concatenation of those subelements.

CDATA

Like a text node, you return the node's `value`. However, since the text in this case may contain angle brackets and ampersands, you need to convert them to a form that displays properly in an HTML pane. Unlike the XML CDATA tag, the HTML `<pre>` tag does not prevent the parsing of character-format tags, break tags and the like. So you have to convert left-angle brackets (<) and ampersands (&) to get them to display properly.

On the other hand, there are quite a few node types you are *not* processing with the code above. It's worth a moment to examine them and understand why:

Attribute

These nodes do not appear in the DOM, but are obtained by invoking `getAttributes` on element nodes.

Entity

These nodes also do not appear in the DOM. They are obtained by invoking `getEntities` on `DocType` nodes.

Processing Instruction

These nodes don't contain displayable data.

Comment

Ditto. Nothing you want to display here.

Document

This is the root node for the DOM. There's no data to display for that.

DocType

The `DocType` node contains the DTD specification, with or without external pointers. It only appears under the root node, and has no data to display in the tree.

Document Fragment

This node is equivalent to a document node. It's a root node that the DOM specification intends for holding intermediate results during cut/paste operations, for example. Like a document node, there's no data to display.

Notation

We're just flat out ignoring this one. These nodes are used to include binary data in the DOM. As discussed earlier in Referencing Binary Entities and Using the DTDHandler and EntityResolver (page 207), the MIME types (in conjunction with namespaces) make a better mechanism for that.

Display the Content in the JTree

With the content-concatenation out of the way, only a few small programming steps remain. The first is to modify `toString` so that it uses the node's content for identifying information. Add the code highlighted below to do that:

```
public class DomEcho extends JPanel
{
  ...
  public class AdapterNode
  {
    ...
    public String toString() {
      ...
      if (! nodeName.startsWith("#")) {
        s += ": " + nodeName;
      }
      if (compress) {
        String t = content().trim();
        int x = t.indexOf(");
        if (x >= 0) t = t.substring(0, x);
        s += " " + t;
        return s;
      }
      if (domNode.getNodeValue() != null) {
        ...
      }
      return s;
    }
```

Wire the JTree to the JEditorPane

Returning now to the app's constructor, create a tree selection listener and use to wire the `JTree` to the `JEditorPane`:

```
public class DomEcho extends JPanel
{
  ...
  public DomEcho()
  {
    ...
    // Build right-side view
    JEditorPane htmlPane = new JEditorPane("text/html","");
    htmlPane.setEditable(false);
    JScrollPane htmlView = new JScrollPane(htmlPane);
    htmlView.setPreferredSize(
      new Dimension( rightWidth, windowHeight ));

      tree.addTreeSelectionListener(
        new TreeSelectionListener() {
          public void valueChanged(TreeSelectionEvent e)
          {
            TreePath p = e.getNewLeadSelectionPath();
            if (p != null) {
             AdapterNode adpNode =
                (AdapterNode)
                  p.getLastPathComponent();
             htmlPane.setText(adpNode.content());
            }
          }
        }
      );
```

Now, when a `JTree` node is selected, it's contents are delivered to the `htmlPane`.

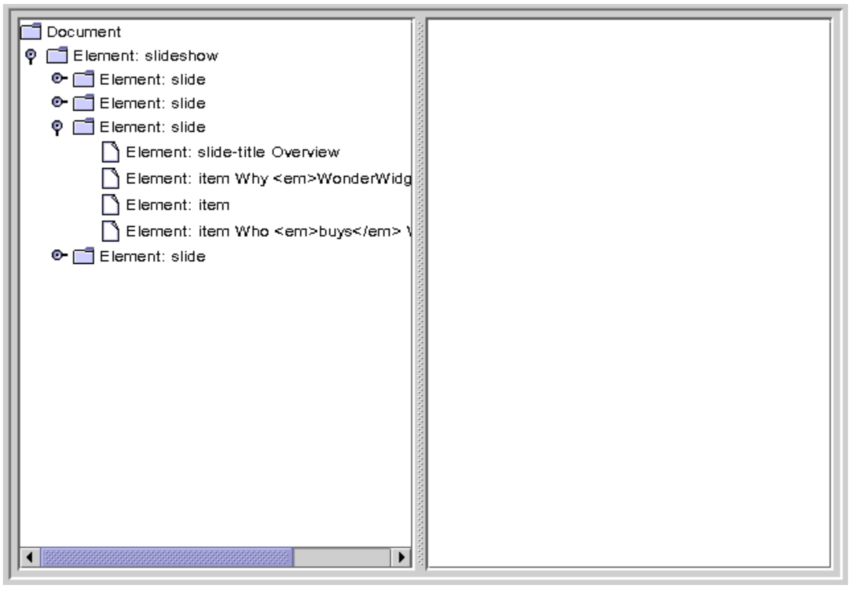
Note: The `TreeSelectionListener` in this example is created using an anonymous inner-class adapter. If you are programming for the 1.1 version of the platform, you'll need to define an external class for this purpose.

If you compile this version of the app, you'll discover immediately that the `htmlPane` needs to be specified as `final` to be referenced in an inner class, so add the keyword highlighted below:

```
public DomEcho04()
{
  ...
  // Build right-side view
  final JEditorPane htmlPane = new
    JEditorPane("text/html","");
  htmlPane.setEditable(false);
  JScrollPane htmlView = new JScrollPane(htmlPane);
  htmlView.setPreferredSize(
    new Dimension( rightWidth, windowHeight ));
```

Run the App

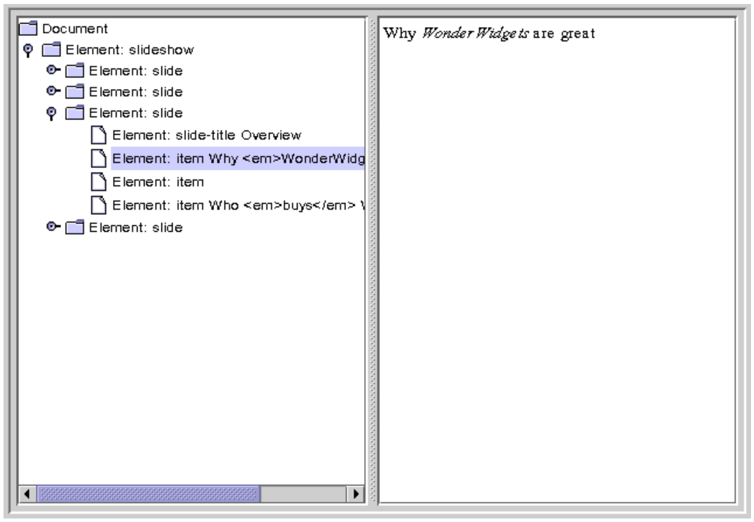
When you compile the application and run it on `slideSample10.xml` (the browsable version is `slideSample10-xml.html`), you get a display like that shown in Figure 7–9. Expanding the hierarchy shows that the `JTree` now includes identifying text for a node whenever possible.



```
Document
  Element: slideshow
    Element: slide
    Element: slide
    Element: slide
      Element: slide-title Overview
      Element: item Why <em>WonderWidg
      Element: item
      Element: item Who <em>buys</em> V
    Element: slide
```

Figure 7–9 Collapsed Hierarchy Showing Text in Nodes

Selecting an item that includes XHTML subelements produces a display like that shown in Figure 7–10:

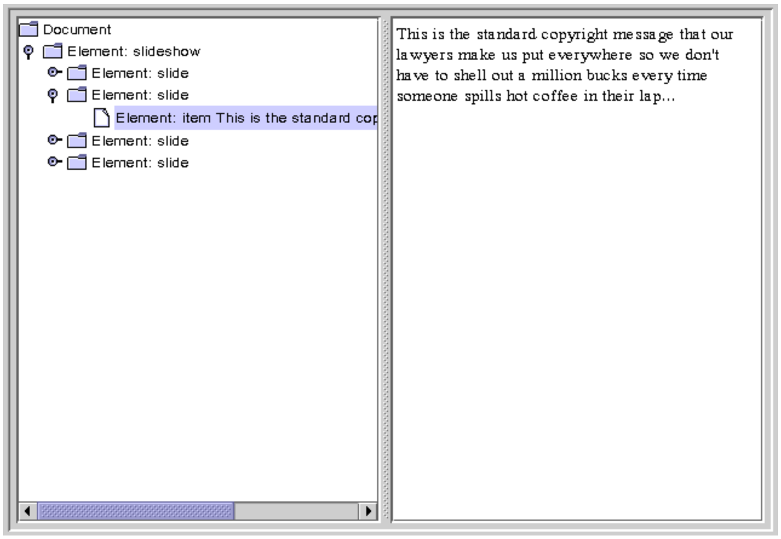


```
Document
  Element: slideshow
    Element: slide
    Element: slide
    Element: slide
      Element: slide-title Overview
      Element: item Why <em>WonderWidg
      Element: item
      Element: item Who <em>buys</em> \
    Element: slide
```

Why *Wonder Widgets* are great

Figure 7–10 Node with `<em>` Tag Selected

Selecting a node that contains an entity reference causes the entity text to be included, as shown in Figure 7–11:

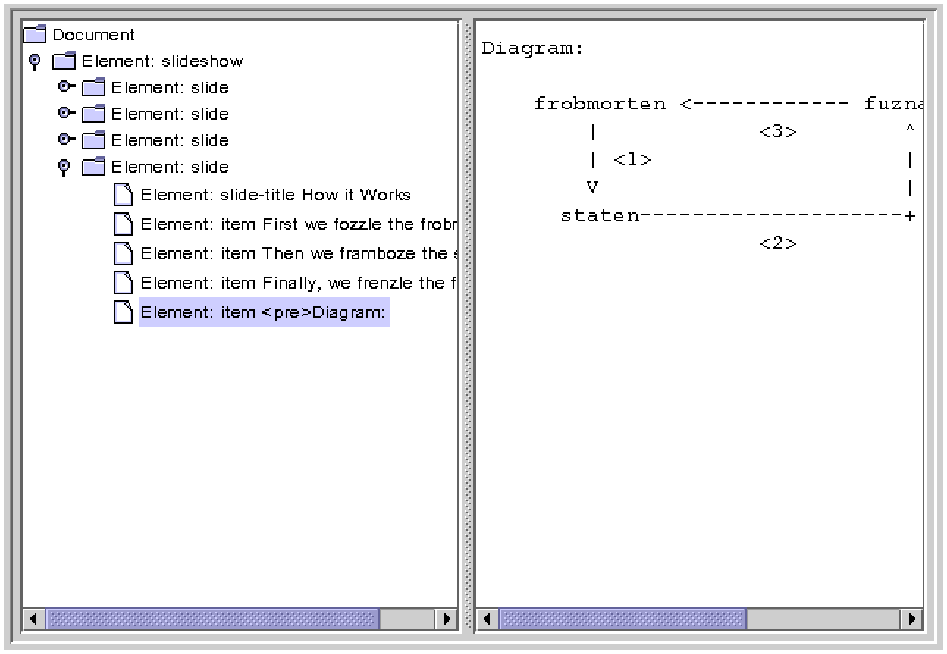


Document
Element: slideshow
Element: slide
Element: slide
Element: item This is the standard cop
Element: slide
Element: slide

This is the standard copyright message that our lawyers make us put everywhere so we don't have to shell out a million bucks every time someone spills hot coffee in their lap...

Figure 7–11 Node with Entity Reference Selected

Finally, selecting a node that includes a CDATA section produces results like those shown in Figure 7–12:



```
Document
  Element: slideshow
    Element: slide
    Element: slide
    Element: slide
    Element: slide
      Element: slide-title How it Works
      Element: item First we fozzle the frobr
      Element: item Then we framboze the s
      Element: item Finally, we frenzle the f
      Element: item <pre>Diagram:
```

```
Diagram:

    frobmorten <------------ fuzna
        |            <3>        ^
        | <1>                   |
        v                       |
      staten--------------------+
                     <2>
```

Figure 7–12 Node with CDATA Component Selected

Extra Credit

Now that you have the application working, here are some ways you might think about extending it in the future:

Use Title Text to Identify Slides
Special case the `slide` element so that the contents of the `title` node is used as the identifying text. When selected, convert the title node's contents to a centered `H1` tag, and ignore the `title` element when constructing the tree.

Convert Item Elements to Lists
Remove `item` elements from the `JTree` and convert them to HTML lists using `<ul>`, `<li>`, `</ul>` tags, including them in the slide's content when the slide is selected.

Handling Modifications

A full discussion of the mechanisms for modifying the `JTree's` underlying data model is beyond the scope of this tutorial. However, a few words on the subject are in order.

Most importantly, note that if you allow the user to modifying the structure by manipulating the `JTree`, you have take the compression into account when you figure out where to apply the change. For example, if you are displaying text in the tree and the user modifies that, the changes would have to be applied to text subelements, and perhaps require a rearrangement of the XHTML subtree.

When you make those changes, you'll need to understand more about the interactions between a `JTree`, it's `TreeModel`, and an underlying data model. That subject is covered in depth in the Swing Connection article, *Understanding the TreeModel* at `http://java.sun.com/products/jfc/tsc/articles/jtree/index.html`.

Finishing Up

You now understand pretty much what there is know about the structure of a DOM, and you know how to adapt a DOM to create a user-friendly display in a `JTree`. It has taken quite a bit of coding, but in return you have obtained valuable tools for exposing a DOM's structure and a template for GUI apps. In the next section, you'll make a couple of minor modifications to the code that turn the application into a vehicle for experimentation, and then experiment with building and manipulating a DOM.

Creating and Manipulating a DOM

By now, you understand the structure of the nodes that make up a DOM. A DOM is actually very easy to create. This section of the DOM tutorial is going to take much less work than anything you've see up to now. All the foregoing work, however, generated the basic understanding that will make this section a piece of cake.

Obtaining a DOM from the Factory

In this version of the application, you're still going to create a document builder factory, but this time you're going to tell it create a new DOM instead of parsing an existing XML document. You'll keep all the existing functionality intact, however, and add the new functionality in such a way that you can "flick a switch" to get back the parsing behavior.

Note: The code discussed in this section is in `DomEcho05.java`.

Modify the Code

Start by turning off the compression feature. As you work with the DOM in this section, you're going to want to see all the nodes:

```
public class DomEcho05  extends JPanel
{
  ...
  ~~boolean compress = true;~~
  boolean compress = false;
```

Next, you need to create a `buildDom` method that creates the `document` object. The easiest way to do that is to create the method and then copy the DOM-construction section from the `main` method to create the `buildDom`. The modifications shown below show you the changes you need to make to make that code suitable for the `buildDom` method.

```
public class DomEcho05  extends JPanel
{
  ...
  public static void makeFrame() {
    ...
  }
  public static void buildDom()
  {
    DocumentBuilderFactory factory =
      DocumentBuilderFactory.newInstance();
    try {
      DocumentBuilder builder =
        factory.newDocumentBuilder();
      ~~document = builder.parse( new File(argv[0]) );~~
      document = builder.newDocument();
    ~~} catch (SAXException sxe) {~~
```

```
          ~~...~~
        } catch (ParserConfigurationException pce) {
          // Parser with specified options can't be built
          pce.printStackTrace();
        ~~} catch (IOException ioe) {~~
          ~~...~~
      }
    }
```

In this code, you replaced the line that does the parsing with one that creates a DOM. Then, since the code is no longer parsing an existing file, you removed exceptions which are no longer thrown: `SAXException` and `IOException`.

And since you are going to be working with `Element` objects, add the statement to import that class at the top of the program:

```
import org.w3c.dom.Document;
import org.w3c.dom.DOMException;
import org.w3c.dom.Element;
```

Create Element and Text Nodes

Now, for your first experiment, add the `Document` operations to create a root node and several children:

```
public class DomEcho05  extends JPanel
{
  ...
  public static void buildDom()
  {
    DocumentBuilderFactory factory =
      DocumentBuilderFactory.newInstance();
    try {
      DocumentBuilder builder =
        factory.newDocumentBuilder();
      document = builder.newDocument();
      // Create from whole cloth
       Element root =
        (Element)
          document.createElement("rootElement");
      document.appendChild(root);
      root.appendChild(
        document.createTextNode("Some") );
      root.appendChild(
        document.createTextNode(" ")    );
      root.appendChild(
            document.createTextNode("text") );
      } catch (ParserConfigurationException pce) {
         // Parser with specified options can't be built
         pce.printStackTrace();
      }
   }
```

Finally, modify the argument-list checking code at the top of the `main` method so you invoke `buildDom` and `makeFrame` instead of generating an error, as shown below:

```
public class DomEcho05  extends JPanel
{
   ...
   public static void main(String argv[])
   {
      if (argv.length != 1) {
         ~~System.err.println("...");~~
         ~~System.exit(1);~~
         buildDom();
         makeFrame();
         return;
      }
```

That’s all there is to it! Now, if you supply an argument the specified file is parsed and, if you don’t, the experimental code that builds a DOM is executed.

Run the App

Compile and run the program with no arguments produces the result shown in Figure 7–13:

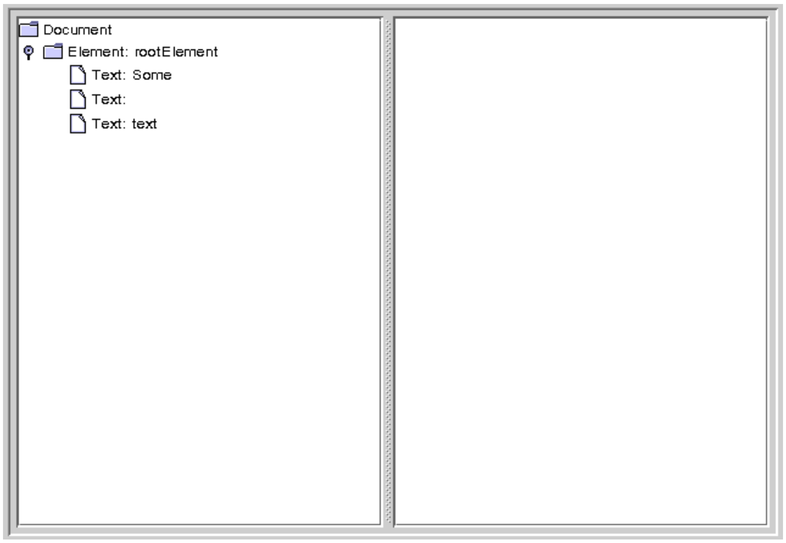


Figure 7–13 Element Node and Text Nodes Created

Normalizing the DOM

In this experiment, you'll manipulate the DOM you created by normalizing it after it has been constructed.

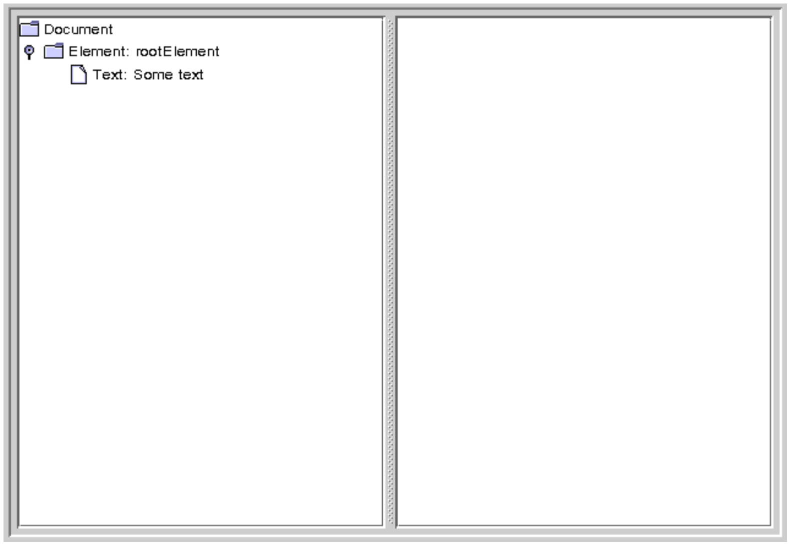
Note: The code discussed in this section is in `DomEcho06.java`.

Add the code highlighted below to normalize the DOM:.

```
public static void buildDom()
{
  DocumentBuilderFactory factory =
     DocumentBuilderFactory.newInstance();
  try {
     ...
     root.appendChild( document.createTextNode("Some") );
     root.appendChild( document.createTextNode(" ")    );
     root.appendChild( document.createTextNode("text") );
     document.getDocumentElement().normalize();

  } catch (ParserConfigurationException pce) {
        ...
```

In this code, `getDocumentElement` returns the document's root node, and the `normalize` operation manipulates the tree under it.

When you compile and run the application now, the result looks like Figure 7–14:



Document
Element: rootElement
Text: Some text

Figure 7–14 Text Nodes Merged After Normalization

Here, you can see that the adjacent text nodes have been combined into a single node. The normalize operation is one that you will typically want to use after making modifications to a DOM, to ensure that the resulting DOM is as compact as possible.

Note: Now that you have this program to experiment with, see what happens to other combinations of `CDATA`, entity references, and text nodes when you normalize the tree.

Other Operations

To complete this section, we'll take a quick look at some of the other operations you might want to apply to a DOM, including:

- Traversing nodes
- Searching for nodes
- Obtaining node content
- Creating attributes
- Removing and changing nodes
- Inserting nodes

Traversing Nodes

The `org.w3c.dom.Node` interface defines a number of methods you can use to traverse nodes, including `getFirstChild`, `getLastChild`, `getNextSibling`, `getPreviousSibling`, and `getParentNode`. Those operations are sufficient to get from anywhere in the tree to any other location in the tree.

Searching for Nodes

However, when you are searching for a node with a particular name, there is a bit more to take into account. Although it is tempting to get the first child and inspect it to see if it is the right one, the search has to account for the fact that the first child in the sublist could be a comment or a processing instruction. If the XML data wasn't validated, it could even be a text node containing ignorable whitespace.

In essence, you need to look through the list of child nodes, ignoring the ones that are of no concern, and examining the ones you care about. Here is an example of the kind of routine you need to write when searching for nodes in a DOM hierarchy. It is presented here in its entirety (complete with comments) so you can use it for a template in your applications.

```
/**
 * Find the named subnode in a node's sublist.
  * <li>Ignores comments and processing instructions.
  * <li>Ignores TEXT nodes (likely to exist and contain
ignorable whitespace,
  *     if not validating.
  * <li>Ignores CDATA nodes and EntityRef nodes.
  * <li>Examines element nodes to find one with the specified
name.
  * </ul>
  * @param name  the tag name for the element to find
  * @param node  the element node to start searching from
  * @return the Node found
  */
public Node findSubNode(String name, Node node) {
  if (node.getNodeType() != Node.ELEMENT_NODE) {
     System.err.println("Error: Search node not of element
type");
     System.exit(22);
  }

  if (! node.hasChildNodes()) return null;

  NodeList list = node.getChildNodes();
  for (int i=0; i < list.getLength(); i++) {
     Node subnode = list.item(i);
     if (subnode.getNodeType() == Node.ELEMENT_NODE) {
        if (subnode.getNodeName() == name) return subnode;
     }
  }
  return null;
}
```

For a deeper explanation of this code, see Increasing the Complexity (page 215) in When to Use DOM.

Note, too, that you can use APIs described in Summary of Lexical Controls (page 250) to modify the kind of DOM the parser constructs. The nice thing about this code, though, is that will work for most any DOM.

Obtaining Node Content

When you want to get the text that a node contains, you once again need to look through the list of child nodes, ignoring entries that are of no concern, and accumulating the text you find in `TEXT` nodes, `CDATA` nodes, and `EntityRef` nodes.

Here is an example of the kind of routine you need to use for that process:

```
/**
  * Return the text that a node contains. This routine:<ul>
  * <li>Ignores comments and processing instructions.
  * <li>Concatenates TEXT nodes, CDATA nodes, and the results of
  *     recursively processing EntityRef nodes.
  * <li>Ignores any element nodes in the sublist.
  *     (Other possible options are to recurse into element
sublists
  *      or throw an exception.)
  * </ul>
  * @param    node  a  DOM node
  * @return   a String representing its contents
  */
public String getText(Node node) {
  StringBuffer result = new StringBuffer();
  if (! node.hasChildNodes()) return "";

  NodeList list = node.getChildNodes();
  for (int i=0; i < list.getLength(); i++) {
    Node subnode = list.item(i);
    if (subnode.getNodeType() == Node.TEXT_NODE) {
      result.append(subnode.getNodeValue());
    }
    else if (subnode.getNodeType() ==
         Node.CDATA_SECTION_NODE)
    {
      result.append(subnode.getNodeValue());
    }
    else if (subnode.getNodeType() ==
         Node.ENTITY_REFERENCE_NODE)
    {
      // Recurse into the subtree for text
      // (and ignore comments)
      result.append(getText(subnode));
    }
  }
  return result.toString();
}
```

For a deeper explanation of this code, see Increasing the Complexity (page 215) in When to Use DOM.

Again, you can simplify this code by using the APIs described in Summary of Lexical Controls (page 250) to modify the kind of DOM the parser constructs. But the nice thing about this code, once again, is that will work for most any DOM.

Creating Attributes

The `org.w3c.dom.Element` interface, which extends Node, defines a `setAttribute` operation, which adds an attribute to that node. (A better name from the

Java platform standpoint would have been addAttribute, since the attribute is not a property of the class, and since a new object is created.)

You can also use the Document's createAttribute operation to create an instance of Attribute, and use an overloaded version of setAttribute to add that.

Removing and Changing Nodes

To remove a node, you use its parent Node's removeChild method. To change it, you can either use the parent node's replaceChild operation or the node's setNodeValue operation.

Inserting Nodes

The important thing to remember when creating new nodes is that when you create an element node, the only data you specify is a name. In effect, that node gives you a hook to hang things on. You "hang an item on the hook" by adding to its list of child nodes. For example, you might add a text node, a CDATA node, or an attribute node. As you build, keep in mind the structure you examined in the exercises you've seen in this tutorial. Remember: Each node in the hierarchy is extremely simple, containing only one data element.

Finishing Up

Congratulations! You've learned how a DOM is structured and how to manipulate it. And you now have a DomEcho application that you can use to display a DOM's structure, condense it down to GUI-compatible dimensions, and experiment with to see how various operations affect the structure. Have fun with it!

Using Namespaces

As you saw previously, one way or another it is necessary to resolve the conflict between the title element defined in slideshow.dtd and the one defined in xhtml.dtd when the same name is used for different purposes. In the previous exercise, you hyphenated the name in order to put it into a different "namespace". In this section, you'll see how to use the XML namespace standard to do the same thing without renaming the element.

The primary goal of the namespace specification is to let the document author tell the parser which DTD or schema to use when parsing a given element. The parser can then consult the appropriate DTD or schema for an element definition. Of course, it is also important to keep the parser from aborting when a "duplicate" definition is found, and yet still generate an error if the document references an element like `title` without *qualifying* it (identifying the DTD or schema to use for the definition).

Note: Namespaces apply to attributes as well as to elements. In this section, we consider only elements. For more information on attributes, consult the namespace specification at `http://www.w3.org/TR/REC-xml-names/`.

Defining a Namespace in a DTD

In a DTD, you define a namespace that an element belongs to by adding an attribute to the element's definition, where the attribute name is `xmlns` ("xml namespace"). For example, you could do that in `slideshow.dtd` by adding an entry like the following in the `title` element's attribute-list definition:

```
<!ELEMENT title (%inline;)*>
<!ATTLIST title
  xmlns CDATA #FIXED "http://www.example.com/slideshow"
>
```

Declaring the attribute as `FIXED` has several important features:

- It prevents the document from specifying any non-matching value for the `xmlns` attribute (as described in Defining Attributes in the DTD).
- The element defined in this DTD is made unique (because the parser understands the `xmlns` attribute), so it does not conflict with an element that has the same name in another DTD. That allows multiple DTDs to use the same element name without generating a parser error.
- When a document specifies the `xmlns` attribute for a tag, the document selects the element definition with a matching attribute.

To be thorough, every element name in your DTD would get the exact same attribute, with the same value. (Here, though, we're only concerned about the `title` element.) Note, too, that you are using a `CDATA` string to supply the URI. In this case, we've specified an URL. But you could also specify a URN, possibly by specifying a prefix like `urn:` instead of `http:`. (URNs are currently being researched. They're not seeing a lot of action at the moment, but that could change in the future.)

Referencing a Namespace

When a document uses an element name that exists in only one of the.DTDs or schemas it references, the name does not need to be qualified. But when an element name that has multiple definitions is used, some sort of qualification is a necessity.

Note: In point of fact, an element name is always qualified by it's *default namespace*, as defined by name of the DTD file it resides in. As long as there as is only one definition for the name, the qualification is implicit.

You qualify a reference to an element name by specifying the `xmlns` attribute, as shown here:

```
<title xmlns="http://www.example.com/slideshow">
  Overview
</title>
```

The specified namespace applies to that element, and to any elements contained within it.

Defining a Namespace Prefix

When you only need one namespace reference, it's not such a big deal. But when you need to make the same reference several times, adding `xmlns` attributes becomes unwieldy. It also makes it harder to change the name of the namespace at a later date.

The alternative is to define a *namespace prefix*, which as simple as specifying xmlns, a colon (:) and the prefix name before the attribute value, as shown here:

```
<SL:slideshow xmlns:SL='http:/www.example.com/slideshow'
     ...>
  ...
</SL:slideshow>
```

This definition sets up SL as a prefix that can be used to qualify the current element name and any element within it. Since the prefix can be used on any of the contained elements, it makes the most sense to define it on the XML document's root element, as shown here.

Note: The namespace URI can contain characters which are not valid in an XML name, so it cannot be used as a prefix directly. The prefix definition associates an XML name with the URI, which allows the prefix name to be used instead. It also makes it easier to change references to the URI in the future.

When the prefix is used to qualify an element name, the end-tag also includes the prefix, as highlighted here:

```
<SL:slideshow xmlns:SL='http:/www.example.com/slideshow'
        ...>
  ...
  <slide>
     <SL:title>Overview</SL:title>
  </slide>
  ...
</SL:slideshow>
```

Finally, note that multiple prefixes can be defined in the same element, as shown here:

```
<SL:slideshow xmlns:SL='http:/www.example.com/slideshow'
        xmlns:xhtml='urn:...'>
  ...
</SL:slideshow>
```

With this kind of arrangement, all of the prefix definitions are together in one place, and you can use them anywhere they are needed in the document. This example also suggests the use of URN to define the xhtml prefix, instead of an URL. That definition would conceivably allow the application to reference a local copy of the XHTML DTD or some mirrored version, with a potentially beneficial impact on performance.

Validating with XML Schema

Now that you understand more about namespaces, you're ready to take a deeper look at the process of XML Schema validation. Although a full treatment of

XML Schema is beyond the scope of this tutorial, this section will show you the steps you need to take to validate an XML document using an XML Schema definition. (To learn more about XML Schema, you can review the online tutorial, *XML Schema Part 0: Primer*, at http://www.w3.org/TR/xmlschema-0/. You can also examine the sample programs that are part of the JAXP download. They use a simple XML Schema definition to validate personnel data stored in an XML file.)

Note: There are multiple schema-definition languages, including RELAX NG, Schematron, and the W3C “XML Schema” standard. (Even a DTD qualifies as a “schema”, although it is the only one that does not use XML syntax to describe schema constraints.) However, “XML Schema” presents us with a terminology challenge. While the phrase “XML Schema schema” would be precise, we’ll use the phrase “XML Schema definition” to avoid the appearance of redundancy.

At the end of this section, you’ll also learn how to use an XML Schema definition to validate a document that contains elements from multiple namespaces.

Overview of the Validation Process

To be notified of validation errors in an XML document,

1. The factory must configured, and the appropriate error handler set.
2. The document must be associated with at least one schema, and possibly more.

Configuring the DocumentBuilder Factory

It’s helpful to start by defining the constants you’ll use when configuring the factory. (These are same constants you define when using XML Schema for SAX parsing.)

```
static final String JAXP_SCHEMA_LANGUAGE =
     "http://java.sun.com/xml/jaxp/properties/schemaLanguage";

static final String W3C_XML_SCHEMA =
     "http://www.w3.org/2001/XMLSchema";
```

Next, you need to configure `DocumentBuilderFactory` to generate a namespace-aware, validating parser that uses XML Schema:

```
...
  DocumentBuilderFactory factory =
       DocumentBuilderFactory.newInstance()
  factory.setNamespaceAware(true);
  factory.setValidating(true);
try {
  factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
}
catch (IllegalArgumentException x) {
  // Happens if the parser does not support JAXP 1.2
  ...
}
```

Since JAXP-compliant parsers are not namespace-aware by default, it is necessary to set the property for schema validation to work. You also set a factory attribute specify the parser language to use. (For SAX parsing, on the other hand, you set a property on the parser generated by the factory.)

Associating a Document with a Schema

Now that the program is ready to validate with an XML Schema definition, it is only necessary to ensure that the XML document is associated with (at least) one. There are two ways to do that:

1. With a schema declaration in the XML document.
2. By specifying the schema(s) to use in the application.

Note: When the application specifies the schema(s) to use, it overrides any schema declarations in the document.

To specify the schema definition in the document, you would create XML like this:

```
<documentRoot
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation='YourSchemaDefinition.xsd'
>
  ...
```

The first attribute defines the XML NameSpace (xmlns) prefix, “xsi”, where “xsi” stands for “XML Schema Instance”. The second line specifies the schema to use for elements in the document that do *not* have a namespace prefix — that is, for the elements you typically define in any simple, uncomplicated XML document. (You’ll see how to deal with multiple namespaces in the next section.)

To can also specify the schema file in the application, like this:

```
static final String schemaSource = "YourSchemaDefinition.xsd";
static final String JAXP_SCHEMA_SOURCE =
    "http://java.sun.com/xml/jaxp/properties/schemaSource";
...
DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance()
...
factory.setAttribute(JAXP_SCHEMA_SOURCE,
    new File(schemaSource));
```

Here, too, there are mechanisms at your disposal that will let you specify multiple schemas. We’ll take a look at those next.

Validating with Multiple Namespaces

Namespaces let you combine elements that serve different purposes in the same document, without having to worry about overlapping names.

Note: The material discussed in this section also applies to validating when using the SAX parser. You’re seeing it here, because at this point you’ve learned enough about namespaces for the discussion to make sense.

To contrive an example, consider an XML data set that keeps track of personnel data. The data set may include information from the w2 tax form, as well as information from the employee’s hiring form, with both elements named `<form>` in their respective schemas.

If a prefix is defined for the “tax” namespace, and another prefix defined for the “hiring” namespace, then the personnel data could include segments like this:

```
<employee id=”...”>
  <name>....</name>
  <tax:form>
     ...w2 tax form data...
```

```
    </tax:form>
    <hiring:form>
      ...employment history, etc....
    </hiring:form>
  </employee>
```

The contents of the `tax:form` element would obviously be different from the contents of the `hiring:form`, and would have to be validated differently.

Note, too, that there is a “default” namespace in this example, that the unqualified element names `employee` and `name` belong to. For the document to be properly validated, the schema for that namespace must be declared, as well as the schemas for the `tax` and `hiring` namespaces.

Note: The “default” namespace is actually a *specific* namespace. It is defined as the “namespace that has no name”. So you can’t simply use one namespace as your default this week, and another namespace as the default later on. This “unnamed namespace” or “null namespace” is like the number zero. It doesn’t have any value, to speak of (no name), but it is still precisely defined. So a namespace that does have a name can never be used as the “default” namespace.

When parsed, each element in the data set will be validated against the appropriate schema, as long as those schemas have been declared. Again, the schemas can either be declared as part of the XML data set, or in the program. (It is also possible to mix the declarations. In general, though, it is a good idea to keep all of the declarations together in one place.)

Declaring the Schemas in the XML Data Set

To declare the schemas to use for the example above in the data set, the XML code would look something like this:

```
  <documentRoot
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="employeeDatabase.xsd"
    xsi:schemaLocation=
      ”http://www.irs.gov/ fullpath/w2TaxForm.xsd
       http://www.ourcompany.com/ relpath/hiringForm.xsd“
    xmlns:tax="http://www.irs.gov/"
    xmlns:hiring="http://www.ourcompany.com/"
  >
    ...
```

The noNamespaceSchemaLocation declaration is something you've seen before, as are the last two entries, which define the namespace prefixes tax and hiring. What's new is the entry in the middle, which defines the locations of the schemas to use for each namespace referenced in the document.

The xsi:schemaLocation declaration consists of entry pairs, where the first entry in each pair is a fully qualified URI that specifies the namespace, and the second entry contains a full path or a relative path to the schema definition. (In general, fully qualified paths are recommended. That way, only one copy of the schema will tend to exist.)

Of particular note is the fact that the namespace prefixes cannot be used when defining the schema locations. The xsi:schemaLocation declaration only understands namespace names, not prefixes.

Declaring the Schemas in the Application

To declare the equivalent schemas in the application, the code would look something like this:

```
static final String employeeSchema = "employeeDatabase.xsd";
static final String taxSchema = "w2TaxForm.xsd";
static final String hiringSchema = "hiringForm.xsd";

static final String[] schemas = {
     employeeSchema,
     taxSchema,
     hiringSchema,
     };

static final String JAXP_SCHEMA_SOURCE =
  "http://java.sun.com/xml/jaxp/properties/schemaSource";

...
DocumentBuilderFactory factory =
     DocumentBuilderFactory.newInstance()
...
factory.setAttribute(JAXP_SCHEMA_SOURCE, schemas);
```

Here, the array of strings that points to the schema definitions (.xsd files) is passed as the argument to factory.setAttribute method. Note the differences from when you were declaring the schemas to use as part of the XML data set:

- There is no special declaration for the "default" (unnamed) schema.

- You don't specify the namespace name. Instead, you only give pointers to the `.xsd` files.

To make the namespace assignments, the parser reads the .xsd files, and finds in them the name of the *target namespace* they apply to. Since the files are specified with URIs, the parser can use an EntityResolver (if one has been defined) to find a local copy of the schema.

If the schema definition does not define a target namespace, then it applies to the "default" (unnamed, or null) namespace. So, in the example above, you would expect to see these target namespace declarations in the schemas:

- `employeeDatabase.xsd` — none
- `w2TaxForm.xsd` — `http://www.irs.gov/`
- `hiringForm.xsd` — `http://www.ourcompany.com`

At this point, you have seen two possible values for the schema source property when invoking the `factory.setAttribute()` method, a File object in `factory.setAttribute(JAXP_SCHEMA_SOURCE, new File(schemaSource)).` and an array of strings in `factory.setAttribute(JAXP_SCHEMA_SOURCE, schemas)`. Here is a complete list of the possible values for that argument:

- String that points to the URI of the schema
- InputStream with the contents of the schema
- SAX InputSource
- File
- an array of Objects, each of which is one of the types defined above.

Note: An array of Objects can be used only when the schema language (like `http://java.sun.com/xml/jaxp/properties/schemaLanguage`) has the ability to assemble a schema at runtime. Also: When an array of Objects is passed it is illegal to have two schemas that share the same namespace.

Further Information

For further information on the TreeModel, see:

- *Understanding the TreeModel*: `http://java.sun.com/products/jfc/tsc/articles/jtree/index.html`

For further information on the W3C Document Object Model (DOM), see:

- The DOM standard page: `http://www.w3.org/DOM/`

For more information on schema-based validation mechanisms, see:

- The W3C standard validation mechanism, XML Schema: `http://www.w3.org/XML/Schema`
- RELAX NG's regular-expression based validation mechanism: `http://www.oasis-open.org/committees/relax-ng/`
- Schematron's assertion-based validation mechansim: `http://www.ascc.net/xml/resource/schematron/schematron.html`

8

XML Stylesheet Language for Transformations

Eric Armstrong

THE XML Stylesheet Language for Transformations (XSLT) defines mechanisms for addressing XML data (XPath) and for specifying transformations on the data, in order to convert it into other forms. JAXP includes two implementations of XSLT, an interpreting version (Xalan) and a compiling version (XSLTC) that lets you save pre-compiled versions of desired transformations as *translets*, for the most efficient runtime processing later on.

In this chapter, you'll learn how to use both Xalan and XSLTC. You'll write out a Document Object Model (DOM) as an XML file, and you'll see how to generate a DOM from an arbitrary data file in order to convert it to XML. Finally, you'll convert XML data into a different form, unlocking the mysteries of the XPath addressing mechanism along the way.

Note: The examples in this chapter can be found in `<JWSDP_HOME>/docs/tutorial/examples/jaxp/xslt/samples`.

Introducing XSLT and XPath

The XML Stylesheet Language (XSL) has three major subcomponents:

XSL-FO
The “flow object” standard. By far the largest subcomponent, this standard gives mechanisms for describing font sizes, page layouts, and how information “flows” from one page to another. This subcomponent is *not* covered by JAXP, nor is it included in this tutorial.

XSLT
This is the transformation language, which lets you define a transformation from XML into some other format. For example, you might use XSLT to produce HTML, or a different XML structure. You could even use it to produce plain text or to put the information in some other document format. (And as you’ll see in Generating XML from an Arbitrary Data Structure (page 312), a clever application can press it into service to manipulate non-XML data, as well.)

XPath
At bottom, XSLT is a language that lets you specify what sorts of things to do when a particular element is encountered. But to write a program for different parts of an XML data structure, you need to be able to specify the part of the structure you are talking about at any given time. XPath is that specification language. It is an addressing mechanism that lets you specify a path to an element so that, for example, `<article><title>` can be distinguished from `<person><title>`. That way, you can describe different kinds of translations for the different `<title>` elements.

The remainder of this section describes the packages that make up the JAXP Transformation APIs. It then discusses the factory configuration parameters you use to select the Xalan or XSLTC transformation engine.

The JAXP Transformation Packages

Here is a description of the packages that make up the JAXP Transformation APIs:

`javax.xml.transform`
This package defines the factory class you use to get a `Transformer` object. You then configure the transformer with input (Source) and output (Result) objects, and invoke its `transform()` method to make the transformation

happen. The source and result objects are created using classes from one of the other three packages.

(Whether you get the Xalan interpreting transformer or the XSLTC compiling transformer is determined by factory configuration settings, which will be discussed momentarily.)

`javax.xml.transform.dom`
: Defines the `DOMSource` and `DOMResult` classes that let you use a DOM as an input to or output from a transformation.

`javax.xml.transform.sax`
: Defines the `SAXSource` and `SAXResult` classes that let you use a SAX event generator as input to a transformation, or deliver SAX events as output to a SAX event processor.

`javax.xml.transform.stream`
: Defines the `StreamSource` and `StreamResult` classes that let you use an I/O stream as an input to or output from a transformation.

Choosing the Transformation Engine

This section provides the information you need to help you choose between the interpreting transformer (Xalan) and the compiling transformer (XSLTC).

Performance Considerations

For a single-pass translation, the interpreting transformer (Xalan) tends to be slightly faster than the compiling transformer (XSLTC), because it isn't generating and saving the byte-codes in the small Java classes that are run as translets.

But when a transformation will be used multiple times, it makes sense to use the XSLTC transformation engine because, in such settings, XSLTC is the clear winner when it comes to memory requirements and performance.

An XSLTC translet tends to be small, because it implements only those translations that the stylesheet actually performs. And it tends to be fast, both because it is smaller and because the lexical handling necessary to interpret the stylesheet has already been performed. Finally, translets tends to load faster and generally be more sparing of system resources, due to their small size.

For example, a servlet that will be running for long periods of time tends to benefit by using XSLTC. Similarly, a transformation that is run from the command line tends to run faster when XSLTC is used. You'll see more about that process in Transforming from the Command Line (page 351).

In addition to making it possible to cache translets, XSLTC provides a number of other options to help you maximize performance:

- Control of inlining

  By default, XSLTC "inlines" transformation code, which means that the code responsible for translating an element contains the transformation code for all possible subelements of that element.

  For small and medium-size stylesheets, that implementation produces the fastest possible code. However, complex stylesheets tend to produce translets that are extremely large.

  To solve that problem, XSLTC lets you disable inlining. To do that, you use the `-n` option when compiling XSLTC translets from the command line. When generating an XSLTC transformer using a JAXP factory class, you use the factory's `setAttribute()` method to set the "`disable-inlining`" `feature` with code like this:

  ```
  TransformerFactory tf = new TransformerFactory();
  tf.setAttribute("disable-inlining", Boolean.TRUE);
  ```

- Document-model caching

  When XSLTC operates on XML data, it creates it's own internal Document Object Model (something like the W3C DOM you've already seen, only simpler). Since the construction of the document model takes time, XSLTC provides a way to cache the model, to help speed up subsequent transformations.

  That feature can come in handy in a servlet that serves up XML documents, for example. If a transform converts them to HTML when they are accessed on the Web, then caching the in-memory representation of the document can have a potentially large impact on performance. Here is a sample of the code you would use:

  ```
  final SAXParser parser = factory.newSAXParser();
  final XMLReader reader = parser.getXMLReader();

  XSLTCSource source = new XSLTCSource();
  source.build(reader, xmlfile);
  ```

The source object can then be reused in multiple transformations, without having to re-read the file.

- Caching of compiled stylesheets

  XSLTC also lets you save compiled versions of stylesheets, so you can use them to create multiple Transformer objects more rapidly. For example, that kind of capability can improve the startup time of a multi-threaded servlet. If the servlet generates a hundred threads to service input requests, it can compile the stylesheet once and then use the compiled version to generate a transformer for each thread.

  Precompiled stylesheets are stored in Templates objects. When you create a Transformer object directly (without using a Templates object), you use code like this:

```
TransformerFactory factory =
    TransformerFactory.newInstance();
Transformer xformer = factory.newTransformer(myStyleSheet);
xformer.transform(myXmlInput,
    new StreamResult(System.out));
```

  But you can also create an intermediate Templates object that you can save and reuse, like this:

```
TransformerFactory factory =
    TransformerFactory.newInstance();
Templates templates = factory.newTemplates(myStyleSheet);
Transformer xformer = templates.newTransformer();
xformer.transform(myXmlInput,
    new StreamResult(System.out));
```

Note: There are also rules for things to do and things to avoid when designing your stylesheets, in order to get maximum performance with XSLT. For more information on that subject, see http://xml.apache.org/xalan-j/xsltc/xsltc_performance.html.

Functionality Considerations

While XSLTC tends to be a higher performance choice for many applications, Xalan has some advantages in functionality. Among those advantages are the support for the standard query language, SQL.

Making Your Choice

Whether you get the Xalan or XSLTC transformation engine is determined by factory configuration settings. By default, the JAXP factory creates a Xalan transformer. To get an XSLTC transformer, the preferred method is to set the `TransformationFactory` system property like this:

```
javax.xml.transform.TransformerFactory=
      org.apache.xalan.xsltc.trax.TransformerFactoryImpl
```

At times, though, it is not possible to set a system property — for example, because the application is a servlet, and changing the system property would affect other servlets running in the same container. In that case, you can instantiate the XSLTC transformation engine directly, with a command like this:

```
new org.apache.xalan.xsltc.trax.TransformerFactoryImpl(..)
```

You could also pass the factory value to the application, and use the ClassLoader to create an instance of it at runtime.

Note: To explicitly specify the Xalan transformer, you would use the value `org.apache.xalan.processor.TransformerFactoryImpl`, instead of `org.apache.xalan.xsltc.trax.TransformerFactoryImpl`.

There is also a “smart transformer” that uses the Xalan transform engine when you generate `Transformer` objects, and the XSLTC transform engine when you generate intermediate `Templates` objects. To get an instance of the smart transformer, use the value `org.apache.xalan.xsltc.trax.SmartTransformerImpl` either to set the transformer factory system property or use that class to instantiate a parser directly.

How XPath Works

The XPath specification is the foundation for a variety of specifications, including XSLT and linking/addressing specifications like XPointer. So an understanding of XPath is fundamental to a lot of advanced XML usage. This section provides a thorough introduction to XPATH in the context of XSLT, so you can refer to it as needed later on.

Note: In this tutorial, you won't actually use XPath until you get to the end of this section, Transforming XML Data with XSLT (page 327). So, if you like, you can skip this section and go on ahead to the next section, Writing Out a DOM as an XML File (page 305). (When you get to the end of that section, there will be a note that refers you back here, so you don't forget!)

XPATH Expressions

In general, an XPath expression specifies a *pattern* that selects a set of XML nodes. XSLT templates then use those patterns when applying transformations. (XPointer, on the other hand, adds mechanisms for defining a *point* or a *range*, so that XPath expressions can be used for addressing.)

The nodes in an XPath expression refer to more than just elements. They also refer to text and attributes, among other things. In fact, the XPath specification defines an abstract document model that defines seven different kinds of nodes:

- root
- element
- text
- attribute
- comment
- processing instruction
- namespace

Note: The root element of the XML data is modeled by an *element* node. The XPath root node contains the document's root element, as well as other information relating to the document.

The XSLT/XPath Data Model

Like the DOM, the XSLT/XPath data model consists of a tree containing a variety of nodes. Under any given element node, there are text nodes, attribute nodes, element nodes, comment nodes, and processing instruction nodes.

In this abstract model, syntactic distinctions disappear, and you are left with a normalized view of the data. In a text node, for example, it makes no difference whether the text was defined in a CDATA section, or if it included entity references. The text node will consist of normalized data, as it exists after all parsing is complete. So the text will contain a < character, regardless of whether an entity reference like `&lt;` or a CDATA section was used to include it. (Similarly, the text will contain an & character, regardless of whether it was delivered using `&amp;` or it was in a CDATA section.)

In this section of the tutorial, we'll deal mostly with element nodes and text nodes. For the other addressing mechanisms, see the XPath Specification.

Templates and Contexts

An XSLT *template* is a set of formatting instructions that apply to the nodes selected by an XPATH expression. In an stylesheet, a XSLT template would look something like this:

```
<xsl:template match="//LIST">
    ...
</xsl:template>
```

The expression `//LIST` selects the set of `LIST` nodes from the input stream. Additional instructions within the template tell the system what to do with them.

The set of nodes selected by such an expression defines the *context* in which other expressions in the template are evaluated. That context can be considered as the whole set — for example, when determining the number of the nodes it contains.

The context can also be considered as a single member of the set, as each member is processed one by one. For example, inside of the `LIST`-processing template, the expression `@type` refers to the `type` attribute of the current `LIST` node. (Similarly, the expression `@*` refers to all of attributes for the current `LIST` element.)

Basic XPath Addressing

An XML document is a tree-structured (hierarchical) collection of nodes. As with a hierarchical directory structure, it is useful to specify a *path* that points a particular node in the hierarchy. (Hence the name of the specification: XPath.) In fact, much of the notation of directory paths is carried over intact:

- The forward slash `/` is used as a path separator.
- An absolute path from the root of the document starts with a /.
- A relative path from a given location starts with anything else.
- A double period `..` indicates the parent of the current node.
- A single period `.` indicates the current node.

For example, In an XHTML document (an XML document that looks like HTML, but which is *well-formed* according to XML rules) the path `/h1/h2/` would indicate an `h2` element under an `h1`. (Recall that in XML, element names are case sensitive, so this kind of specification works much better in XHTML than it would in plain HTML, because HTML is case-insensitive.)

In a pattern-matching specification like XSLT, the specification `/h1/h2` selects *all* `h2` elements that lie under an `h1` element. To select a specific `h2` element, square brackets `[]` are used for indexing (like those used for arrays). The path `/h1[4]/h2[5]` would therefore select the fifth `h2` element under the fourth `h1` element.

Note: In XHTML, all element names are in lowercase. That is a fairly common convention for XML documents. However, uppercase names are easier to read in a tutorial like this one. So, for the remainder of the XSLT tutorial, all XML element names will be in uppercase. (Attribute names, on the other hand, will remain in lowercase.)

A name specified in an XPath expression refers to an element. For example, "`h1`" in `/h1/h2` refers to an `h1` element. To refer to an attribute, you prefix the attribute name with an `@` sign. For example, `@type` refers to the `type` attribute of an element. Assuming you have an XML document with `LIST` elements, for example, the expression `LIST/@type` selects the `type` attribute of the `LIST` element.

Note: Since the expression does not begin with /, the reference specifies a `list` node relative to the current context—whatever position in the document that happens to be.

Basic XPath Expressions

The full range of XPath expressions takes advantage of the wildcards, operators, and functions that XPath defines. You'll be learning more about those shortly. Here, we'll take a look at a couple of the most common XPath expressions, simply to introduce them.

The expression `@type="unordered"` specifies an attribute named `type` whose value is "`unordered`". And you already know that an expression like `LIST/@type` specifies the `type` attribute of a `LIST` element.

You can combine those two notations to get something interesting! In XPath, the square-bracket notation (`[]`) normally associated with indexing is extended to specify *selection criteria*. So the expression `LIST[@type="unordered"]` selects all `LIST` elements whose `type` value is "`unordered`".

Similar expressions exist for elements, where each element has an associated *string-value*. (You'll see how the string-value is determined for a complicated element in a little while. For now, we'll stick with simple elements that have a single text string.)

Suppose you model what's going on in your organization with an XML structure that consists of `PROJECT` elements and `ACTIVITY` elements that have a text string with the project name, multiple `PERSON` elements to list the people involved and, optionally, a `STATUS` element that records the project status. Here are some more examples that use the extended square-bracket notation:

- `/PROJECT[.="MyProject"]`—selects a `PROJECT` named `"MyProject"`.
- `/PROJECT[STATUS]`—selects all projects that have a `STATUS` child element.
- `/PROJECT[STATUS="Critical"]`—selects all projects that have a `STATUS` child element with the string-value "`Critical`".

Combining Index Addresses

The XPath specification defines quite a few addressing mechanisms, and they can be combined in many different ways. As a result, XPath delivers a lot of expressive power for a relatively simple specification. This section illustrates two more interesting combinations:

- `LIST[@type="ordered"][3]`—selects all `LIST` elements of type "`ordered`", and returns the third.

- LIST[3][@type="ordered"]—selects the third LIST element, but only if it is of type "ordered".

Note: Many more combinations of address operators are listed in section 2.5 of the XPath Specification. This is arguably the most useful section of the spec for defining an XSLT transform.

Wildcards

By definition, an unqualified XPath expression selects a set of XML nodes that matches that specified pattern. For example, /HEAD matches all top-level HEAD entries, while /HEAD[1] matches only the first. Table 8–1 lists the wildcards that can be used in XPath expressions to broaden the scope of the pattern matching.

Table 8–1 XPath Wildcard

Wildcard	Meaning
*	Matches any element node (not attributes or text).
node()	Matches any node of any kind: element node, text node, attribute node, processing instruction node, namespace node, or comment node.
@*	Matches any attribute node.

In the project database example, for instance, /*/PERSON[.="Fred"] matches any PROJECT or ACTIVITY element that includes Fred.

Extended-Path Addressing

So far, all of the patterns we've seen have specified an exact number of levels in the hierarchy. For example, /HEAD specifies any HEAD element at the first level in the hierarchy, while /*/* specifies any element at the second level in the hierarchy. To specify an indeterminate level in the hierarchy, use a double forward slash (//). For example, the XPath expression //PARA selects all paragraph elements in a document, wherever they may be found.

The // pattern can also be used within a path. So the expression /HEAD/LIST//PARA indicates all paragraph elements in a subtree that begins from /HEAD/LIST.

XPath Data Types and Operators

XPath expressions yield either a set of nodes, a string, a boolean (true/false value), or a number. Table 8–2 lists the operators that can be used in an Xpath expression

Table 8–2 XPath Operators

Operator	**Meaning**
|	Alternative. For example, PARA|LIST selects all PARA and LIST elements.
or, and	Returns the or/and of two boolean values.
=, !=	Equal or not equal, for booleans, strings, and numbers.
<, >, <=, >=	Less than, greater than, less than or equal to, greater than or equal to—for numbers.
+, -, *, div, mod	Add, subtract, multiply, floating-point divide, and modulus (remainder) operations (e.g. 6 mod 4 = 2)

Finally, expressions can be grouped in parentheses, so you don't have to worry about operator precedence.

Note: "Operator precedence" is a term that answers the question, "If you specify a + b * c, does that mean (a+b) * c or a + (b*c)?". (The operator precedence is roughly the same as that shown in the table.)

String-Value of an Element

Before continuing, it's worthwhile to understand how the string-value of a more complex element is determined. We'll do that now.

The string-value of an element is the concatenation of all descendent text nodes, no matter how deep. So, for a “mixed-model” XML data element like this:

```
<PARA>This paragraph contains a <B>bold</B> word</PARA>
```

The string-value of `<PARA>` is “This paragraph contains a bold word”. In particular, note that `<B>` is a child of `<PARA>` and that the text contained in all children is concatenated to form the string-value.

Also, it is worth understanding that the text in the abstract data model defined by XPath is fully normalized. So whether the XML structure contains the entity reference `&lt;` or “<” in a CDATA section, the element’s string-value will contain the “<” character. Therefore, when generating HTML or XML with an XSLT stylesheet, occurrences of “<” will have to be converted to `&lt;` or enclosed in a CDATA section. Similarly, occurrences of “&” will need to be converted to `&amp;`.

XPath Functions

This section ends with an overview of the XPath functions. You can use XPath functions to select a collection of nodes in the same way that you would use an an element specification like those you have already seen. Other functions return a string, a number, or a boolean value. For example, the expression `/PROJECT/text()` gets the string-value of `PROJECT` nodes.

Many functions depend on the current context. In the example above, the *context* for each invocation of the `text()` function is the `PROJECT` node that is currently selected.

There are many XPath functions—too many to describe in detail here. This section provides a quick listing that shows the available XPath functions, along with a summary of what they do.

Note: Skim the list of functions to get an idea of what’s there. For more information, see Section 4 of the XPath Specification.

Node-set functions

Many XPath expressions select a set of nodes. In essence, they return a *node-set*. One function does that, too.

- `id(...)`—returns the node with the specified id.

(Elements only have an ID when the document has a DTD, which specifies which attribute has the `ID` type.)

Positional functions

These functions return positionally-based numeric values.

- `last()`—returns the index of the last element.

  For example: `/HEAD[last()]` selects the last `HEAD` element.

- `position()`—returns the index position.

  For example: `/HEAD[position() <= 5]` selects the first five `HEAD` elements

- `count(...)`—returns the count of elements.

  For example: `/HEAD[count(HEAD)=0]` selects all `HEAD` elements that have no subheads.

String functions

These functions operate on or return strings.

- `concat(`*`string`*`, `*`string`*`, ...)`—concatenates the string values
- `starts-with(`*`string1`*`, `*`string2`*`)`—returns true if *`string1`* starts with *`string2`*
- `contains(`*`string1`*`, `*`string2`*`)`—returns true if *`string1`* contains *`string2`*
- `substring-before(`*`string1`*`, `*`string2`*`)`—returns the start of *`string1`* before *`string2`* occurs in it
- `substring-after(`*`string1`*`, `*`string2`*`)`—returns the remainder of *`string1`* after *`string2`* occurs in it
- `substring(`*`string`*`, `*`idx`*`)`—returns the substring from the index position to the end, where the index of the first char = 1
- `substring(`*`string`*`, `*`idx`*`, `*`len`*`)`—returns the substring from the index position, of the specified length
- `string-length()`—returns the size of the context-node's string-value

  The *context node* is the currently selected node — the node that was selected by an XPath expression in which a function like `string-length()` is applied.
- `string-length(`*`string`*`)`—returns the size of the specified string
- `normalize-space()`—returns the normalized string-value of the current node (no leading or trailing whitespace, and sequences of whitespace characters converted to a single space)
- `normalize-space(`*`string`*`)`—returns the normalized string-value of the specified string
- `translate(`*`string1`*`, `*`string2`*`, `*`string3`*`)`—converts *`string1`*, replacing occurrences of characters in *`string2`* with the corresponding character from *`string3`*

Note: XPath defines 3 ways to get the text of an element: `text()`, `string(object)`, and the string-value implied by an element name in an expression like this: `/PROJECT[PERSON="Fred"]`.

Boolean functions

These functions operate on or return boolean values:

- `not(...)`—negates the specified boolean value
- `true()`—returns true
- `false()`—returns false
- `lang(`*`string`*`)`—returns true if the language of the context node (specified by `xml:Lang` attributes) is the same as (or a sublanguage of) the specified language. For example: `Lang("en")` is true for `<PARA_xml:Lang="en">...</PARA>`

Numeric functions

These functions operate on or return numeric values.

- `sum(...)`—returns the sum of the numeric value of each node in the specified node-set
- `floor(`*`N`*`)`—returns the largest integer that is not greater than *`N`*
- `ceiling(`*`N`*`)`—returns the smallest integer that is greater than *`N`*
- `round(`*`N`*`)`—returns the integer that is closest to *`N`*

Conversion functions

These functions convert one data type to another.

- `string(...)`—returns the string value of a number, boolean, or node-set
- `boolean(...)`—returns a boolean value for a number, string, or node-set (a non-zero number, a non-empty node-set, and a non-empty string are all true)
- `number(...)`—returns the numeric value of a boolean, string, or node-set (true is 1, false is 0, a string containing a number becomes that number, the string-value of a node-set is converted to a number)

Namespace functions

These functions let you determine the namespace characteristics of a node.

- `local-name()`—returns the name of the current node, minus the namespace prefix
- `local-name(...)`—returns the name of the first node in the specified node set, minus the namespace prefix
- `namespace-uri()`—returns the namespace URI from the current node
- `namespace-uri(...)`—returns the namespace URI from the first node in the specified node set
- `name()`—returns the expanded name (URI plus local name) of the current node
- `name(...)`—returns the expanded name (URI plus local name) of the first node in the specified node set

Summary

XPath operators, functions, wildcards, and node-addressing mechanisms can be combined in wide variety of ways. The introduction you've had so far should give you a good head start at specifying the pattern you need for any particular purpose.

Writing Out a DOM as an XML File

Once you have constructed a DOM, either by parsing an XML file or building it programmatically, you frequently want to save it as XML. This section shows you how to do that using the Xalan transform package.

Using that package, you'll create a transformer object to wire a `DomSource` to a `StreamResult`. You'll then invoke the transformer's `transform()` method to write out the DOM as XML data.

Reading the XML

The first step is to create a DOM in memory by parsing an XML file. By now, you should be getting pretty comfortable with the process.

Note: The code discussed in this section is in `TransformationApp01.java`.

The code below provides a basic template to start from. (It should be familiar. It's basically the same code you wrote at the start of the DOM tutorial. If you saved it then, that version should be pretty much the equivalent of what you see below.)

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

import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

import org.w3c.dom.Document;
import org.w3c.dom.DOMException;

import java.io.*;

public class TransformationApp
{
  static Document document;

  public static void main(String argv[])
  {
    if (argv.length != 1) {
      System.err.println (
        "Usage: java TransformationApp filename");
      System.exit (1);
    }

    DocumentBuilderFactory factory =
      DocumentBuilderFactory.newInstance();
    //factory.setNamespaceAware(true);
    //factory.setValidating(true);

    try {
      File f = new File(argv[0]);
      DocumentBuilder builder =
        factory.newDocumentBuilder();
      document = builder.parse(f);

    } catch (SAXParseException spe) {
      // Error generated by the parser
      System.out.println("\n** Parsing error"
             + ", line " + spe.getLineNumber()
             + ", uri " + spe.getSystemId());
          System.out.println("  " + spe.getMessage() );

          // Use the contained exception, if any
          Exception x = spe;
          if (spe.getException() != null)
             x = spe.getException();
          x.printStackTrace();

       } catch (SAXException sxe) {
          // Error generated by this application
          // (or a parser-initialization error)
          Exception x = sxe;
          if (sxe.getException() != null)
             x = sxe.getException();
          x.printStackTrace();

       } catch (ParserConfigurationException pce) {
          // Parser with specified options can't be built
          pce.printStackTrace();

       } catch (IOException ioe) {
          // I/O error
          ioe.printStackTrace();
       }
    } // main
 }
```

Creating a Transformer

The next step is to create a transformer you can use to transmit the XML to System.out.

Note: The code discussed in this section is in TransformationApp02.java. The file it runs on is slideSample01.xml. The output is in TransformationLog02.txt. (The browsable versions are slideSample01-xml.html and TransformationLog02.html.)

Start by adding the import statements highlighted below:

```
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerConfigurationException;

import javax.xml.transform.dom.DOMSource;

import javax.xml.transform.stream.StreamResult;

import java.io.*;
```

Here, you've added a series of classes which should now be forming a standard pattern: an entity (`Transformer`), the factory to create it (`TransformerFactory`), and the exceptions that can be generated by each. Since a transformation always has a *source* and a *result*, you then imported the classes necessary to use a DOM as a source (`DomSource`), and an output stream for the result (`StreamResult`).

Next, add the code to carry out the transformation:

```
try {
  File f = new File(argv[0]);
  DocumentBuilder builder = factory.newDocumentBuilder();
  document = builder.parse(f);

   // Use a Transformer for output
  TransformerFactory tFactory =
    TransformerFactory.newInstance();
  Transformer transformer = tFactory.newTransformer();

  DOMSource source = new DOMSource(document);
  StreamResult result = new StreamResult(System.out);
  transformer.transform(source, result);
```

Here, you created a transformer object, used the DOM to construct a source object, and used `System.out` to construct a result object. You then told the transformer to operate on the source object and output to the result object.

Note: In this case, the "transformer" isn't actually changing anything. In XSLT terminology, you are using the *identity transform*, which means that the "transformation" generates a copy of the source, unchanged.

Finally, add the code highlighted below to catch the new errors that can be generated:

```
} catch (TransformerConfigurationException tce) {
  // Error generated by the parser
  System.out.println ("* Transformer Factory error");
  System.out.println("  " + tce.getMessage() );

   // Use the contained exception, if any
  Throwable x = tce;
  if (tce.getException() != null)
    x = tce.getException();
  x.printStackTrace();

} catch (TransformerException te) {
  // Error generated by the parser
  System.out.println ("* Transformation error");
  System.out.println("  " + te.getMessage() );

  // Use the contained exception, if any
  Throwable x = te;
  if (te.getException() != null)
    x = te.getException();
  x.printStackTrace();

} catch (SAXParseException spe) {
  ...
```

Notes:

- `TransformerExceptions` are thrown by the transformer object.
- `TransformerConfigurationExceptions` are thrown by the factory.
- To preserve the XML document's `DOCTYPE` setting, it is also necessary to add the following code:

```
import javax.xml.transform.OutputKeys;
...
if (document.getDoctype() != null){
  String systemValue = (new
    File(document.getDoctype().getSystemId())).getName();
  transformer.setOutputProperty(
    OutputKeys.DOCTYPE_SYSTEM, systemValue
    );
}
```

Writing the XML

For instructions on how to compile and run the program, see Compiling and Running the Program (page 143) from the SAX tutorial. (If you're working along, substitute "TransformationApp" for "Echo" as the name of the program. If you are compiling the sample code, use "TransformationApp02".) When you run the program on `slideSample01.xml`, this is the output you see:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- A SAMPLE set of slides -->
<slideshow author="Yours Truly" date="Date of publication"
title="Sample Slide Show">

  <!-- TITLE SLIDE -->
  <slide type="all">
    <title>Wake up to WonderWidgets!</title>
  </slide>

  <!-- OVERVIEW -->
  <slide type="all">
    <title>Overview</title>
    <item>Why <em>WonderWidgets</em> are great</item>
    <item/>
    <item>Who <em>buys</em> WonderWidgets</item>
  </slide>

</slideshow>
```

Note: The order of the attributes may vary, depending on which parser you are using.

To find out more about configuring the factory and handling validation errors, see Reading XML Data into a DOM, Additional Information (page 223).

Writing Out a Subtree of the DOM

It is also possible to operate on a subtree of a DOM. In this section of the tutorial, you'll experiment with that option.

Note: The code discussed in this section is in `TransformationApp03.java`. The output is in `TransformationLog03.txt`. (The browsable version is `TransformationLog03.html`.)

The only difference in the process is that now you will create a `DOMSource` using a node in the DOM, rather than the entire DOM. The first step will be to import the classes you need to get the node you want. Add the code highlighted below to do that:

```
import org.w3c.dom.Document;
import org.w3c.dom.DOMException;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
```

The next step is to find a good node for the experiment. Add the code highlighted below to select the first `<slide>` element:

```
try {
  File f = new File(argv[0]);
  DocumentBuilder builder = factory.newDocumentBuilder();
  document = builder.parse(f);

  // Get the first <slide> element in the DOM
  NodeList list = document.getElementsByTagName("slide");
  Node node = list.item(0);
```

Finally, make the changes shown below to construct a source object that consists of the subtree rooted at that node:

~~DOMSource source = new DOMSource(document);~~

```
DOMSource source = new DOMSource(node);
StreamResult result = new StreamResult(System.out);
transformer.transform(source, result);
```

Now run the app. Your output should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<slide type="all">
    <title>Wake up to WonderWidgets!</title>
  </slide>
```

Clean Up

Because it will be easiest to do now, make the changes shown below to back out the additions you made in this section. (`TransformationApp04.java` contains these changes.)

```
Import org.w3c.dom.DOMException;
~~import org.w3c.dom.Node;~~
~~import org.w3c.dom.NodeList;~~
...
  try {
     ...
     ~~// Get the first <slide> element in the DOM~~
     ~~NodeList list = document.getElementsByTagName("slide");~~
     ~~Node node = list.item(0);~~
     ...
     ~~DOMSource source = new DOMSource(node);~~
     StreamResult result = new StreamResult(System.out);
     transformer.transform(source, result);
```

Summary

At this point, you've seen how to use a transformer to write out a DOM, and how to use a subtree of a DOM as the source object in a transformation. In the next section, you'll see how to use a transformer to create XML from any data structure you are capable of parsing.

Generating XML from an Arbitrary Data Structure

In this section, you'll use XSLT to convert an *arbitrary data structure* to XML.

In general outline, then:

1. You'll modify an existing program that reads the data, in order to make it generate SAX events. (Whether that program is a real parser or simply a data filter of some kind is irrelevant for the moment.)
2. You'll then use the SAX "parser" to construct a `SAXSource` for the transformation.

3. You'll use the same `StreamResult` object you created in the last exercise, so you can see the results. (But note that you could just as easily create a `DOMResult` object to create a DOM in memory.)
4. You'll wire the source to the result, using the transformer object to make the conversion.

For starters, you need a data set you want to convert and a program capable of reading the data. In the next two sections, you'll create a simple data file and a program that reads it.

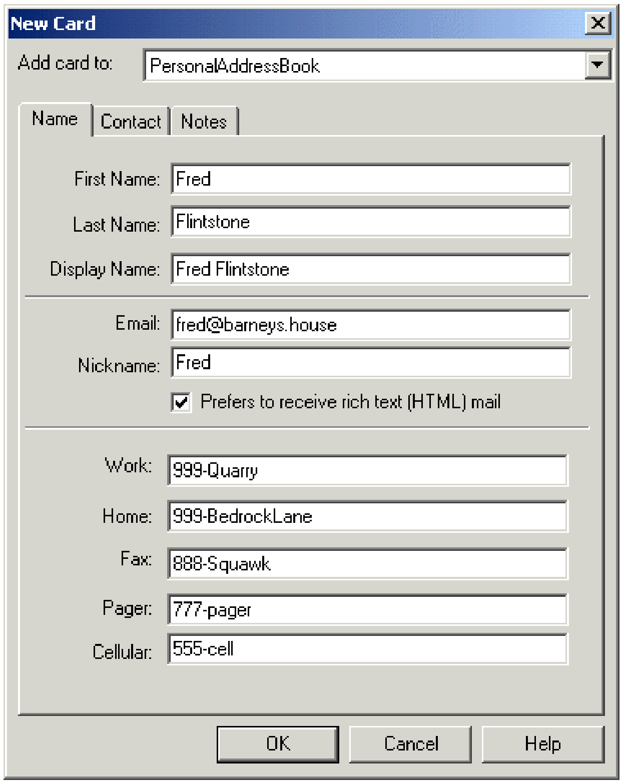
Creating a Simple File

We'll start by creating a data set for an address book. You can duplicate the process, if you like, or simply make use of the data stored in `PersonalAddressBook.ldif`.

The file shown below was produced by creating a new address book in Netscape Messenger, giving it some dummy data (one address card) and then exporting it in LDIF format.

Note: LDIF stands for LDAP Data Interchange Format. LDAP, turn, stands for Lightweight Directory Access Protocol. I prefer to think of LDIF as the "Line Delimited Interchange Format", since that is pretty much what it is.

Figure 8–1 shows the address book entry that was created.



Add card to: PersonalAddressBook

First Name: Fred
Last Name: Flintstone
Display Name: Fred Flintstone
Email: fred@barneys.house
Nickname: Fred
[✓] Prefers to receive rich text (HTML) mail
Work: 999-Quarry
Home: 999-BedrockLane
Fax: 888-Squawk
Pager: 777-pager
Cellular: 555-cell

Figure 8–1 Address Book Entry

Exporting the address book produces a file like the one shown below. The parts of the file that we care about are shown in bold.

```
dn: cn=Fred Flintstone,mail=fred@barneys.house
modifytimestamp: 20010409210816Z
cn: Fred Flintstone
xmozillanickname: Fred
mail: Fred@barneys.house
xmozillausehtmlmail: TRUE
givenname: Fred
sn: Flintstone
telephonenumber: 999-Quarry
homephone: 999-BedrockLane
facsimiletelephonenumber: 888-Squawk
pagerphone: 777-pager
```

```
cellphone: 555-cell
xmozillaanyphone: 999-Quarry
objectclass: top
objectclass: person
```

Note that each line of the file contains a variable name, a colon, and a space followed by a value for the variable. The `sn` variable contains the person's surname (last name) and the variable `cn` contains the DisplayName field from the address book entry.

Creating a Simple Parser

The next step is to create a program that parses the data.

Note: The code discussed in this section is in `AddressBookReader01.java`. The output is in `AddressBookReaderLog01.txt`.

The text for the program is shown below. It's an absurdly simple program that doesn't even loop for multiple entries because, after all, it's just a demo!

```
import java.io.*;

public class AddressBookReader
{

  public static void main(String argv[])
  {
    // Check the arguments
    if (argv.length != 1) {
      System.err.println (
        "Usage: java AddressBookReader filename");
      System.exit (1);
    }
    String filename = argv[0];
    File f = new File(filename);
    AddressBookReader01 reader = new AddressBookReader01();
    reader.parse(f);
  }

  /** Parse the input */
  public void parse(File f)
  {
    try {
            // Get an efficient reader for the file
            FileReader r = new FileReader(f);
            BufferedReader br = new BufferedReader(r);

             // Read the file and display it's contents.
            String line = br.readLine();
            while (null != (line = br.readLine())) {
               if (line.startsWith("xmozillanickname: "))
                  break;
            }
            output("nickname", "xmozillanickname", line);
            line = br.readLine();
            output("email",    "mail",             line);
            line = br.readLine();
            output("html",     "xmozillausehtmlmail", line);
            line = br.readLine();
            output("firstname","givenname",        line);
            line = br.readLine();
            output("lastname", "sn",               line);
            line = br.readLine();
            output("work",     "telephonenumber",  line);
            line = br.readLine();
            output("home",     "homephone",        line);
            line = br.readLine();
            output("fax",      "facsimiletelephonenumber",
               line);
            line = br.readLine();
            output("pager",    "pagerphone",       line);
            line = br.readLine();
            output("cell",     "cellphone",        line);

         }
         catch (Exception e) {
            e.printStackTrace();
         }
      }

      void output(String name, String prefix, String line)
      {
         int startIndex = prefix.length() + 2;
            // 2=length of ": "
         String text = line.substring(startIndex);
         System.out.println(name + ": " + text);
      }
   }
```

This program contains three methods:

main

The `main` method gets the name of the file from the command line, creates an instance of the parser, and sets it to work parsing the file. This method will be going away when we convert the program into a SAX parser. (That's one reason for putting the parsing code into a separate method.)

parse

This method operates on the `File` object sent to it by the main routine. As you can see, it's about as simple as it can get. The only nod to efficiency is the use of a `BufferedReader`, which can become important when you start operating on large files.

output

The `output` method contains the logic for the structure of a line. Starting from the right It takes three arguments. The first argument gives the method a name to display, so we can output "html" as a variable name, instead of "`xmozillausehtmlmail`". The second argument gives the variable name stored in the file (`xmozillausehtmlmail`). The third argument gives the line containing the data. The routine then strips off the variable name from the start of the line and outputs the desired name, plus the data.

Running this program on `PersonalAddressBook.ldif` produces this output:

```
nickname: Fred
email: Fred@barneys.house
html: TRUE
firstname: Fred
lastname: Flintstone
work: 999-Quarry
home: 999-BedrockLane
fax: 888-Squawk
pager: 777-pager
cell: 555-cell
```

I think we can all agree that's a bit more readable.

Modifying the Parser to Generate SAX Events

The next step is to modify the parser to generate SAX events, so you can use it as the basis for a `SAXSource` object in an XSLT `transform`.

Note: The code discussed in this section is in `AddressBookReader02.java`.

Start by importing the additional classes you're going to need:

```
import java.io.*;

import org.xml.sax.*;
import org.xml.sax.helpers.AttributesImpl;
```

Next, modify the application so that it extends `XmlReader`. That change converts the application into a parser that generates the appropriate SAX events.

```
public class AddressBookReader
  implements XMLReader
{
```

Now, remove the `main` method. You won't be needing that any more.

```
~~public static void main(String argv[])~~
~~{~~
  ~~// Check the arguments~~
  ~~if (argv.length != 1) {~~
     ~~System.err.println ("Usage: Java AddressBookReader ~~
~~filename");~~
     ~~System.exit (1);~~
  ~~}~~
  ~~String filename = argv[0];~~
  ~~File f = new File(filename);~~
  ~~AddressBookReader02 reader = new AddressBookReader02();~~
  ~~reader.parse(f);~~
}
```

Add some global variables that will come in handy in a few minutes:

```
public class AddressBookReader
  implements XMLReader
{
  ContentHandler handler;

  // We're not doing namespaces, and we have no
  // attributes on our elements.
  String nsu = "";  // NamespaceURI
```

```
    Attributes atts = new AttributesImpl();
    String rootElement = "addressbook";

    String indent = "\n      "; // for readability!
```

The SAX `ContentHandler` is the object that is going to get the SAX events the parser generates. To make the application into an `XmlReader`, you'll be defining a `setContentHandler` method. The `handler` variable will hold a reference to the object that is sent when `setContentHandler` is invoked.

And, when the parser generates SAX *element* events, it will need to supply namespace and attribute information. Since this is a simple application, you're defining null values for both of those.

You're also defining a root element for the data structure (`addressbook`), and setting up an indent string to improve the readability of the output.

Next, modify the `parse` method so that it takes an `InputSource` as an argument, rather than a `File`, and account for the exceptions it can generate:

```
  public void parse(~~File f)~~InputSource input)
  throws IOException, SAXException
```

Now make the changes shown below to get the reader encapsulated by the `InputSource` object:

```
  try {
    // Get an efficient reader for the file
    ~~FileReader r = new FileReader(f);~~
    java.io.Reader r = input.getCharacterStream();
    BufferedReader Br = new BufferedReader(r);
```

Note: In the next section, you'll create the input source object and what you put in it will, in fact, be a buffered reader. But the `AddressBookReader` could be used by someone else, somewhere down the line. This step makes sure that the processing will be efficient, regardless of the reader you are given.

The next step is to modify the `parse` method to generate SAX events for the start of the document and the root element. Add the code highlighted below to do that:

```
/** Parse the input */
public void parse(InputSource input)
...
{
  try {
     ...
     // Read the file and display its contents.
     String line = br.readLine();
     while (null != (line = br.readLine())) {
        if (line.startsWith("xmozillanickname: ")) break;
     }

     if (handler==null) {
        throw new SAXException("No content handler");
     }

     handler.startDocument();
     handler.startElement(nsu, rootElement,
        rootElement, atts);

     output("nickname", "xmozillanickname", line);
     ...
     output("cell",     "cellphone",        line);

     handler.ignorableWhitespace("\n".toCharArray(),
                0, // start index
                1  // length
                );
     handler.endElement(nsu, rootElement, rootElement);
     handler.endDocument();
  }
  catch (Exception e) {
  ...
```

Here, you first checked to make sure that the parser was properly configured with a `ContentHandler`. (For this app, we don't care about anything else.) You then generated the events for the start of the document and the root element, and finished by sending the end-event for the root element and the end-event for the document.

A couple of items are noteworthy, at this point:

- We haven't bothered to send the `setDocumentLocator` event, since that is optional. Were it important, that event would be sent immediately before the `startDocument` event.
- We've generated an `ignorableWhitespace` event before the end of the root element. This, too, is optional, but it drastically improves the readability of the output, as you'll see in a few moments. (In this case, the whitespace consists of a single newline, which is sent the same way that characters are sent to the `characters` method: as a character array, a starting index, and a length.)

Now that SAX events are being generated for the document and the root element, the next step is to modify the `output` method to generate the appropriate element events for each data item. Make the changes shown below to do that:

```
void output(String name, String prefix, String line)
throws SAXException
{
  int startIndex = prefix.length() + 2; // 2=length of ": "
  String text = line.substring(startIndex);
  ~~System.out.println(name + ": " + text);~~

  int textLength = line.length() - startIndex;
  handler.ignorableWhitespace(indent.toCharArray(),
                 0, // start index
                 indent.length()
                 );
  handler.startElement(nsu, name, name /*"qName"*/, atts);
  handler.characters(line.toCharArray(),
            startIndex,
            textLength);
  handler.endElement(nsu, name, name);
}
```

Since the `ContentHandler` methods can send `SAXExceptions` back to the parser, the parser has to be prepared to deal with them. In this case, we don't expect any, so we'll simply allow the application to fail if any occur.

You then calculate the length of the data, and once again generate some ignorable whitespace for readability. In this case, there is only one level of data, so we can use a fixed-indent string. (If the data were more structured, we would have to calculate how much space to indent, depending on the nesting of the data.)

Note: The indent string makes no difference to the data, but will make the output a lot easier to read. Once everything is working, try generating the result without that string! All of the elements will wind up concatenated end to end, like this: `<addressbook><nickname>Fred</nickname><email>...`

Next, add the method that configures the parser with the `ContentHandler` that is to receive the events it generates:

```
void output(String name, String prefix, String line)
  throws SAXException
{
  ...
}

/** Allow an application to register a content event handler. */
public void setContentHandler(ContentHandler handler) {
  this.handler = handler;
}

/** Return the current content handler. */
public ContentHandler getContentHandler() {
  return this.handler;
}
```

There are several more methods that must be implemented in order to satisfy the `XmlReader` interface. For the purpose of this exercise, we'll generate null methods for all of them. For a production application, though, you may want to consider implementing the error handler methods to produce a more robust app. For now, though, add the code highlighted below to generate null methods for them:

```
/** Allow an application to register an error event handler. */
public void setErrorHandler(ErrorHandler handler)
{ }

/** Return the current error handler. */
public ErrorHandler getErrorHandler()
{ return null; }
```

Finally, add the code highlighted below to generate null methods for the remainder of the `XmlReader` interface. (Most of them are of value to a real SAX parser, but have little bearing on a data-conversion application like this one.)

```
/** Parse an XML document from a system identifier (URI). */
public void parse(String systemId)
throws IOException, SAXException
{ }

 /** Return the current DTD handler. */
public DTDHandler getDTDHandler()
{ return null; }

/** Return the current entity resolver. */
public EntityResolver getEntityResolver()
{ return null; }

/** Allow an application to register an entity resolver. */
public void setEntityResolver(EntityResolver resolver)
{ }

/** Allow an application to register a DTD event handler. */
public void setDTDHandler(DTDHandler handler)
{ }

/** Look up the value of a property. */
public Object getProperty(String name)
{ return null; }

/** Set the value of a property. */
public void setProperty(String name, Object value)
{ }

/** Set the state of a feature. */
public void setFeature(String name, boolean value)
{ }

/** Look up the value of a feature. */
public boolean getFeature(String name)
{ return false; }
```

Congratulations! You now have a parser you can use to generate SAX events. In the next section, you'll use it to construct a SAX source object that will let you transform the data into XML.

Using the Parser as a SAXSource

Given a SAX parser to use as an event source, you can (easily!) construct a transformer to produce a result. In this section, you'll modify the `TransformerApp` you've been working with to produce a stream output result, although you could just as easily produce a DOM result.

Note: The code discussed in this section is in `TransformationApp04.java`. The results of running it are in `TransformationLog04.txt`.

Important!

Make sure you put the `AddressBookReader` aside and open up the `TransformationApp`. The work you do in this section affects the `TransformationApp`! (The look pretty similar, so it's easy to start working on the wrong one.)

Start by making the changes shown below to import the classes you'll need to construct a `SAXSource` object. (You won't be needing the DOM classes at this point, so they are discarded here, although leaving them in doesn't do any harm.)

```
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.ContentHandler;
import org.xml.sax.InputSource;
~~import org.w3c.dom.Document;~~
~~import org.w3c.dom.DOMException;~~
...
~~import javax.xml.transform.dom.DOMSource;~~
import javax.xml.transform.sax.SAXSource;
import javax.xml.transform.stream.StreamResult;
```

Next, remove a few other holdovers from our DOM-processing days, and add the code to create an instance of the `AddressBookReader`:

```
public class TransformationApp
{
  ~~// Global value so it can be ref'd by the tree-adapter~~
  ~~static Document document;~~

   public static void main(String argv[])
  {
    ...
    ~~DocumentBuilderFactory factory =~~
      ~~DocumentBuilderFactory.newInstance();~~
```

```
        ~~//factory.setNamespaceAware(true);~~
        ~~//factory.setValidating(true);~~

        // Create the sax "parser".
        AddressBookReader saxReader = new AddressBookReader();

        try {
          File f = new File(argv[0]);
          ~~DocumentBuilder builder =~~
            ~~factory.newDocumentBuilder();~~
          ~~document = builder.parse(f);~~
```

Guess what! You're almost done. Just a couple of steps to go. Add the code highlighted below to construct a SAXSource object:

```
    // Use a Transformer for output
    ...
    Transformer transformer = tFactory.newTransformer();

    // Use the parser as a SAX source for input
    FileReader fr = new FileReader(f);
    BufferedReader br = new BufferedReader(fr);
    InputSource inputSource = new InputSource(br);
    SAXSource source = new SAXSource(saxReader, inputSource);

    StreamResult result = new StreamResult(System.out);
    transformer.transform(source, result);
```

Here, you constructed a buffered reader (as mentioned earlier) and encapsulated it in an input source object. You then created a `SAXSource` object, passing it the reader and the `InputSource` object, and passed that to the transformer.

When the application runs, the transformer will configure itself as the `ContentHandler` for the SAX parser (the `AddressBookReader`) and tell the parser to operate on the `inputSource` object. Events generated by the parser will then go to the transformer, which will do the appropriate thing and pass the data on to the result object.

Finally, remove the exceptions you no longer need to worry about, since the `TransformationApp` no longer generates them:

```
    ~~catch (SAXParseException spe) {~~
      ~~// Error generated by the parser~~
      ~~System.out.println("\n** Parsing error"~~
        ~~+ ", line " + spe.getLineNumber()~~
        ~~+ ", uri " + spe.getSystemId());~~
      ~~System.out.println("   " + spe.getMessage() );~~
    ~~// Use the contained exception, if any~~
    ~~Exception  x = spe;~~
    ~~if (spe.getException() != null)~~
      ~~x = spe.getException();~~
    ~~x.printStackTrace();~~

  ~~} catch (SAXException sxe) {~~
    ~~// Error generated by this application~~
    ~~// (or a parser-initialization error)~~
    ~~Exception  x = sxe;~~
    ~~if (sxe.getException() != null)~~
      ~~x = sxe.getException();~~
    ~~x.printStackTrace();~~

  ~~} catch (ParserConfigurationException pce) {~~
    ~~// Parser with specified options can't be built~~
    ~~pce.printStackTrace();~~

  } catch (IOException ioe) {
    ...
```

You're done! You have now created a transformer which will use a `SAXSource` as input, and produce a `StreamResult` as output.

Doing the Conversion

Now run the application on the address book file. Your output should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<addressbook>
  <nickname>Fred</nickname>
  <email>fred@barneys.house</email>
  <html>TRUE</html>
  <firstname>Fred</firstname>
  <lastname>Flintstone</lastname>
  <work>999-Quarry</work>
  <home>999-BedrockLane</home>
  <fax>888-Squawk</fax>
  <pager>777-pager</pager>
  <cell>555-cell</cell>
</addressbook>
```

You have now successfully converted an existing data structure to XML. And it wasn't even that hard. Congratulations!

Transforming XML Data with XSLT

The XML Stylesheet Language for Transformations (XSLT) can be used for many purposes. For example, with a sufficiently intelligent stylesheet, you could generate PDF or PostScript output from the XML data. But generally, XSLT is used to generate formatted HTML output, or to create an alternative XML representation of the data.

In this section of the tutorial, you'll use an XSLT transform to translate XML input data to HTML output.

Note: The XSLT specification is large and complex. So this tutorial can only scratch the surface. It will give you enough of a background to get started, so you can undertake simple XSLT processing tasks. It should also give you a head start when you investigate XSLT further. For a more thorough grounding, consult a good reference manual, such as Michael Kay's XSLT Programmer's Reference.

Defining a Simple <article> Document Type

We'll start by defining a very simple document type that could be used for writing articles. Our `<article>` documents will contain these structure tags:

- `<TITLE>` — The title of the article
- `<SECT>` — A section, consisting of a *heading* and a *body*
- `<PARA>` — A paragraph
- `<LIST>` — A list.
- `<ITEM>` — An entry in a list
- `<NOTE>` — An aside, which will be offset from the main text

The slightly unusual aspect of this structure is that we won't create a separate element tag for a section heading. Such elements are commonly created to distinguish the heading text (and any tags it contains) from the body of the section (that is, any structure elements underneath the heading).

Instead, we'll allow the heading to merge seamlessly into the body of a section. That arrangement adds some complexity to the stylesheet, but that will give us a chance to explore XSLT's template-selection mechanisms. It also matches our intuitive expectations about document structure, where the text of a heading is directly followed by structure elements, which can simplify outline-oriented editing.

Note: However, that structure is not easily validated, because XML's mixed-content model allows text anywhere in a section, whereas we want to confine text and inline elements so that they only appear before the first structure element in the body of the section. The assertion-based validator (Schematron (page 58)) can do it, but most other schema mechanisms can't. So we'll dispense with defining a DTD for the document type.

In this structure, sections can be nested. The depth of the nesting will determine what kind of HTML formatting to use for the section heading (for example, `h1` or `h2`). Using a plain `SECT` tag (instead of numbered sections) is also useful with outline-oriented editing, because it lets you move sections around at will without having to worry about changing the numbering for that section or for any of the other sections that might be affected by the move.

For lists, we'll use a `type` attribute to specify whether the list entries are `unordered` (bulleted), `alpha` (enumerated with lower case letters), `ALPHA` (enumerated with uppercase letters), or `numbered`.

We'll also allow for some inline tags that change the appearance of the text:

- `<B>` — bold
- `<I>` — italics
- `<U>` — underline
- `<DEF>` — definition
- `<LINK>` — link to a URL

Note: An *inline* tag does not generate a line break, so a style change caused by an inline tag does not affect the flow of text on the page (although it will affect the appearance of that text). A *structure* tag, on the other hand, demarcates a new segment of text, so at a minimum it always generates a line break, in addition to other format changes.

The `<DEF>` tag will be used for terms that are defined in the text. Such terms will be displayed in italics, the way they ordinarily are in a document. But using a special tag in the XML will allow an index program to find such definitions and add them to an index, along with keywords in headings. In the *Note* above, for example, the definitions of inline tags and structure tags could have been marked with `<DEF>` tags, for future indexing.

Finally, the `LINK` tag serves two purposes. First, it will let us create a link to a URL without having to put the URL in twice — so we can code `<link>http//...</link>` instead of `<a href="http//...">http//...</a>`. Of course, we'll also want to allow a form that looks like `<link target="...">...name...</link>`. That leads to the second reason for the `<link>` tag—it will give us an opportunity to play with conditional expressions in XSLT.

Note: Although the article structure is exceedingly simple (consisting of only 11 tags), it raises enough interesting problems to get a good view of XSLT's basic capabilities. But we'll still leave large areas of the specification untouched. The last part of this tutorial will point out the major features we skipped.

Creating a Test Document

Here, you'll create a simple test document using nested `<SECT>` elements, a few `<PARA>` elements, a `<NOTE>` element, a `<LINK>`, and a `<LIST type="unordered">`. The idea is to create a document with one of everything, so we can explore the more interesting translation mechanisms.

Note: The sample data described here is contained in `article1.xml`. (The browsable version is `article1-xml.html`.)

To make the test document, create a file called `article.xml` and enter the XML data shown below.

```
<?xml version="1.0"?>
<ARTICLE>
  <TITLE>A Sample Article</TITLE>
  <SECT>The First Major Section
    <PARA>This section will introduce a subsection.</PARA>
    <SECT>The Subsection Heading
      <PARA>This is the text of the subsection.
      </PARA>
    </SECT>
  </SECT>
</ARTICLE>
```

Note that in the XML file, the subsection is totally contained within the major section. (In HTML, on the other hand, headings do not *contain* the body of a section.) The result is an outline structure that is harder to edit in plain-text form, like this, but is much easier to edit with an outline-oriented editor.

Someday, given an tree-oriented XML editor that understands inline tags like `<B>` and `<I>`, it should be possible to edit an article of this kind in outline form, without requiring a complicated stylesheet. (Such an editor would allow the writer to focus on the structure of the article, leaving layout until much later in the process.) In such an editor, the article-fragment above would look something like this:

```
<ARTICLE>
  <TITLE>A Sample Article
  <SECT>The First Major Section
    <PARA>This section will introduce a subsection.
    <SECT>The Subheading
      <PARA>This is the text of the subsection. Note that ...
```

Note: At the moment, tree-structured editors exist, but they treat inline tags like `<B>` and `<I>` the same way that they treat other structure tags, which can make the "outline" a bit difficult to read.

Writing an XSLT Transform

In this part of the tutorial, you'll begin writing an XSLT transform that will convert the XML article and render it in HTML.

Note: The transform described in this section is contained in `article1a.xsl`. (The browsable version is `article1a-xsl.html`.)

Start by creating a normal XML document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

Then add the lines highlighted below to create an XSL stylesheet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0"
  >

</xsl:stylesheet>
```

Now, set it up to produce HTML-compatible output:

```
<xsl:stylesheet
  ...
  >
  <xsl:output method="html"/>

   ...

</xsl:stylesheet>
```

We'll get into the detailed reasons for that entry later on in this section. But for now, note that if you want to output anything besides well-formed XML, then you'll need an `<xsl:output>` tag like the one shown, specifying either "`text`" or "`html`". (The default value is "`xml`".)

Note: When you specify XML output, you can add the `indent` attribute to produce nicely indented XML output. The specification looks like this: `<xsl:output method="xml" indent="yes"/>`.

Processing the Basic Structure Elements

You'll start filling in the stylesheet by processing the elements that go into creating a table of contents — the root element, the title element, and headings. You'll also process the `PARA` element defined in the test document.

Note: If on first reading you skipped the section of this tutorial that discusses the XPAth addressing mechanisms, How XPath Works (page 294), now is a good time to go back and review that section.

Begin by adding the main instruction that processes the root element:

```
   <xsl:template match="/">
     <html><body>
       <xsl:apply-templates/>
     </body></html>
   </xsl:template>

 </xsl:stylesheet>
```

The new XSL commands are shown in bold. (Note that they are defined in the "`xsl`" namespace.) The instruction `<xsl:apply-templates>` processes the children of the current node. In this case, the current node is the root node.

Despite its simplicity, this example illustrates a number of important ideas, so it's worth understanding thoroughly. The first concept is that a stylesheet contains a number of *templates*, defined with the `<xsl:template>` tag. Each template contains a `match` attribute, which selects the elements that the template will be applied to, using the XPath addressing mechanisms described in How XPath Works (page 294).

Within the template, tags that do not start with the `xsl:` namespace prefix are simply copied. The newlines and whitespace that follow them are also copied, which helps to make the resulting output readable.

Note: When a newline is not present, whitespace is generally ignored. To include whitespace in the output in such cases, or to include other text, you can use the `<xsl:text>` tag. Basically, an XSLT stylesheet expects to process tags. So everything it sees needs to be either an `<xsl:..>` tag, some other tag, or whitespace.

In this case, the non-XSL tags are HTML tags. So when the root tag is matched, XSLT outputs the HTML start-tags, processes any templates that apply to children of the root, and then outputs the HTML end-tags.

Process the <TITLE> Element

Next, add a template to process the article title:

```
   <xsl:template match="/ARTICLE/TITLE">
     <h1 align="center"> <xsl:apply-templates/> </h1>
   </xsl:template>

 </xsl:stylesheet>
```

In this case, you specified a complete path to the `TITLE` element, and output some HTML to make the text of the title into a large, centered heading. In this case, the `apply-templates` tag ensures that if the title contains any inline tags like italics, links, or underlining, they will be processed as well.

More importantly, the `apply-templates` instruction causes the *text* of the title to be processed. Like the DOM data model, the XSLT data model is based on the concept of *text nodes* contained in *element nodes* (which, in turn, can be contained in other element nodes, and so on). That hierarchical structure constitutes the source tree. There is also a result tree, which contains the output.

XSLT works by transforming the source tree into the result tree. To visualize the result of XSLT operations, it is helpful to understand the structure of those trees, and their contents. (For more on this subject, see The XSLT/XPath Data Model (page 295).)

Process Headings

To continue processing the basic structure elements, add a template to process the top-level headings:

```
    <xsl:template match="/ARTICLE/SECT">
      <h2> <xsl:apply-templates
        select="text()|B|I|U|DEF|LINK"/> </h2>
      <xsl:apply-templates select="SECT|PARA|LIST|NOTE"/>
    </xsl:template>

  </xsl:stylesheet>
```

Here, you've specified the path to the topmost `SECT` elements. But this time, you've applied templates in two stages, using the `select` attribute. For the first stage, you selected text nodes using the XPath `text()` function, as well as inline tags like bold and italics. (The vertical pipe (|) is used to match multiple items — text, *or* a bold tag, *or* an italics tag, etc.) In the second stage, you selected the other structure elements contained in the file, for sections, paragraphs, lists, and notes.

Using the `select` attribute let you put the text and inline elements between the `<h2>...</h2>` tags, while making sure that all of the structure tags in the section are processed afterwards. In other words, you made sure that the nesting of the headings in the XML document is *not* reflected in the HTML formatting, which is important for HTML output.

In general, using the `select` clause lets you apply all templates to a subset of the information available in the current context. As another example, this template selects all attributes of the current node:

```
<xsl:apply-templates select="@*"/></attributes>
```

Next, add the virtually identical template to process subheadings that are nested one level deeper:

```
   <xsl:template match="/ARTICLE/SECT/SECT">
      <h3> <xsl:apply-templates
         select="text()|B|I|U|DEF|LINK"/> </h3>
      <xsl:apply-templates select="SECT|PARA|LIST|NOTE"/>
   </xsl:template>

</xsl:stylesheet>
```

Generate a Runtime Message

You could add templates for deeper headings, too, but at some point you have to stop, if only because HTML only goes down to five levels. But for this example, you'll stop at two levels of section headings. But if the XML input happens to contain a third level, you'll want to deliver an error message to the user. This section shows you how to do that.

Note: We *could* continue processing SECT elements that are further down, by selecting them with the expression `/SECT/SECT//SECT`. The `//` selects any SECT elements, at any depth, as defined by the XPath addressing mechanism. But we'll take the opportunity to play with messaging, instead.

Add the following template to generate an error when a section is encountered that is nested too deep:

```
   <xsl:template match="/ARTICLE/SECT/SECT/SECT">
      <xsl:message terminate="yes">
         Error: Sections can only be nested 2 deep.
      </xsl:message>
   </xsl:template>

</xsl:stylesheet>
```

The `terminate="yes"` clause causes the transformation process to stop after the message is generated. Without it, processing could still go on with everything in that section being ignored.

As an additional exercise, you could expand the stylesheet to handle sections nested up to four sections deep, generating `<h2>...<h5>` tags. Generate an error on any section nested five levels deep.

Finally, finish up the stylesheet by adding a template to process the `PARA` tag:

```
  <xsl:template match="PARA">
    <p><xsl:apply-templates/></p>
  </xsl:template>

</xsl:stylesheet>
```

Writing the Basic Program

In this part of the tutorial, you'll modify the program that used XSLT to echo an XML file unchanged, changing it so it uses your stylesheet.

Note: The code shown in this section is contained in `Stylizer.java`. The result is `stylizer1a.html`. (The browser-displayable version of the HTML source is `stylizer1a-src.html`.)

Start by copying `TransformationApp02`, which parses an XML file and writes to `System.out`. Save it as `Stylizer.java`.

Next, modify occurrences of the class name and the usage section of the program:

```
public class ~~TransformationApp~~Stylizer
{
  if (argv.length != ~~1~~ 2) {
    System.err.println (
      ~~"Usage: java TransformationApp filename");~~
      "Usage: java Stylizer stylesheet xmlfile");
    System.exit (1);
  }
  ...
```

Then modify the program to use the stylesheet when creating the `Transformer` object.

```
...
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;
...

public class Stylizer
{
  ...
  public static void main (String argv[])
  {
    ...
    try {
      ~~File f = new File(argv[0]);~~
      File stylesheet = new File(argv[0]);
      File datafile  = new File(argv[1]);

      DocumentBuilder builder =
        factory.newDocumentBuilder();
      document = builder.parse(~~f~~ datafile);
      ...
      StreamSource stylesource =
        new StreamSource(stylesheet);
      Transformer transformer =
        Factory.newTransformer(stylesource);
      ...
```

This code uses the file to create a `StreamSource` object, and then passes the source object to the factory class to get the transformer.

Note: You can simplify the code somewhat by eliminating the `DOMSource` class entirely. Instead of creating a `DOMSource` object for the XML file, create a `Stream-Source` object for it, as well as for the stylesheet.

Now compile and run the program using `article1a.xsl` on `article1.xml`. The results should look like this:

```
<html>
<body>

<h1 align="center">A Sample Article</h1>
```

```
<h2>The First Major Section

    </h2>

<p>This section will introduce a subsection.</p>
<h3>The Subsection Heading

         </h3>

<p>This is the text of the subsection.

            </p>

</body>
</html>
```

At this point, there is quite a bit of excess whitespace in the output. You'll see how to eliminate most of it in the next section.

Trimming the Whitespace

If you recall, when you took a look at the structure of a DOM, there were many text nodes that contained nothing but ignorable whitespace. Most of the excess whitespace in the output came from these nodes. Fortunately, XSL gives you a way to eliminate them. (For more about the node structure, see The XSLT/XPath Data Model (page 295).)

Note: The stylesheet described here is `article1b.xsl`. The result is `stylizer1b.html`. (The browser-displayable versions are `article1b-xsl.html` and `stylizer1b-src.html`.)

To remove some of the excess whitespace, add the line highlighted below to the stylesheet.

```
<xsl:stylesheet ...
  >
  <xsl:output method="html"/>
  <xsl:strip-space elements="SECT"/>
  ...
```

This instruction tells XSL to remove any text nodes under SECT elements that contain nothing but whitespace. Nodes that contain text other than whitespace will not be affected, and other kinds of nodes are not affected.

Now, when you run the program, the result looks like this:

```
<html>
<body>

<h1 align="center">A Sample Article</h1>

<h2>The First Major Section
     </h2>
<p>This section will introduce a subsection.</p>
<h3>The Subsection Heading
        </h3>
<p>This is the text of the subsection.
        </p>

</body>
</html>
```

That's quite an improvement. There are still newline characters and white space after the headings, but those come from the way the XML is written:

```
<SECT>The First Major Section
____<PARA>This section will introduce a subsection.</PARA>
^^^^
```

Here, you can see that the section heading ends with a newline and indentation space, before the PARA entry starts. That's not a big worry, because the browsers that will process the HTML routinely compress and ignore the excess space. But there is still one more formatting tool at our disposal.

Note: The stylesheet described here is article1c.xsl. The result is stylizer1c.html. (The browser-displayable versions are article1c-xsl.html and stylizer1c-src.html.)

To get rid of that last little bit of whitespace, add this template to the stylesheet:

```
  <xsl:template match="text()">
    <xsl:value-of select="normalize-space()"/>
  </xsl:template>

</xsl:stylesheet>
```

The output now looks like this:

```
<html>
<body>
<h1 align="center">A Sample Article</h1>
<h2>The First Major Section</h2>
<p>This section will introduce a subsection.</p>
<h3>The Subsection Heading</h3>
<p>This is the text of the subsection.</p>
</body>
</html>
```

That is quite a bit better. Of course, it would be nicer if it were indented, but that turns out to be somewhat harder than expected! Here are some possible avenues of attack, along with the difficulties:

Indent option

Unfortunately, the `indent="yes"` option that can be applied to XML output is not available for HTML output. Even if that option were available, it wouldn't help, because HTML elements are rarely nested! Although HTML source is frequently indented to show the *implied* structure, the HTML tags themselves are not nested in a way that creates a *real* structure.

Indent variables

The `<xsl:text>` function lets you add any text you want, including whitespace. So, it could conceivably be used to output indentation space. The problem is to vary the *amount* of indentation space. XSLT variables seem like a good idea, but they don't work here. The reason is that when you assign a value to a variable in a template, the value is only known *within* that template (statically, at compile time value). Even if the variable is defined globally, the assigned value is not stored in a way that lets it be dynamically known by other templates at runtime. Once `<apply-templates/>` invokes other templates, they are unaware of any variable settings made in other templates.

Parameterized templates

Using a “parameterized template” is another way to modify a template’s behavior. But determining the amount of indentation space to pass as the parameter remains the crux of the problem!

At the moment, then, there does not appear to be any good way to control the indentation of HTML-formatted output. That would be inconvenient if you needed to display or edit the HTML as plain text. But it’s not a problem if you do your editing on the XML form, only use the HTML version for display in a browser. (When you view `stylizer1c.html`, for example, you see the results you expect.)

Processing the Remaining Structure Elements

In this section, you’ll process the `LIST` and `NOTE` elements that add additional structure to an article.

Note: The sample document described in this section is `article2.xml`, and the stylesheet used to manipulate it is `article2.xsl`. The result is `stylizer2.html`. (The browser-displayable versions are `article2-xml.html`, `article2-xsl.html`, and `stylizer2-src.html`.)

Start by adding some test data to the sample document:

```
<?xml version="1.0"?>
<ARTICLE>
  <TITLE>A Sample Article</TITLE>
  <SECT>The First Major Section
    ...
  </SECT>
  <SECT>The Second Major Section
    <PARA>This section adds a LIST and a NOTE.
    <PARA>Here is the LIST:
      <LIST type="ordered">
        <ITEM>Pears</ITEM>
        <ITEM>Grapes</ITEM>
      </LIST>
    </PARA>
    <PARA>And here is the NOTE:
      <NOTE>Don't forget to go to the hardware store
        on your way to the grocery!
        </NOTE>
      </PARA>
    </SECT>
  </ARTICLE>
```

Note: Although the list and note in the XML file are contained in their respective paragraphs, it really makes no difference whether they are contained or not—the generated HTML will be the same, either way. But having them contained will make them easier to deal with in an outline-oriented editor.

Modify <PARA> handling

Next, modify the PARA template to account for the fact that we are now allowing some of the structure elements to be embedded with a paragraph:

```
<xsl:template match="PARA">
  ~~<p><xsl:apply-templates/></p>~~
  <p> <xsl:apply-templates select="text()|B|I|U|DEF|LINK"/>
    </p>
  <xsl:apply-templates select="PARA|LIST|NOTE"/>
</xsl:template>
```

This modification uses the same technique you used for section headings. The only difference is that SECT elements are not expected within a paragraph. (However, a paragraph could easily exist inside another paragraph, as quoted material, for example.)

Process <LIST> and <ITEM> elements

Now you're ready to add a template to process LIST elements:

```
  <xsl:template match="LIST">
    <xsl:if test="@type='ordered'">
      <ol>
      <xsl:apply-templates/>
      </ol>
    </xsl:if>
    <xsl:if test="@type='unordered'">
      <ul>
      <xsl:apply-templates/>
      </ul>
    </xsl:if>
  </xsl:template>

</xsl:stylesheet>
```

The `<xsl:if>` tag uses the `test=""` attribute to specify a boolean condition. In this case, the value of the `type` attribute is tested, and the list that is generated changes depending on whether the value is `ordered` or `unordered`.

The two important things to note for this example are:

- There is no `else` clause, nor is there a `return` or `exit` statement, so it takes two `<xsl:if>` tags to cover the two options. (Or the `<xsl:choose>` tag could have been used, which provides case-statement functionality.)
- Single quotes are required around the attribute values. Otherwise, the XSLT processor attempts to interpret the word `ordered` as an XPath function, instead of as a string.

Now finish up `LIST` processing by handling `ITEM` elements:

```
  <xsl:template match="ITEM">
    <li><xsl:apply-templates/>
    </li>
  </xsl:template>

</xsl:stylesheet>
```

Ordering Templates in a Stylesheet

By now, you should have the idea that templates are independent of one another, so it doesn't generally matter where they occur in a file. So from here on, we'll just show the template you need to add. (For the sake of comparison, they're always added at the end of the example stylesheet.)

Order *does* make a difference when two templates can apply to the same node. In that case, the one that is defined *last* is the one that is found and processed. For example, to change the ordering of an indented list to use lowercase alphabetics, you could specify a template pattern that looks like this: `//LIST//LIST`. In that template, you would use the HTML option to generate an alphabetic enumeration, instead of a numeric one.

But such an element could also be identified by the pattern `//LIST`. To make sure the proper processing is done, the template that specifies `//LIST` would have to appear *before* the template the specifies `//LIST//LIST`.

Process <NOTE> Elements

The last remaining structure element is the `NOTE` element. Add the template shown below to handle that.

```
    <xsl:template match="NOTE">
         <blockquote><b>Note:</b><br/>
      <xsl:apply-templates/>
      </p></blockquote>
    </xsl:template>

  </xsl:stylesheet>
```

This code brings up an interesting issue that results from the inclusion of the `<br/>` tag. To be well-formed XML, the tag must be specified in the stylesheet as `<br/>`, but that tag is not recognized by many browsers. And while most browsers recognize the sequence `<br></br>`, they all treat it like a paragraph break, instead of a single line break.

In other words, the transformation *must* generate a `<br>` tag, but the stylesheet must specify `<br/>`. That brings us to the major reason for that special output tag we added early in the stylesheet:

```
  <xsl:stylesheet ... >
    <xsl:output method="html"/>
    ...
  </xsl:stylesheet>
```

That output specification converts empty tags like `<br/>` to their HTML form, `<br>`, on output. That conversion is important, because most browsers do not recognize the empty tags. Here is a list of the affected tags:

```
  area         frame        isindex
  base         hr           link
  basefont     img          meta
  br           input        param
  col
```

To summarize, by default XSLT produces well-formed XML on output. And since an XSL stylesheet is well-formed XML to start with, you cannot easily put a tag like `<br>` in the middle of it. The “`<xsl:output method="html"/>`” solves the problem, so you can code `<br/>` in the stylesheet, but get `<br>` in the output.

The other major reason for specifying `<xsl:output method="html"/>` is that, as with the specification `<xsl:output method="text"/>`, generated text is *not* escaped. For example, if the stylesheet includes the `&lt;` entity reference, it will appear as the `<` character in the generated text. When XML is generated, on the other hand, the `&lt;` entity reference in the stylesheet would be unchanged, so it would appear as `&lt;` in the generated text.

Note: If you actually want `&lt;` to be generated as part of the HTML output, you'll need to encode it as `&amp;lt;`—that sequence becomes `&lt;` on output, because only the `&amp;` is converted to an `&` character.

Run the Program

Here is the HTML that is generated for the second section when you run the program now:

```
...
<h2>The Second Major Section</h2>
<p>This section adds a LIST and a NOTE.</p>
<p>Here is the LIST:</p>
<ol>
<li>Pears</li>
<li>Grapes</li>
</ol>
<p>And here is the NOTE:</p>
<blockquote>
<b>Note:</b>
<br>Don't forget to go to the hardware store on your way to the
grocery!
</blockquote>
```

Process Inline (Content) Elements

The only remaining tags in the `ARTICLE` type are the *inline* tags — the ones that don't create a line break in the output, but which instead are integrated into the stream of text they are part of.

Inline elements are different from structure elements, in that they are part of the content of a tag. If you think of an element as a node in a document tree, then each node has both *content* and *structure*. The content is composed of the text and inline tags it contains. The structure consists of the other elements (structure elements) under the tag.

Note: The sample document described in this section is `article3.xml`, and the stylesheet used to manipulate it is `article3.xsl`. The result is `stylizer3.html`. (The browser-displayable versions are `article3-xml.html`, `article3-xsl.html`, and `stylizer3-src.html`.)

Start by adding one more bit of test data to the sample document:

```
<?xml version="1.0"?>
<ARTICLE>
  <TITLE>A Sample Article</TITLE>
  <SECT>The First Major Section
     ...
  </SECT>
  <SECT>The Second Major Section
     ...
  </SECT>
  <SECT>The <I>Third</I> Major Section
     <PARA>In addition to the inline tag in the heading,
        this section defines the term <DEF>inline</DEF>,
        which literally means "no line break". It also
        adds a simple link to the main page for the Java
        platform (<LINK>http://java.sun.com</LINK>),
        as well as a link to the
        <LINK target="http://java.sun.com/xml">XML</LINK>
        page.
     </PARA>
  </SECT>
</ARTICLE>
```

Now, process the inline `<DEF>` elements in paragraphs, renaming them to HTML italics tags:

```
<xsl:template match="DEF">
  <i> <xsl:apply-templates/> </i>
</xsl:template>
```

Next, comment out the text-node normalization. It has served its purpose, and now you're to the point that you need to preserve important spaces:

```
<!--
  <xsl:template match="text()">
     <xsl:value-of select="normalize-space()"/>
  </xsl:template>
-->
```

This modification keeps us from losing spaces before tags like `<I>` and `<DEF>`. (Try the program without this modification to see the result.)

Now, process basic inline HTML elements like `<B>`, `<I>`, `<U>` for bold, italics, and underlining.

```
<xsl:template match="B|I|U">
  <xsl:element name="{name()}">
     <xsl:apply-templates/>
  </xsl:element>
</xsl:template>
```

The `<xsl:element>` tag lets you compute the element you want to generate. Here, you generate the appropriate inline tag using the name of the current element. In particular, note the use of curly braces (`{}`) in the `name=".."` expression. Those curly braces cause the text inside the quotes to be processed as an XPath expression, instead of being interpreted as a literal string. Here, they cause the XPath `name()` function to return the name of the current node.

Curly braces are recognized anywhere that an *attribute value template* can occur. (Attribute value templates are defined in section 7.6.2 of the XSLT specification, and they appear several places in the template definitions.). In such expressions, curly braces can also be used to refer to the value of an attribute, `{@foo}`, or to the content of an element `{foo}`.

Note: You can also generate attributes using `<xsl:attribute>`. For more information, see Section 7.1.3 of the XSLT Specification.

The last remaining element is the LINK tag. The easiest way to process that tag will be to set up a *named template* that we can drive with a parameter:

```
<xsl:template name="htmLink">
  <xsl:param name="dest" select="UNDEFINED"/>
  <xsl:element name="a">
    <xsl:attribute name="href">
      <xsl:value-of select="$dest"/>
    </xsl:attribute>
    <xsl:apply-templates/>
  </xsl:element>
</xsl:template>
```

The major difference in this template is that, instead of specifying a match clause, you gave the template a name with the name="" clause. So this template only gets executed when you invoke it.

Within the template, you also specified a parameter named dest, using the <xsl:param> tag. For a bit of error checking, you used the select clause to give that parameter a default value of UNDEFINED. To reference the variable in the <xsl:value-of> tag, you specified "$dest".

Note: Recall that an entry in quotes is interpreted as an expression, unless it is further enclosed in single quotes. That's why the single quotes were needed earlier, in "@type='ordered'"—to make sure that ordered was interpreted as a string.

The <xsl:element> tag generates an element. Previously, we have been able to simply specify the element we want by coding something like <html>. But here you are dynamically generating the content of the HTML anchor (<a>) in the body of the <xsl:element> tag. And you are dynamically generating the href attribute of the anchor using the <xsl:attribute> tag.

The last important part of the template is the <apply-templates> tag, which inserts the text from the text node under the LINK element. Without it, there would be no text in the generated HTML link.

Next, add the template for the LINK tag, and call the named template from within it:

```
<xsl:template match="LINK">
  <xsl:if test="@target">
    <!--Target attribute specified.-->
    <xsl:call-template name="htmLink">
      <xsl:with-param name="dest" select="@target"/>
      </xsl:call-template>
    </xsl:if>
  </xsl:template>

  <xsl:template name="htmLink">
    ...
```

The `test="@target"` clause returns true if the `target` attribute exists in the `LINK` tag. So this `<xsl-if>` tag generates HTML links when the text of the link and the target defined for it are different.

The `<xsl:call-template>` tag invokes the named template, while `<xsl:with-param>` specifies a parameter using the `name` clause, and its value using the `select` clause.

As the very last step in the stylesheet construction process, add the `<xsl-if>` tag shown below to process `LINK` tags that do not have a `target` attribute.

```
  <xsl:template match="LINK">
    <xsl:if test="@target">
      ...
    </xsl:if>

    <xsl:if test="not(@target)">
      <xsl:call-template name="htmLink">
        <xsl:with-param name="dest">
          <xsl:apply-templates/>
        </xsl:with-param>
      </xsl:call-template>
    </xsl:if>
  </xsl:template>
```

The `not(...)` clause inverts the previous test (remember, there is no `else` clause). So this part of the template is interpreted when the `target` attribute is not specified. This time, the parameter value comes not from a `select` clause, but from the *contents* of the `<xsl:with-param>` element.

Note: Just to make it explicit: Parameters and variables (which are discussed in a few moments in Appendix 8, What Else Can XSLT Do?What Else Can XSLT Do? (page 349) can have their value specified *either* by a `select` clause, which lets you use XPath expressions, *or* by the content of the element, which lets you use XSLT tags.

The content of the parameter, in this case, is generated by the `<xsl:apply-templates/>` tag, which inserts the contents of the text node under the `LINK` element.

Run the Program

When you run the program now, the results should look something like this:

```
...
<h2>The <I>Third</I> Major Section
    </h2>
<p>In addition to the inline tag in the heading, this section
         defines the term <i>inline</i>, which literally means
         "no line break". It also adds a simple link to the
         main page for the Java platform (<a
         href="http://java.sun.com">http://java.sun.com</a>),
         as well as a link to the
         <a href="http://java.sun.com/xml">XML</a> page.
  </p>
```

Good work! You have now converted a rather complex XML file to HTML. (As seemingly simple as it appear at first, it certainly provided a lot of opportunity for exploration.)

Printing the HTML

You have now converted an XML file to HTML. One day, someone will produce an HTML-aware printing engine that you'll be able to find and use through the Java Printing Service API. At that point, you'll have ability to print an arbitrary XML file by generating HTML—all you'll have to do is set up a stylesheet and use your browser.

What Else Can XSLT Do?

As lengthy as this section of the tutorial has been, it has still only scratched the surface of XSLT's capabilities. Many additional possibilities await you in the XSLT Specification. Here are a few of the things to look for:

import (Section 2.6.2) and include (Section 2.6.1)
Use these statements to modularize and combine XSLT stylesheets. The `include` statement simply inserts any definitions from the included file. The

`import` statement lets you override definitions in the imported file with definitions in your own stylesheet.

for-each loops (Section 8)
Loop over a collection of items and process each one, in turn.

choose (case statement) for conditional processing (Section 9.2)
Branch to one of multiple processing paths depending on an input value.

generating numbers (Section 7.7)
Dynamically generate numbered sections, numbered elements, and numeric literals. XSLT provides three numbering modes:

- **single**: Numbers items under a single heading, like an ordered list in HTML.
- **multiple:** Produces multi-level numbering like “A.1.3”.
- **any:** Consecutively numbers items wherever they appear, as with footnotes in a chapter.

formatting numbers (Section 12.3)
Control enumeration formatting, so you get numerics (`format="1"`), uppercase alphabetics (`format="A"`), lowercase alphabetics (`format="a"`), or compound numbers, like “A.1”, as well as numbers and currency amounts suited for a specific international locale.

sorting output (Section 10)
Produce output in some desired sorting order.

mode-based templates (Section 5.7)
Process an element multiple times, each time in a different “mode”. You add a `mode` attribute to templates, and then specify `<apply-templates mode="...">` to apply only the templates with a matching mode. Combine with the `<apply-templates select="...">` attribute to apply mode-based processing to a subset of the input data.

variables (Section 11)
Variables, like parameters, let you control a template’s behavior. But they are not as valuable as you might think. The value of a variable is only known within the scope of the current template or `<xsl:if>` tag (for example) in which it is defined. You can’t pass a value from one template to another, or even from an enclosed part of a template to another part of the same template.

These statements are true even for a “global” variable. You can change its value in a template, but the change only applies to that template. And when the expression used to define the global variable is evaluated, that evaluation takes place in the context of the structure’s root node. In other words, global variables are essentially runtime constants. Those constants can be useful for changing the behavior of a template, especially when coupled with `include` and `import` statements. But variables are not a general-purpose data-management mechanism.

The Trouble with Variables

It is tempting to create a single template and set a variable for the destination of the link, rather than go to the trouble of setting up a parameterized template and calling it two different ways. The idea would be to set the variable to a default value (say, the text of the `LINK` tag) and then, if `target` attribute exists, set the destination variable to the value of the `target` attribute.

That would be a good idea—if it worked. But once again, the issue is that variables are only known in the scope within which they are defined. So when you code an `<xsl:if>` tag to change the value of the variable, the value is only known within the context of the `<xsl:if>` tag. Once `</xsl:if>` is encountered, any change to the variable's setting is lost.

A similarly tempting idea is the possibility of replacing the `text()|B|I|U|DEF|LINK` specification with a variable (`$inline`). But since the value of the variable is determined by where it is defined, the value of a global `inline` variable consists of text nodes, `<B>` nodes, and so on, that happen to exist at the root level. In other words, the value of such a variable, in this case, is null.

Transforming from the Command Line

When you are running a transformation from the command line, it makes a lot of sense to use XSLTC. Although the Xalan interpreting transformer contains a command-line mechanism as well, it doesn't save the pre-compiled byte-codes as *translets* for later use, as XSLTC does.

There are two steps to running XSLTC from the command line:

1. Compile the translet.
2. Run the compiled translet on the data.

Note: For detailed information on this subject, you can also consult the excellent usage guide at `http://xml.apache.org/xalan-j/xsltc_usage.html`.

Compiling the Translet

To compile the `article3.xsl` stylesheet into a translet, execute this command:

```
java org.apache.xalan.xsltc.cmdline.Compile article3.xsl
```

Note: For version 1.3 of the Java platform, you'll need to include the appropriate classpath settings, as described in Compiling and Running the Program (page 143).

The result is a class file (the translet) named `article3.class`.

Here are the arguments that can be specified when compiling a translet:

```
java org.apache.xalan.xsltc.cmdline.Compile
     -o transletName -d directory -j jarFile
     -p packageName {-u stylesheetURI | stylesheetFile }
```

where:

- -o *transletName*

  Specifies the name of the generated translet class (the output class). The `.class` suffix is optional. If not present, it is automatically added to the name specified by the *stylesheet* argument.

- -d *directory*

  Specifies the destination directory.
  (Default is the current working directory.)

- -j *jarFile*

  Outputs the generated translet class files into a JAR file named *jarFile*.jar. When this option is used, only the JAR file is created.

- -p *packageName*

  Specifies a package name for the generated translet classes.

- -u *stylesheetURI*

  Specifies the stylesheet with a URI such as `http://myserver/stylesheet1.xsl`.

- *stylesheetFile*

  (No flag) The pathname of the stylesheet file.

Running the Translet

To run the compiled translet on the sample file `article3.xml`, execute this command:

```
java org.apache.xalan.xsltc.cmdline.Transform
     article3.xml article3
```

Note: Again set the classpath, as described in Compiling and Running the Program (page 143), if you are running on version 1.3 of the Java platform.

This command adds the current directory to the classpath, so the translet can be found. The output goes to `System.out`.

Here are the possible arguments that can be specified when running a translet:

```
java org.apache.xalan.xsltc.cmdline.Transform
     {-u documentURI | documentFilename}
     className [name=value...]
```

where:

- -u *`documentURI`*

  Specifies the XML input document with a URI.
- *`documentFilename`*

  Specifies the filename for an XML input document.
- *`className`*

  The translet that performs the transformation. (Here, you can't specify the `.class` suffix, the same way you omit it when running a java application.)
- *`name=value`* ...

  Optional set of one or more stylesheet parameters specified as name-value pairs.

Concatenating Transformations with a Filter Chain

It is sometimes useful to create a *filter chain* — a concatenation of XSLT transformations in which the output of one transformation becomes the input of the next. This section of the tutorial shows you how to do that.

Writing the Program

Start by writing a program to do the filtering. This example will show the full source code, but you can use one of the programs you've been working on as a basis, to make things easier.

Note: The code described here is contained in `FilterChain.java`.

The sample program includes the import statements that identify the package locations for each class:

```
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;

import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.InputSource;
import org.xml.sax.XMLReader;
import org.xml.sax.XMLFilter;

import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.TransformerConfigurationException;

import javax.xml.transform.sax.SAXTransformerFactory;
import javax.xml.transform.sax.SAXSource;
import javax.xml.transform.sax.SAXResult;

import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;

import java.io.*;
```

The program also includes the standard error handlers you’re used to. They’re listed here, just so they are all gathered together in one place:

```
}
catch (TransformerConfigurationException tce) {
  // Error generated by the parser
  System.out.println ("* Transformer Factory error");
  System.out.println("   " + tce.getMessage() );

  // Use the contained exception, if any
  Throwable x = tce;
  if (tce.getException() != null)
     x = tce.getException();
  x.printStackTrace();
}
catch (TransformerException te) {
  // Error generated by the parser
  System.out.println ("* Transformation error");
  System.out.println("   " + te.getMessage() );

  // Use the contained exception, if any
  Throwable x = te;
  if (te.getException() != null)
     x = te.getException();
  x.printStackTrace();
}
catch (SAXException sxe) {
  // Error generated by this application
  // (or a parser-initialization error)
  Exception  x = sxe;
  if (sxe.getException() != null)
     x = sxe.getException();
  x.printStackTrace();
}
catch (ParserConfigurationException pce) {
  // Parser with specified options can't be built
  pce.printStackTrace();
}
catch (IOException ioe) {
  // I/O error
  ioe.printStackTrace();
}
```

In between the import statements and the error handling, the core of the program consists of the code shown below.

```
public static void main (String argv[])
{
  if (argv.length != 3) {
    System.err.println (
      "Usage: java FilterChain style1 style2 xmlfile");
    System.exit (1);
  }

   try {
    // Read the arguments
    File stylesheet1 = new File(argv[0]);
    File stylesheet2 = new File(argv[1]);
    File datafile = new File(argv[2]);

     // Set up the input stream
    BufferedInputStream bis = new
      BufferedInputStream(newFileInputStream(datafile));
    InputSource input = new InputSource(bis);

     // Set up to read the input file (see Note #1)
    SAXParserFactory spf = SAXParserFactory.newInstance();
    spf.setNamespaceAware(true);
    SAXParser parser = spf.newSAXParser();
    XMLReader reader = parser.getXMLReader();

     // Create the filters (see Note #2)
    SAXTransformerFactory stf =
      (SAXTransformerFactory)
        TransformerFactory.newInstance();
    XMLFilter filter1 = stf.newXMLFilter(
      new StreamSource(stylesheet1));
    XMLFilter filter2 = stf.newXMLFilter(
      new StreamSource(stylesheet2));

    // Wire the output of the reader to filter1 (see Note #3)
    // and the output of filter1 to filter2
    filter1.setParent(reader);
    filter2.setParent(filter1);

     // Set up the output stream
    StreamResult result = new StreamResult(System.out);

  // Set up the transformer to process the SAX events generated
  // by the last filter in the chain
    Transformer transformer = stf.newTransformer();
```

```
      SAXSource transformSource = new SAXSource(
         filter2, input);
      transformer.transform(transformSource, result);
   } catch (...) {
      ...
```

Notes:

1. The Xalan transformation engine currently requires a namespace-aware SAX parser. XSLTC does not make that requirement.
2. This weird bit of code is explained by the fact that `SAXTransformerFactory` extends `TransformerFactory`, adding methods to obtain filter objects. The `newInstance()` method is a static method defined in `TransformerFactory`, which (naturally enough) returns a `TransformerFactory` object. In reality, though, it returns a `SAXTransformerFactory`. So, to get at the extra methods defined by `SAXTransformerFactory`, the return value must be cast to the actual type.
3. An `XMLFilter` object is both a SAX reader and a SAX content handler. As a SAX reader, it generates SAX events to whatever object has registered to receive them. As a content handler, it consumes SAX events generated by its “parent” object — which is, of necessity, a SAX reader, as well. (Calling the event generator a “parent” must make sense when looking at the internal architecture. From an external perspective, the name doesn’t appear to be particularly fitting.) The fact that filters both generate and consume SAX events allows them to be chained together.

Understanding How the Filter Chain Works

The code listed above shows you how to set up the transformation. Figure 8–2 should help you understand what’s happening when it executes.

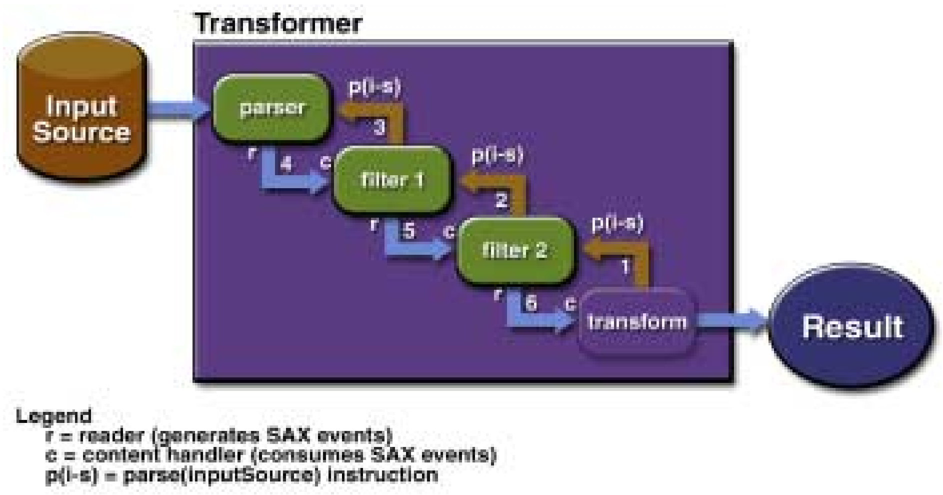

Transformer
Input Source
parser
p(i-s)
3
r
4
c
filter 1
p(i-s)
2
r
5
c
filter 2
p(i-s)
1
r
6
c
transform
Result
Legend
r = reader (generates SAX events)
c = content handler (consumes SAX events)
p(i-s) = parse(inputSource) instruction

Figure 8–2 Operation of Chained Filters

When you create the transformer, you pass it at a `SAXSource` object, which encapsulates a reader (in this case, `filter2`) and an input stream. You also pass it a pointer to the result stream, where it directs its output. The diagram shows what happens when you invoke `transform()` on the transformer. Here is an explanation of the steps:

1. The transformer sets up an internal object as the content handler for `filter2`, and tells it to parse the input source.
2. `filter2`, in turn, sets itself up as the content handler for `filter1`, and tells *it* to parse the input source.
3. `filter1`, in turn, tells the `parser` object to parse the input source.
4. The `parser` does so, generating SAX events which it passes to `filter1`.
5. `filter1`, acting in its capacity as a content handler, processes the events and does its transformations. Then, acting in its capacity as a SAX reader (`XMLReader`), it sends SAX events to `filter2`.
6. `filter2` does the same, sending its events to the transformer's content handler, which generates the output stream.

Testing the Program

To try out the program, you'll create an XML file based on a tiny fraction of the XML DocBook format, and convert it to the ARTICLE format defined here. Then you'll apply the ARTICLE stylesheet to generate an HTML version.

Note: This example processes small-docbook-article.xml using docbookToArticle.xsl and article1c.xsl. The result is filterout.html (The browser-displayable versions are small-docbook-article-xml.html, docbookToArticle-xsl.html, article1c-xsl.html, and filterout-src.html.) See the O'Reilly Web pages for a good description of the DocBook article format.

Start by creating a small article that uses a minute subset of the XML DocBook format:

```
<?xml version="1.0"?>
<Article>
  <ArtHeader>
    <Title>Title of my (Docbook) article</Title>
  </ArtHeader>
  <Sect1>
    <Title>Title of Section 1.</Title>
    <Para>This is a paragraph.</Para>
  </Sect1>
</Article>
```

Next, create a stylesheet to convert it into the ARTICLE format:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0"
  >
  <xsl:output method="xml"/> (see Note #1)

   <xsl:template match="/">
    <ARTICLE>
      <xsl:apply-templates/>
    </ARTICLE>
  </xsl:template>

  <!-- Lower level titles strip element tag --> (see Note #2)

  <!-- Top-level title -->
  <xsl:template match="/Article/ArtHeader/Title"> (Note #3)
```

```
      <TITLE> <xsl:apply-templates/> </TITLE>
   </xsl:template>

    <xsl:template match="//Sect1"> (see Note #4)
      <SECT><xsl:apply-templates/></SECT>
   </xsl:template>

    <xsl:template match="Para">
      <PARA><xsl:apply-templates/></PARA> (see Note #5)
   </xsl:template>

</xsl:stylesheet>
```

Notes:

1. This time, the stylesheet is generating XML output.
2. The template that follows (for the top-level title element) matches only the main title. For section titles, the `TITLE` tag gets stripped. (Since no template conversion governs those title elements, they are ignored. The text nodes they contain, however, are still echoed as a result of XSLT's built in template rules— so only the tag is ignored, not the text. More on that below.)
3. The title from the DocBook article header becomes the `ARTICLE` title.
4. Numbered section tags are converted to plain `SECT` tags.
5. This template carries out a case conversion, so `Para` becomes `PARA`.

Although it hasn't been mentioned explicitly, XSLT defines a number of built-in (default) template rules. The complete set is listed in Section 5.8 of the specification. Mainly, they provide for the automatic copying of text and attribute nodes, and for skipping comments and processing instructions. They also dictate that inner elements are processed, even when their containing tags don't have templates. That is the reason that the text node in the section title is processed, even though the section title is not covered by any template.

Now, run the `FilterChain` program, passing it the stylesheet above (`docbook-ToArticle.xsl`), the `ARTICLE` stylesheet (`article1c.xsl`), and the small Doc-

Book file (`small-docbook-article.xml`), in that order. The result should like this:

```
<html>
<body>
<h1 align="center">Title of my (Docbook) article</h1>
<h2>Title of Section 1.</h2>
<p>This is a paragraph.</p>
</body>
</html>
```

Note: ***This output was generated using JAXP 1.0. However, the first filter in the chain is not currently translating any of the tags in the input file. Until that defect is fixed, the output you see will consist of concatenated plain text in the HTML output, like this:*** **"**`Title of my (Docbook) article Title of Section 1. This is a paragraph.`**".**

Conclusion

Congratulations! You have completed the XSLT tutorial. There is a lot you do with XML and XSLT, and you are now prepared to explore the many exciting possibilities that await.

Further Information

For more information on XSL stylesheets, XSLT, and transformation engines, see:

- A great introduction to XSLT that starts with a simple HTML page and uses XSLT to customize it, one step at a time: `http://www.xfront.com/rescuing-xslt.html`
- Extensible Stylesheet Language (XSL): `http://www.w3.org/Style/XSL/`
- The XML Path Language: `http://www.w3.org/TR/xpath`
- The Xalan transformation engine: `http://xml.apache.org/xalan-j/`
- The XSLTC transformation engine: `http://xml.apache.org/xalan-j/`
- Tips for using XSLTC: `http://xml.apache.org/xalan-j/xsltc_usage.html`

- Designing stylesheets to maximize performance with XSLTC: `http://xml.apache.org/xalan-j/xsltc/xsltc_performance.html`

9

Binding XML Schema to Java Classes with JAXB

Scott Fordin

THE Java™ Architecture for XML Binding (JAXB) is a Java technology that enables you to generate Java classes from XML schemas. As part of this process, JAXB also provides methods for unmarshalling XML instance documents into Java content trees, and then marshalling Java content trees back into XML instance documents. Put another way, JAXB provides a fast and convenient way to bind XML schemas to Java representations, making it easy for Java developers to incorporate XML data and processing functions in Java applications.

What this all means is that you can leverage the flexibility of platform-neutral XML data in Java applications without having to deal with or even know XML programming techniques. Moreover, you can take advantage of XML strengths without having to rely on heavyweight, complex XML processing models like SAX or DOM. JAXB hides the details and gets rid of the extraneous relationships in SAX and DOM—generated JAXB classes describe only the relationships actually defined in the source schemas. The result is highly portable XML data joined with highly portable Java code that can be used to create flexible, lightweight applications and Web services.

This chapter describes the JAXB architecture, functions, and core concepts. You should read this chapter before proceeding to Chapter 10, which provides sample code and step-by-step procedures for using JAXB.

JAXB Architecture

This section describes the components and interactions in the JAXB processing model. After providing a general overview, this section goes into more detail about core JAXB features. The topics in this section include:

- Architectural Overview
- The JAXB Binding Process
- JAXB Binding Framework
- More About javax.xml.bind
- More About Unmarshalling
- More About Marshalling
- More About Validation

Architectural Overview

Figure 9–1 shows the components that make up a JAXB implementation.

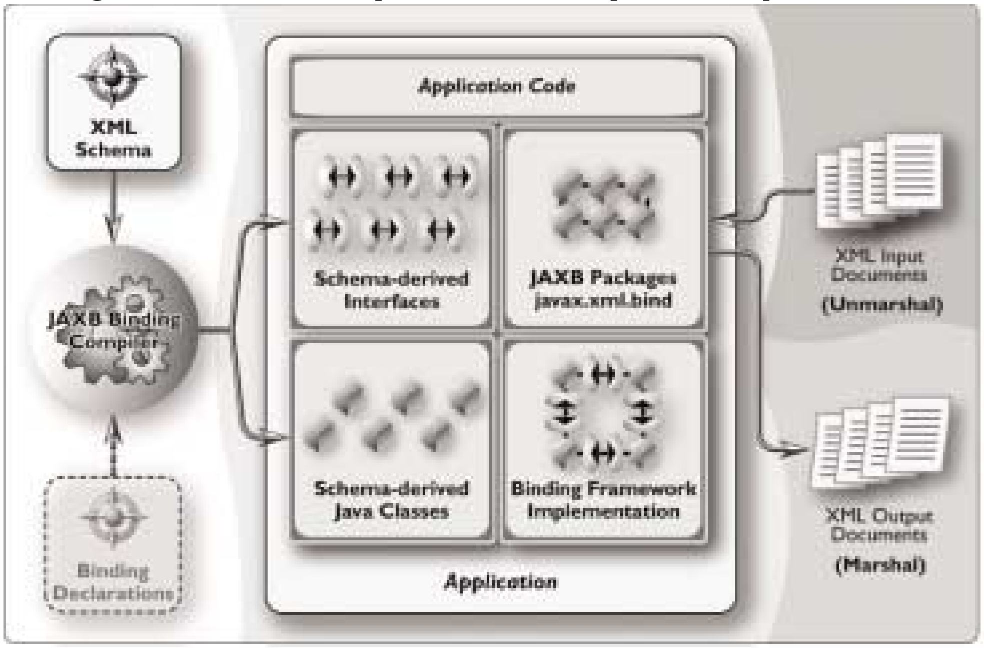

XML Schema
JAXB Binding Compiler
Binding Declarations
Application Code
Schema-derived Interfaces
JAXB Packages javax.xml.bind
Schema-derived Java Classes
Binding Framework Implementation
Application
XML Input Documents (Unmarshal)
XML Output Documents (Marshal)

Figure 9–1 JAXB Architectural Overview

As shown in Figure 9–1, a JAXB implementation comprises the following eight core components.

Table 9–1 Core Components in a JAXB Implementation

Component	**Description**
XML Schema	An XML schema uses XML syntax to describe the relationships among elements, attributes and entities in an XML document. The purpose of an XML schema is to define a class of XML documents that must adhere to a particular set of structural rules and data constraints. For example, you may want to define separate schemas for chapter-oriented books, for an online purchase order system, or for a personnel database. In the context of JAXB, an XML document containing data that is constrained by an XML schema is referred to as a *document instance*, and the structure and data within a document instance is referred to as a *content tree*.
Binding Declarations	By default, the JAXB binding compiler binds Java classes and packages to a source XML schema based on rules defined in Section 5, "Binding XML Schema to Java Representations," in the *JAXB Specification*. In most cases, the default binding rules are sufficient to generate a robust set of schema-derived classes from a wide range of schemas. There may be times, however, when the default binding rules are not sufficient for your needs. JAXB supports customizations and overrides to the default binding rules by means of *binding declarations* made either inline as annotations in a source schema, or as statements in an external binding customization file that is passed to the JAXB binding compiler. Note that custom JAXB binding declarations also allow you to customize your generated JAXB classes beyond the XML-specific constraints in an XML schema to include Java-specific refinements such as class and package name mappings.
Binding Compiler	The JAXB binding compiler is the core of the JAXB processing model. Its function is to transform, or bind, a source XML schema to a set of JAXB *content classes* in the Java programming language. Basically, you run the JAXB binding compiler using an XML schema (optionally with custom binding declarations) as input, and the binding compiler generates Java classes that map to constraints in the source XML schema.
Binding Framework Implementation	The JAXB binding framework implementation is a runtime API that provides interfaces for unmarshalling, marshalling, and validating XML content in a Java application. The binding framework comprises interfaces in the `javax.xml.bind` package.
Schema-Derived Classes	These are the schema-derived classes generated by the binding JAXB compiler. The specific classes will vary depending on the input schema.

Table 9–1 Core Components in a JAXB Implementation (Continued)

Component	Description
Java Application	In the context of JAXB, a Java application is a client application that uses the JAXB binding framework to unmarshal XML data, validate and modify Java content objects, and marshal Java content back to XML data. Typically, the JAXB binding framework is wrapped in a larger Java application that may provide UI features, XML transformation functions, data processing, or whatever else is desired.
XML Input Documents	XML content that is unmarshalled as input to the JAXB binding framework -- that is, an XML instance document, from which a Java representation in the form of a content tree is generated. In practice, the term "document" may not have the conventional meaning, as an XML instance document does not have to be a completely formed, selfstanding document file; it can instead take the form of streams of data passed between applications, or of sets of database fields, or of *XML infosets*, in which blocks of information contain just enough information to describe where they fit in the schema structure. In JAXB, the unmarshalling process supports *validation* of the XML input document against the constraints defined in the source schema. This validation process is optional, however, and there may be cases in which you know by other means that an input document is valid and so you may choose for performance reasons to skip validation during unmarshalling. In any case, validation before (by means of a third-party application) or during unmarshalling is important, because it assures that an XML document generated during marshalling will also be valid with respect to the source schema. Validation is discussed more later in this chapter.
XML Output Documents	XML content that is marshalled out to an XML document. In JAXB, marshalling involves parsing an XML content object tree and writing out an XML document that is an accurate representation of the original XML document, and is valid with respect the source schema. JAXB can marshal XML data to XML documents, SAX content handlers, and DOM nodes.

The JAXB Binding Process

Figure 9–2 shows what occurs during the JAXB binding process.

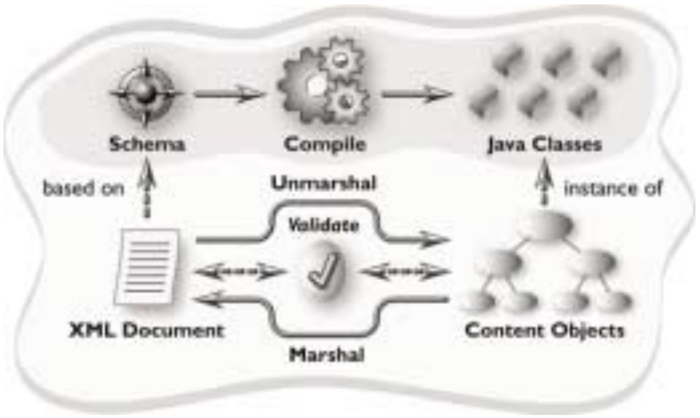


Figure 9–2 Steps in the JAXB Binding Process

The general steps in the JAXB data binding process are:

1. Generate classes. An XML schema is used as input to the JAXB binding compiler to generate JAXB classes based on that schema.
2. Compile classes. All of the generated classes, source files, and application code must be compiled.
3. Unmarshal. XML documents written according to the constraints in the source schema are unmarshalled by the JAXB binding framework. Note that JAXB also supports unmarshalling XML data from sources other than files/documents, such as DOM nodes, string buffers, SAX Sources, and so forth.
4. Generate content tree. The unmarshalling process generates a content tree of data objects instantiated from the generated JAXB classes; this content tree represents the structure and content of the source XML documents.
5. Validate (optional). The unmarshalling process optionally involves validation of the source XML documents before generating the content tree. Note that if you modify the content tree in Step 6, below, you can also use the JAXB Validate operation to validate the changes before marshalling the content back to an XML document.

6. Process content. The client application can modify the XML data represented by the Java content tree by means of interfaces generated by the binding compiler.
7. Marshal. The processed content tree is marshalled out to one or more XML output documents. The content may be validated before marshalling.

To summarize, using JAXB involves two discrete sets of activities:

- Generate and compile JAXB classes from a source schema, and build an application that implements these classes
- Run the application to unmarshal, process, validate, and marshal XML content through the JAXB binding framework

These two steps are usually performed at separate times in two distinct phases. Typically, for example, there is an application development phase in which JAXB classes are generated and compiled, and a binding implementation is built, followed by a deployment phase in which the generated JAXB classes are used to process XML content in an ongoing "live" production setting.

Note: Unmarshalling is not the only means by which a content tree may be created. Schema-derived content classes also support the programmatic construction of content trees by direct invocation of the appropriate factory methods. Once created, a content tree may be revalidated, either in whole or in part, at any time. See Sample Application 3 (page 414) for an example of using the `ObjectFactory` class to directly add content to a content tree.

JAXB Binding Framework

The JAXB binding framework is implemented in three Java packages:

- The `javax.xml.bind` package defines abstract classes and interfaces that are used directly with content classes.

  The `javax.xml.bind` package defines the `Unmarshaller`, `Validator`, and `Marshaller` classes, which are auxiliary objects for providing their respective operations.

  The `JAXBContext` class is the entry point for a Java application into the JAXB framework. A `JAXBContext` instance manages the binding relationship between XML element names to Java content interfaces for a JAXB implementation to be used by the unmarshal, marshal and validation operations.

The `javax.xml.bind` package also defines a rich hierarchy of validation event and exception classes for use when marshalling or unmarshalling errors occur, when constraints are violated, and when other types of errors are detected.

- The `javax.xml.bind.util` package contains utility classes that may be used by client applications to manage marshalling, unmarshalling, and validation events.
- The `javax.xml.bind.helper` package provides partial default implementations for some of the javax.xml.bind interfaces. Implementations of JAXB can extend these classes and implement the abstract methods. These APIs are not intended to be directly used by applications using JAXB architecture.

The main package in the JAXB binding framework, `javax.bind.xml`, is described in more detail below.

More About javax.xml.bind

The three core functions provided by the primary binding framework package, `javax.xml.bind`, are marshalling, unmarshalling, and validation. The main client entry point into the binding framework is the `JAXBContext` class.

`JAXBContext` provides an abstraction for managing the XML/Java binding information necessary to implement the unmarshal, marshal and validate operations. A client application obtains new instances of this class by means of the `newInstance(contextPath)` method; for example:

```
JAXBContext jc = JAXBContext.newInstance(
"com.acme.foo:com.acme.bar" );
```

The `contextPath` parameter contains a list of Java package names that contain schema-derived interfaces—specifically the interfaces generated by the JAXB binding compiler. The value of this parameter initializes the `JAXBContext` object to enable management of the schema-derived interfaces. To this end, the JAXB provider implementation must supply an implementation class containing a method with the following signature:

```
public static JAXBContext createContext( String contextPath,
ClassLoader classLoader )

        throws JAXBException;
```

Note: The JAXB provider implementation must generate a `jaxb.properties` file in each package containing schema-derived classes. This property file must contain a property named `javax.xml.bind.context.factory` whose value is the name of the class that implements the `createContext` API.

The class supplied by the provider does not have to be assignable to `javax.xml.bind.JAXBContext`, it simply has to provide a class that implements the `createContext` API. By allowing for multiple Java packages to be specified, the `JAXBContext` instance allows for the management of multiple schemas at one time.

More About Unmarshalling

The `Unmarshaller` class in the `javax.xml.bind` package provides the client application the ability to convert XML data into a tree of Java content objects. The `unmarshal` method for a schema (within a namespace) allows for any global XML element declared in the schema to be unmarshalled as the root of an instance document. The `JAXBContext` object allows the merging of global elements across a set of schemas (listed in the `contextPath`). Since each schema in the schema set can belong to distinct namespaces, the unification of schemas to an unmarshalling context should be namespace-independent. This means that a client application is able to unmarshal XML documents that are instances of any of the schemas listed in the `contextPath`; for example:

```
JAXBContext jc = JAXBContext.newInstance(
  "com.acme.foo:com.acme.bar" );

Unmarshaller u = jc.createUnmarshaller();

FooObject fooObj =
  (FooObject)u.unmarshal( new File( "foo.xml" ) ); // ok

BarObject barObj =
  (BarObject)u.unmarshal( new File( "bar.xml" ) ); // ok

BazObject bazObj =
  (BazObject)u.unmarshal( new File( "baz.xml" ) );
  // error, "com.acme.baz" not in contextPath
```

A client application may also generate Java content trees explicitly rather than unmarshalling existing XML data. To do so, the application needs to have access and knowledge about each of the schema-derived `ObjectFactory` classes that exist in each of Java packages contained in the `contextPath`. For each schema-derived Java class, there will be a static factory method that produces objects of that type. For example, assume that after compiling a schema, you have a package `com.acme.foo` that contains a schema-derived interface named `PurchaseOrder`. To create objects of that type, the client application would use the following factory method:

```
ObjectFactory objFactory = new ObjectFactory();

com.acme.foo.PurchaseOrder po =
  objFactory.createPurchaseOrder();
```

Note: Because multiple `ObjectFactory` classes are generated when there are multiple packages on the `contextPath`, if you have multiple packages on the `contextPath`, you should use the complete package name when referencing an `ObjectFactory` class in one of those packages.

Once the client application has an instance of the schema-derived object, it can use the mutator methods to set content on it.

Note: The JAXB provider implementation must generate a class in each package that contains all of the necessary object factory methods for that package named `ObjectFactory` as well as the `newInstance( javaContentInterface )` method.

More About Marshalling

The `Marshaller` class in the `javax.xml.bind` package provides the client application the ability to convert a Java content tree back into XML data. There is no difference between marshalling a content tree that is created manually using the factory methods and marshalling a content tree that is the result an unmarshal operation. Clients can marshal a Java content tree back to XML data to a `java.io.OutputStream` or a `java.io.Writer`. The marshalling process can alternatively produce SAX2 event streams to a registered `ContentHandler` or produce a DOM `Node` object.

A simple example that unmarshals an XML document and then marshals it back out is a follows:

```
JAXBContext jc = JAXBContext.newInstance( "com.acme.foo" );

// unmarshal from foo.xml
Unmarshaller u = jc.createUnmarshaller();
FooObject fooObj =
  (FooObject)u.unmarshal( new File( "foo.xml" ) );

// marshal to System.out
Marshaller m = jc.createMarshaller();
m.marshal( fooObj, System.out );
```

By default, the `Marshaller` uses UTF-8 encoding when generating XML data to a `java.io.OutputStream` or a `java.io.Writer`. Use the `setProperty` API to change the output encoding used during these marshal operations. Client applications are expected to supply a valid character encoding name as defined in the W3C XML 1.0 Recommendation (`http://www.w3.org/TR/2000/REC-xml-20001006#charencoding`) and supported by your Java Platform.

Client applications are not required to validate the Java content tree prior to calling one of the marshal APIs. There is also no requirement that the Java content tree be valid with respect to its original schema in order to marshal it back into XML data. Different JAXB Providers can support marshalling invalid Java content trees at varying levels, however all JAXB providers must be able to marshal a valid content tree back to XML data. A JAXB provider must throw a `MarshalException` when it is unable to complete the marshal operation due to invalid content. Some JAXB providers will fully allow marshalling invalid content, others will fail on the first validation error.

Table 9–2 shows the properties that the `Marshaller` class supports.

Table 9–2 Marshaller Properties

Property	**Description**
`jaxb.encoding`	Value must be a `java.lang.String`; the output encoding to use when marshalling the XML data. The `Marshaller` will use “UTF-8” by default if this property is not specified.

Table 9–2 Marshaller Properties (Continued)

Property	**Description**
`jaxb.formatted.output`	Value must be a `java.lang.Boolean`; controls whether or not the `Marshaller` will format the resulting XML data with line breaks and indentation. A `true` value for this property indicates human readable indented XML data, while a `false` value indicates unformatted XML data. The `Marshaller` defaults to `false` (unformatted) if this property is not specified.
`jaxb.schemaLocation`	Value must be a `java.lang.String`; allows the client application to specify an `xsi:schemaLocation` attribute in the generated XML data. The format of the `schemaLocation` attribute value is discussed in an easy to understand, non-normative form in Section 5.6 of the *W3C XML Schema Part 0: Primer* and specified in Section 2.6 of the *W3C XML Schema Part 1: Structures*.
`jaxb.noNamespaceSchemaLocation`	Value must be a `java.lang.String`; allows the client application to specify an `xsi:noNamespaceSchemaLocation` attribute in the generated XML data.

More About Validation

The `Validator` class in the `javax.xml.bind` package is responsible for controlling the validation of content trees during runtime. When the unmarshalling process incorporates validation and it successfully completes without any validation errors, both the input document and the resulting content tree are guaranteed to be valid. By contrast, the marshalling process does not actually perform validation. If only validated content trees are marshalled, this guarantees that generated XML documents are always valid with respect to the source schema.

Some XML parsers, like SAX and DOM, allow schema validation to be disabled, and there are cases in which you may want to disable schema validation to improve processing speed and/or to process documents containing invalid or incomplete content. JAXB supports these processing scenarios by means of the exception handling you choose implement in your JAXB-enabled application. In general, if a JAXB implementation cannot unambiguously complete unmarshalling or marshalling, it will terminate processing with an exception.

Note: The `Validator` class is responsible for managing On-Demand Validation (see below). The `Unmarshaller` class is responsible for managing Unmarshal-Time Validation during the unmarshal operations. Although there is no formal method of enabling validation during the marshal operations, the `Marshaller` may detect errors, which will be reported to the `ValidationEventHandler` registered on it.

A JAXB client can perform two types of validation:

- **Unmarshal-Time validation** enables a client application to receive information about validation errors and warnings detected while unmarshalling XML data into a Java content tree, and is completely orthogonal to the other types of validation. To enable or disable it, use the `Unmarshaller.setValidating` method. All JAXB Providers are required to support this operation.
- **On-Demand validation** enables a client application to receive information about validation errors and warnings detected in the Java content tree. At any point, client applications can call the `Validator.validate` method on the Java content tree (or any sub-tree of it). All JAXB Providers are required to support this operation.

If the client application does not set an event handler on its `Validator`, `Unmarshaller`, or `Marshaller` prior to calling the validate, unmarshal, or marshal methods, then a default event handler will receive notification of any errors or warnings encountered. The default event handler will cause the current operation to halt after encountering the first error or fatal error (but will attempt to continue after receiving warnings).

There are three ways to handle events encountered during the unmarshal, validate, and marshal operations:

- Use the default event handler.

  The default event handler will be used if you do not specify one via the `setEventHandler` APIs on `Validator`, `Unmarshaller`, or `Marshaller`.

- Implement and register a custom event handler.

  Client applications that require sophisticated event processing can implement the `ValidationEventHandler` interface and register it with the `Unmarshaller` and/or `Validator`.

- Use the `ValidationEventCollector` utility.

  For convenience, a specialized event handler is provided that simply collects any `ValidationEvent` objects created during the unmarshal, validate, and marshal operations and returns them to the client application as a `java.util.Collection`.

Validation events are handled differently, depending on how the client application is configured to process them. However, there are certain cases where a JAXB Provider indicates that it is no longer able to reliably detect and report errors. In these cases, the JAXB Provider will set the severity of the `ValidationEvent` to `FATAL_ERROR` to indicate that the unmarshal, validate, or marshal operations should be terminated. The default event handler and `ValidationEventCollector` utility class must terminate processing after being notified of a fatal error. Client applications that supply their own `ValidationEventHandler` should also terminate processing after being notified of a fatal error. If not, unexpected behavior may occur.

XML Schemas

Because XML schemas are such an important component of the JAXB processing model—and because other data binding facilities like JAXP work with DTDs instead of schemas—it is useful to review here some basics about what XML schemas are and how they work.

XML Schemas are a powerful way to describe allowable elements, attributes, entities, and relationships in an XML document. A more robust alternative to DTDs, the purpose of an XML schema is to define classes of XML documents that must adhere to a particular set of structural and data constraints—that is, you may want to define separate schemas for chapter-oriented books, for an online purchase order system, or for a personnel database. In the context of JAXB, an XML document containing data that is constrained by an XML schema is referred to as a *document instance*, and the structure and data within a document instance is referred to as a *content tree*.

Note: In practice, the term “document” is not always accurate, as an XML instance document does not have to be a completely formed, selfstanding document file; it can instead take the form of streams of data passed between applications, or of sets of database fields, or of *XML infosets* in which blocks of information contain just enough information to describe where they fit in the schema structure.

The following sample code is taken from the W3C's *Schema Part 0: Primer* (`http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/`), and illustrates an XML document, `po.xml`, for a simple purchase order.

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
  <shipTo country="US">
     <name>Alice Smith</name>
     <street>123 Maple Street</street>
     <city>Mill Valley</city>
     <state>CA</state>
     <zip>90952</zip>
  </shipTo>
  <billTo country="US">
     <name>Robert Smith</name>
     <street>8 Oak Avenue</street>
     <city>Old Town</city>
     <state>PA</state>
     <zip>95819</zip>
  </billTo>
<comment>Hurry, my lawn is going wild!</comment>
  <items>
     <item partNum="872-AA">
        <productName>Lawnmower</productName>
        <quantity>1</quantity>
        <USPrice>148.95</USPrice>
        <comment>Confirm this is electric</comment>
     </item>
     <item partNum="926-AA">
        <productName>Baby Monitor</productName>
        <quantity>1</quantity>
        <USPrice>39.98</USPrice>
        <shipDate>1999-05-21</shipDate>
     </item>
  </items>
</purchaseOrder>
```

The root element, `purchaseOrder`, contains the child elements `shipTo`, `billTo`, `comment`, and `items`. All of these child elements except `comment` contain other child elements. The leaves of the tree are the child elements like `name`, `street`, `city`, and `state`, which do not contain any further child elements. Elements that contain other child elements or can accept attributes are referred to as *complex types*. Elements that contain only `PCDATA` and no child elements are referred to as *simple types*.

The complex types and some of the simple types in `po.xml` are defined in the purchase order schema below. Again, this example schema, `po.xsd`, is derived from the W3C's *Schema Part 0: Primer* (`http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/`).

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
<xsd:element name="comment" type="xsd:string"/>
<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="items" type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>

<xsd:complexType name="USAddress">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="state" type="xsd:string"/>
    <xsd:element name="zip" type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN"
       fixed="US"/>
</xsd:complexType>

<xsd:complexType name="Items">
  <xsd:sequence>
    <xsd:element name="item" minOccurs="1"
                 maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="productName"
                       type="xsd:string"/>
          <xsd:element name="quantity">
            <xsd:simpleType>
              <xsd:restriction base="xsd:positiveInteger">
                <xsd:maxExclusive value="100"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="USPrice" type="xsd:decimal"/>
          <xsd:element ref="comment" minOccurs="0"/>
```

```
              <xsd:element name="shipDate" type="xsd:date"
                           minOccurs="0"/>
            </xsd:sequence>
            <xsd:attribute name="partNum" type="SKU"
                           use="required"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>

    <!-- Stock Keeping Unit, a code for identifying products -->
    <xsd:simpleType name="SKU">
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="\d{3}-[A-Z]{2}"/>
      </xsd:restriction>
    </xsd:simpleType>

    </xsd:schema>
```

In this example, the schema comprises, similar to a DTD, a main or root `schema` element and several child elements, `element`, `complexType`, and `simpleType`. Unlike a DTD, this schema also specifies as attributes data types like `decimal`, `date`, `fixed`, and `string`. The schema also specifies constraints like `pattern value`, `minOccurs`, and `positiveInteger`, among others. In DTDs, you can only specify data types for textual data (`PCDATA` and `CDATA`); XML schema supports more complex textual and numeric data types and constraints, all of which have direct analogs in the Java language.

Note that every element in this schema has the prefix `xsd:`, which is associated with the W3C XML Schema namespace. To this end, the namespace declaration, `xmlns:xsd="http://www.w3.org/2001/XMLSchema"`, is declared as an attribute to the `schema` element.

Namespace support is another important feature of XML schemas because it provides a means to differentiate between elements written against different schemas or used for varying purposes, but which may happen to have the same name as other elements in a document. For example, suppose you declared two namespaces in your schema, one for `foo` and another for `bar`. Two XML documents are combined, one from a billing database and another from an shipping database, each of which was written against a different schema. By specifying namespaces in your schema, you can differentiate between, say, `foo:address` and `bar:address`.

Representing XML Content

This section describes how JAXB represents XML content as Java objects. Specifically, the topics in this section are as follows:

- Binding XML Names to Java Identifiers
- Java Representation of XML Schema

Binding XML Names to Java Identifiers

XML schema languages use *XML names*—strings that match the *Name* production defined in *XML 1.0 (Second Edition)* (`http://www.w3.org/XML/`) to label schema components. This set of strings is much larger than the set of valid Java class, method, and constant identifiers. To resolve this discrepancy, JAXB uses several name-mapping algorithms.

The JAXB name-mapping algorithm maps XML names to Java identifiers in a way that adheres to standard Java API design guidelines, generates identifiers that retain obvious connections to the corresponding schema, and is unlikely to result in many collisions.

Refer to Chapter 10 for information about changing default XML name mappings. See Appendix C in the *JAXB Specification* for complete details about the JAXB naming algorithm.

Java Representation of XML Schema

JAXB supports the grouping of generated classes and interfaces in Java packages. A package comprises:

- A name, which is either derived directly from the XML namespace URI, or specified by a binding customization of the XML namespace URI
- A set of Java content interfaces representing the content models declared within the schema
- A Set of Java element interfaces representing element declarations occurring within the schema
- An `ObjectFactory` class containing:

    - An instance factory method for each Java content interface and Java element interface within the package; for example, given a Java content interface named `Foo`, the derived factory method would be:

      ```
      public Foo createFoo() throws JAXBException;
      ```

    - Dynamic instance factory allocator; creates an instance of the specified Java content interface; for example:

      ```
      public Object newInstance(Class javaContentInterface)
         throws JAXBException;
      ```

    - `getProperty` and `setProperty` APIs that allow the manipulation of provider-specified properties
- Set of typesafe enum classes
- Package javadoc

Binding XML Schemas

This section describes the default XML-to-Java bindings used by JAXB. All of these bindings can be overridden on global or case-by-case levels by means of a custom binding declaration. The topics in this section are as follows:

- Simple Type Definitions
- Default Data Type Bindings
- Default Binding Rules Summary

See the *JAXB Specification* for complete information about the default JAXB bindings.

Simple Type Definitions

A schema component using a simple type definition typically binds to a Java property. Since there are different kinds of such schema components, the following Java property attributes (common to the schema components) include:

- Base type
- Collection type, if any
- Predicate

The rest of the Java property attributes are specified in the schema component using the `simple` type definition.

Default Data Type Bindings

The Java language provides a richer set of data type than XML schema. Table 9–3 lists the mapping of XML data types to Java data types in JAXB.

Table 9–3 JAXB Mapping of XML Schema Built-in Data Types

XML Schema Type	Java Data Type
`xsd:string`	`java.lang.String`
`xsd:integer`	`java.math.BigInteger`
`xsd:int`	`int`
`xsd.long`	`long`
`xsd:short`	`short`
`xsd:decimal`	`java.math.BigDecimal`
`xsd:float`	`float`
`xsd:double`	`double`
`xsd:boolean`	`boolean`
`xsd:byte`	`byte`
`xsd:QName`	`javax.xml.namespace.QName`
`xsd:dateTime`	`java.util.Calendar`
`xsd:base64Binary`	`byte[]`
`xsd:hexBinary`	`byte[]`
`xsd:unsignedInt`	`long`
`xsd:unsignedShort`	`int`
`xsd:unsignedByte`	`short`
`xsd:time`	`java.util.Calendar`

Table 9–3 JAXB Mapping of XML Schema Built-in Data Types (Continued)

XML Schema Type	**Java Data Type**
`xsd:date`	`java.util.Calendar`
`xsd:anySimpleType`	`java.lang.String`

Default Binding Rules Summary

The JAXB binding model follows the default binding rules summarized below:

- Bind the following to Java package:
  - XML Namespace URI
- Bind the following XML Schema components to Java content interface:
  - Named complex type
  - Anonymous inlined type definition of an element declaration
- Bind to typesafe enum class:
  - A named simple type definition with a basetype that derives from "`xsd:NCName`" and has enumeration facets.
- Bind the following XML Schema components to a Java Element interface:
  - A global element declaration to a Element interface.
  - Local element declaration that can be inserted into a general content list.
- Bind to Java property:
  - Attribute use
  - Particle with a term that is an element reference or local element declaration.
- Bind model group with a repeating occurrence and complex type definitions with mixed `{content type}` to:
  - A general content property; a List content-property that holds Java instances representing element information items and character data items.

Customizing JAXB Bindings

The default JAXB bindings can be overridden at a global scope or on a case-by-case basis as needed by using custom binding declarations. As described previously, JAXB uses default binding rules that can be customized by means of binding declarations made in either of two ways:

- As inline annotations in a source XML schema
- As declarations in an external binding customizations file that is passed to the JAXB binding compiler

Custom JAXB binding declarations also allow you to customize your generated JAXB classes beyond the XML-specific constraints in an XML schema to include Java-specific refinements such as class and package name mappings.

You do not need to provide a binding instruction for every declaration in your schema to generate Java classes. For example, the binding compiler uses a general name-mapping algorithm to bind XML names to names that are acceptable in the Java programming language. However, if you want to use a different naming scheme for your classes, you can specify custom binding declarations to make the binding compiler generate different names. There are many other customizations you can make with the binding declaration, including:

- Name the package, derived classes, and methods
- Assign types to the methods within the derived classes
- Choose which elements to bind to classes
- Decide how to bind each attribute and element declaration to a property in the appropriate content class
- Choose the type of each attribute-value or content specification

---

Note: Relying on the default JAXB binding behavior rather than requiring a binding declaration for each XML Schema component bound to a Java representation makes it easier to keep pace with changes in the source schema. In most cases, the default rules are robust enough that a usable binding can be produced with no custom binding declaration at all.

Code examples showing how to customize JAXB bindings are provided in Chapter 10.

Scope

When a customization value is defined in a binding declaration, it is associated with a *scope*. A scope of a customization value is the set of schema elements to which it applies. If a customization value applies to a schema element, then the schema element is said to be covered by the scope of the customization value.

Table 9–4 lists the four scopes for custom bindings.

Table 9–4 Custom Binding Scopes

Scope	**Description**
Global	A customization value defined in `<globalBindings>` has global scope. A global scope covers all the schema elements in the source schema and (recursively) any schemas that are included or imported by the source schema.
Schema	A customization value defined in `<schemaBindings>` has schema scope. A schema scope covers all the schema elements in the target name space of a schema.
Definition	A customization value in binding declarations of a type definition and global declaration has definition scope. A definition scope covers all schema elements that reference the type definition or the global declaration.
Component	A customization value in a binding declaration has component scope if the customization value applies only to the schema element that was annotated with the binding declaration.

Scope Inheritance

The different scopes form a taxonomy. The taxonomy defines both the inheritance and overriding semantics of customization values. A customization value defined in one scope is inherited for use in a binding declaration covered by another scope as shown by the following inheritance hierarchy:

- A schema element in schema scope inherits a customization value defined in global scope.
- A schema element in definition scope inherits a customization value defined in schema or global scope.
- A schema element in component scope inherits a customization value defined in definition, schema or global scope.

Similarly, a customization value defined in one scope can override a customization value inherited from another scope as shown below:

- Value in schema scope overrides a value inherited from global scope.
- Value in definition scope overrides a value inherited from schema scope or global scope.
- Value in component scope overrides a value inherited from definition, schema or global scope.

What is Not Supported

See Section E.2, “Not Required XML Schema Concepts,” in the *JAXB Specification* for the latest information about unsupported or non-required schema concepts.

JAXB APIs and Tools

The JAXB APIs and tools are shipped in the `jaxb-1.0` subdirectory of the Java WSDP. This directory contains a set of sample applications, javadoc API documentation, a JAXB binding compiler (`xjc`), implementations of the runtime binding framework APIs contained in the `javax.xml.bind` package. For instructions on using the JAXB, see Chapter 10.

10

Using JAXB

Scott Fordin

THIS chapter provides hands-on instructions for using the Java Architecture for XML Binding (JAXB). Specifically, this chapter provides nine sample Java applications, each of which demonstrates and builds upon key JAXB features and concepts. It is recommended that you follow these procedures in the order presented.

After reading this chapter, you should feel comfortable enough with JAXB that you can:

- Generate JAXB Java classes from an XML schema
- Use schema-derived JAXB classes to unmarshal and marshal XML content in a Java application
- Create a Java content tree from scratch using schema-derived JAXB classes
- Validate XML content during unmarshalling and at runtime
- Customize JAXB schema-to-Java bindings

The primary goals of the basic sample applications are to highlight the core set of JAXB functions using default settings and bindings. After familiarizing yourself with these core features and functions, you may wish to continue with Customizing JAXB Bindings (page 422) for instructions on using four additional sample applications that demonstrate how to modify the default JAXB bindings.

Note: The Purchase Order schema, `po.xsd`, and the Purchase Order XML file, `po.xml`, used in these samples are derived from the W3C XML Schema Part 0: Primer (`http://www.w3.org/TR/xmlschema-0/`), edited by David C. Fallside.

General Usage Instructions

This section provides general usage instructions for the sample applications used in this chapter, including how to build and run the applications both manually and using the Ant build tool, and provides details about the default schema-to-JAXB bindings used in these examples. Specifically, the topics in this section are as follows:

- Description
- System Requirements
- Using the Sample Applications
- Configuring and Running the Samples Manually
- Configuring and Running the Samples With Ant
- About the Schema-to-Java Bindings
- Schema-Derived JAXB Classes

Description

There are nine sample applications in this chapter; the first five demonstrate basic JAXB concepts like ummarshalling, marshalling, and validating XML content, while the last four demonstrate various ways you can customize the binding of XML schemas to Java objects. Each of the sample applications in this chapter is based on a *Purchase Order* scenario. With the exception of Sample Application 9, each uses an XML document, `po.xml`, written against an XML schema, `po.xsd`.

Table 10–1 Sample JAXB Application Descriptions

Sample Application Name	Description
Sample Application 1	Demonstrates how to unmarshal an XML document into a Java content tree and access the data contained within it.
Sample Application 2	Demonstrates how to modify a Java content tree.
Sample Application 3	Demonstrates how to use the *`ObjectFactory`* class to create a Java content tree from scratch and then marshal it to XML data.
Sample Application 4	Demonstrates how to enable validation during unmarshalling.
Sample Application 5	Demonstrates how to validate a Java content tree at runtime.
Sample Application 6	Demonstrates how to customize the default JAXB bindings by means of inline annotations in an XML schema.
Sample Application 7	Similar to Sample Application 6, this sample illustrates alternate, more terse bindings of XML `simpleType` definitions to Java datatypes.
Sample Application 8	Illustrates how to use an external binding declarations file to pass binding customizations for a read-only schema to the JAXB binding compiler.
Sample Application 9	Illustrates how to use customizations to resolve name conflicts reported by the JAXB binding compiler. Additionally, this sample illustrates how to bind a choice model group to a Java interface, and how to manipulate a JAXB `List` property. It is recommended that you first run `ant fail` in the application directory to see the errors reported by the JAXB binding compiler, and then look at `binding.xjb` to see how the errors were resolved. Running `ant` alone uses the binding customizations to resolve the name conflicts while compiling the schema.

Note: These sample applications are all located in the `$JAXB_HOME/examples/users-guide` directory.

Each sample application directory contains several base files:

- `po.xsd` is the XML schema you will use as input to the JAXB binding compiler, and from which schema-derived JAXB Java classes will be generated. For Sample Applications 6 and 7, this file contains inline binding customizations. Note that Sample Application 9 uses `example.xsd` rather than `po.xsd`.
- `po.xml` is the *Purchase Order* XML file containing sample XML content, and is the file you will unmarshal into a Java content tree in each example. This file is almost exactly the same in each sample application, with minor content differences to highlight different JAXB concepts. Note that Sample Application 9 uses `example.xml` rather than `po.xml`.
- `Main.java` is the main Java class for each sample application.
- `build.xml` is an Ant project file provided for your convenience. As shown later in this chapter, you can generate and compile schema-derived JAXB classes manually using standard Java and JAXB commands, or you can use Ant to generate, compile, and run the classes automatically. The `build.xml` file varies across the sample applications.
- `MyDatatypeConverter.java` in Sample Application 6 is a Java class used to provide custom datatype conversions.
- `binding.xjb` in Sample Applications 8 and 9 is an external binding declarations file that is passed to the JAXB binding compiler to customize the default JAXB bindings.
- `example.xsd` in Sample Application 9 is a short schema file containing deliberate naming conflicts, with the purpose of illustrating how to resolve such conflicts with custom JAXB bindings.

System Requirements

The use the JAXB sample applications described here, you need Java SDK, Standard Edition 1.3.1 or later software. Instructions are provided for using the applications under the Solaris/Linux and Windows NT/2000/XP operating environments. Instructions are provided for running the applications manually or automatically using `Ant`, which is shipped with the JWSDP (see Building the Examples, page xiv).

Using the Sample Applications

As with all applications that implement schema-derived JAXB classes, as described above, there are two distinct phases in using JAXB:

1. Generating and compiling JAXB Java classes from an XML source schema
2. Unmarshalling, validating, processing, and marshalling XML content

In the case of these sample applications, you have a choice of performing these steps "by hand," or by using `Ant` with the `build.xml` project file included in each sample application directory.

Note: It is recommended that you familiarize yourself with the manual process for at least Sample Application 1. The manual process is similar for each of the sample applications.

Configuring and Running the Samples Manually

This section describes how to configure and run Sample Application 1. The instructions for the other sample applications are essentially the same; just change the `SampleApp1` directory to the directory for the application you want to use.

Solaris/Linux

1. Set environment variables:

   ```
   export JAVA_HOME=<your J2SE installation directory>
   export JWSDP_HOME=<your JWSDP1.1 installation directory>
   export JAXB_HOME=$JWSDP_HOME/jaxb-1.0
   export JAXB_LIBS=$JAXB_HOME/lib
   export JAXP_LIBS=$JWSDP_HOME/jaxp-1.2.2/lib
   export JWSDP_LIBS=$JWSDP_HOME/jwsdp-shared/lib
   ```

2. Set your `PATH`:

```
export PATH=$JAXB_HOME/bin:$JWSDP_HOME/jwsdp-shared/bin:$PATH
```

3. Update your CLASSPATH:

   ```
   export CLASSPATH=$JAXB_LIBS/jaxb-api.jar: \
   $JAXB_LIBS/jaxb-ri.jar: \
   $JAXB_LIBS/jaxb-xjc.jar: \
   $JAXB_LIBS/jaxb-libs.jar: \
   $JAXP_LIBS/jaxp-api.jar: \
   $JAXP_LIBS/endorsed/xercesImpl.jar:                         \
   $JAXP_LIBS/endorsed/xalan.jar: \
   $JAXP_LIBS/endorsed/sax.jar: \
   $JAXP_LIBS/endorsed/dom.jar: \
   $JWSDP_LIBS/jax-qname.jar: \
   $JWSDP_LIBS/namespace.jar:.
   ```

4. Change to the desired sample application directory.

   For example, to run Sample Application 1:

   ```
   cd $JAXB_HOME/examples/users-guide/SampleApp1
   ```

5. Use the xjc.sh command to generate JAXB Java classes from the source XML schema.

   ```
   $JAXB_HOME/bin/xjc.sh po.xsd -p primer.po
   ```

   po.xsd is the name of the source XML schema. The -p primer.po switch tells the JAXB compiler to put the generated classes in a Java package named primer.po. For the purposes of this example, the package name must be primer.po. See JAXB Compiler Options (page 396) for a complete list of JAXB binding compiler options.

6. Generate API documentation for the application using the Javadoc tool (optional).

```
$JAVA_HOME/bin/javadoc -package primer.po -sourcepath .
-d docs/api -windowtitle "Generated Interfaces for po.xsd"
```

7. Compile the generated JAXB Java classes.

```
$JAVA_HOME/bin/javac Main.java primer/po/*.java primer/
po/impl/*.java
```

8. Run the Main class.

   ```
   $JAVA_HOME/bin/java Main
   ```

   The po.xml file is unmarshalled into a Java content tree, and the XML data in the content tree is written to System.out.

Windows NT/2000/XP

1. Set environment variables:

   ```
   set JAVA_HOME=<your J2SE installation directory>
   set JWSDP_HOME=<your JWSDP1.1 installation directory>
   set JAXB_HOME=%JWSDP_HOME%\jaxb-1.0
   set JAXB_LIBS=%JAXB_HOME%\lib
   set JAXP_LIBS=%JWSDP_HOME%\jaxp-1.2.2\lib
   set JWSDP_LIBS=%JWSDP_HOME%\jwsdp-shared\lib
   ```

2. Set your `PATH`:

```
set PATH=%JAXB_HOME%\bin;%JWSDP_HOME%\jwsdp-shared\bin;%PATH%
```

3. Update your `CLASSPATH`:

   ```
   set CLASSPATH=%JAXB_LIBS%\jaxb-api.jar;
   %JAXB_LIBS%\jaxb-ri.jar;
   %JAXB_LIBS%\jaxb-xjc.jar;
   %JAXB_LIBS%\jaxb-libs.jar;
   %JAXP_LIBS%\jaxb-api.jar;
   %JAXP_LIBS%\endorsed\xercesImpl.jar;
   %JAXP_LIBS%\endorsed\xalan.jar;
   %JAXP_LIBS%\endorsed\sax.jar;
   %JAXP_LIBS%\endorsed\dom.jar;
   %JWSDP_LIBS%\jax-qname.jar;
   %JWSDP_LIBS%\namespace.jar;.
   ```

   The line breaks shown above are for legibility only; be sure to enter your `CLASSPATH` on a single line.

4. Change to the desired sample application directory.

   For example, to run Sample Application 1:

   ```
   cd %JAXB_HOME%\examples\users-guide\SampleApp1
   ```

5. Use the `xjc.bat` command to generate JAXB Java classes from the source XML schema.

   ```
   %JAXB_HOME%\bin\xjc.bat po.xsd -p primer.po
   ```

   `po.xsd` is the name of the source XML schema. The `-p primer.po` switch tells the JAXB compiler to put the generated classes in a Java package named `primer.po`. For the purposes of this example, the package name must be `primer.po`. See JAXB Compiler Options (page 396) for a complete list of JAXB binding compiler options.

6. Generate API documentation for the application using the Javadoc tool (optional).

   ```
   %JAVA_HOME%\bin\javadoc -package primer.po -sourcepath .
   -d docs\api -windowtitle "Generated Interfaces for po.xsd"
   ```

7. Compile the schema-derived JAXB Java classes.

   ```
   %JAVA_HOME%\bin\javac Main.java primer\po\*.java
    primer\po\impl\*.java
   ```

8. Run the `Main` class.

   ```
   %JAVA_HOME%\bin\java Main
   ```

   The `po.xml` file is unmarshalled into a Java content tree, and the XML data in the content tree is written to `System.out`.

The schema-derived JAXB classes and how they are bound to the source schema is described in About the Schema-to-Java Bindings (page 398). The methods used for building and processing the Java content tree in each of the basic applications are analyzed in Basic Sample Applications (page 409).

Configuring and Running the Samples With Ant

The `build.xml` file included in each sample application directory is an Ant project file. The Apache Ant build tool is included with the Java Web Services Developer Pack, and you can use this project to automatically perform all the steps listed in Configuring and Running the Samples Manually (page 391). Specifically, using Ant with the included `build.xml` project files does the following:

1. Updates your `CLASSPATH` to include the necessary schema-derived JAXB classes.
2. Runs the JAXB binding compiler to generate JAXB Java classes from the XML source schema, `po.xsd`, and puts the classes in a package named `primer.po`.
3. Generates API documentation from the schema-derived JAXB classes using the Javadoc tool.
4. Compiles the schema-derived JAXB classes.
5. Runs the `Main` class for the sample application.

As mentioned previously, it is recommended that you familiarize yourself with the manual steps for performing these tasks for at least the first sample application.

Solaris/Linux

1. Set environment variables:

   ```
   export JAVA_HOME=<your J2SE installation directory>
   export JWSDP_HOME=<your JWSDP1.1 installation directory>
   export JAXB_HOME=$JWSDP_HOME/jaxb-1.0
   export ANT_HOME=$JWSDP_HOME/jakarta-ant-1.5.1
   ```

2. Set your PATH:

```
export PATH=$JAXB_HOME/bin:$JWSDP_HOME/jwsdp-shared/bin:$PATH
```

3. Change to the desired sample application directory.

   For example, to run Sample Application 1:

   ```
   cd $JAXB_HOME/examples/users-guide/SampleApp1
   ```

4. Run Ant:

   ```
   $ANT_HOME/bin/ant -emacs
   ```

5. Repeat these steps for each sample application.

Windows NT/2000/XP

1. Set environment variables:

   ```
   set JAVA_HOME=<your J2SE installation directory>
   set JWSDP_HOME=<your JWSDP1.1 installation directory>
   set JAXB_HOME=%JWSDP_HOME%\jaxb-1.0
   set ANT_HOME=%JWSDP_HOME%\jakarta-ant-1.5.1
   ```

2. Set your PATH:

```
set PATH=%JAXB_HOME%\bin;%JWSDP_HOME%\jwsdp-shared\bin;%PATH%
```

3. Change to the desired sample application directory.

   For example, to run Sample Application 1:

   ```
   cd %JAXB_HOME%\examples\users-guide\SampleApp1
   ```

4. Run Ant:

   ```
   %ANT_HOME%\bin\ant -emacs
   ```

5. Repeat these steps for each sample application.

The schema-derived JAXB classes and how they are bound to the source schema is described in About the Schema-to-Java Bindings (page 398). The methods used for building and processing the Java content tree are described in Basic Sample Applications (page 409).

JAXB Compiler Options

The JAXB schema binding compiler is located in the `<JWSDP_HOME>/jaxb-1.0/bin` directory. There are two scripts in this directory: `xjc.sh` (Solaris/Linux) and `xjc.bat` (Windows).

Both `xjc.sh` and `xjc.bat` take the same command-line options. You can display quick usage instructions by invoking the scripts without any options, or with the `-help` switch. The syntax is as follows:

```
xjc [-options ...] <schema>
```

The `xjc` command-line options are listed in Table 10–2.

Table 10–2 `xjc` Command-Line Options

Option or Argument	**Description**
`<schema>`	One or more schema files to compile.
`-nv`	Do not perform strict validation of the input schema(s). By default, `xjc` performs strict validation of the source schema before processing. Note that this does not mean the binding compiler will not perform any validation; it simply means that it will perform less-strict validation.
`-extension`	By default, `xjc` strictly enforces the rules outlined in the Compatibility chapter of the *JAXB Specification*. Specifically, Appendix E.2 defines a set of W3C XML Schema features that are not completely supported by JAXB v1.0. In some cases, you may be able to use these extensions with the `-extension` switch. In the default (strict) mode, you are also limited to using only the binding customizations defined in the specification. By using the `-extension` switch, you can enable the JAXB Vendor Extensions.

Table 10–2 `xjc` Command-Line Options (Continued)

Option or Argument	Description

`-b` *`<file>`*	Specify one or more external binding files to process (each binding file must have it's own `-b` switch). The syntax of the external binding files is extremely flexible. You may have a single binding file that contains customizations for multiple schemas, or you can break the customizations into multiple bindings files; for example:
	
	`xjc schema1.xsd schema2.xsd schema3.xsd -b bindings123.xjb`
	`xjc schema1.xsd schema2.xsd schema3.xsd -b bindings1.xjb -b bindings2.xjb -b bindings3.xjb`
	
	Note that the ordering of schema files and binding files on the command line does not matter.

`-d` *`<dir>`*	By default, `xjc` will generate Java content classes in the current directory. Use this option to specify an alternate output directory. The directory must already exist; `xjc` will not create it for you.
`-p` *`<pkg>`*	Specifies the target package for schema-derived classes. This option overrides any binding customization for package name as well as the default package name algorithm defined in the *JAXB Specification*.
`-host` *`<proxyHost>`*	Set `http.proxyHost` to *`<proxyHost>`*.
`-port` *`<proxyPort>`*	Set `http.proxyPort` to *`<proxyPort>`*.
`-classpath` *`<arg>`*	Specify where to find client application class files used by the `<jxb:javaType>` and `<xjc:superClass>` customizations.
`-readOnly`	Generated source files will be marked read-only. By default, `xjc` does not write-protect the schema-derived source files it generates.
`-help`	Display this help message.

The command invoked by the `xjc.sh` and `xjc.bat` scripts is equivalent to the Java command:

```
$JAVA_HOME/bin/java -jar $JAXB_HOME/lib/jaxb-xjc.jar
```

About the Schema-to-Java Bindings

When you run the JAXB binding compiler against the `po.xsd` XML schema used in the first five sample applications, the JAXB binding compiler generates a Java package named `primer.po` containing eleven classes, making a total of twelve classes in each of the first five sample applications:

Table 10–3 Schema-Derived JAXB Classes in Sample Applications 1 Through 5

Class	Description
`primer/po/Comment.java`	Public interface extending `javax.xml.bind.Element`; binds to the global schema `element` named `comment`. Note that JAXB generates element interfaces for all global element declarations.
`primer/po/Items.java`	Public interface that binds to the schema `complexType` named `Items`.
`primer/po/ObjectFactory.java`	Public class extending `com.sun.xml.bind.DefaultJAXB-ContextImpl`; used to create instances of specified interfaces. For example, the `ObjectFactory` `createComment()` method instantiates a `Comment` object.
`primer/po/PurchaseOrder.java`	Public interface extending `javax.xml.bind.Element`, and `PurchaseOrderType`; binds to the global schema `element` named `PurchaseOrder`.
`primer/po/PurchaseOrderType.java`	Public interface that binds to the schema `complexType` named `PurchaseOrderType`.
`primer/po/USAddress.java`	Public interface that binds to the schema `complexType` named `USAddress`.
`primer/po/impl/CommentImpl.java`	Implementation of `Comment.java`.
`primer/po/impl/ItemsImpl.java`	Implementation of `Items.java`
`primer/po/impl/PurchaseOrderImpl.java`	Implementation of `PurchaseOrder.java`
`primer/po/impl/PurchaseOrderType-Impl.java`	Implementation of `PurchaseOrderType.java`

Table 10–3 Schema-Derived JAXB Classes in Sample Applications 1 Through 5 (Continued)

Class	**Description**
`primer/po/impl/ USAddressImpl.java`	Implementation of `USAddress.java`

Note: You should never directly use the generated implementation classes—that is, `*Impl.java` in the *<packagename>*`/impl` directory. These classes are not directly referenceable because the class names in this directory are not standardized by the JAXB specification. The `ObjectFactory` method is the only portable means to create an instance of a schema-derived interface. There is also an `ObjectFactory.newInstance(Class JAXBinterface)` method that enables you to create instances of interfaces.

These classes and their specific bindings to the source XML schema for Sample Applications 1 through 5 are described below.

Table 10–4 Schema-to-Java Bindings for Sample Applications 1 Through 5

XML Schema	**JAXB Binding**
`<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">`	
`<xsd:element name="purchaseOrder" type="PurchaseOrderType"/>`	`PurchaseOrder.java`
`<xsd:element name="comment" type="xsd:string"/>`	`Comment.java`
`<xsd:complexType name="PurchaseOrderType">` `<xsd:sequence>` `<xsd:element name="shipTo" type="USAddress"/>` `<xsd:element name="billTo" type="USAddress"/>` `<xsd:element ref="comment" minOccurs="0"/>` `<xsd:element name="items" type="Items"/>` `</xsd:sequence>` `<xsd:attribute name="orderDate" type="xsd:date"/>` `</xsd:complexType>`	`PurchaseOrderType.java`

Table 10–4 Schema-to-Java Bindings for Sample Applications 1 Through 5

XML Schema	JAXB Binding

`<xsd:complexType name="USAddress">`
  `<xsd:sequence>`
    `<xsd:element name="name" type="xsd:string"/>`
    `<xsd:element name="street" type="xsd:string"/>`
    `<xsd:element name="city" type="xsd:string"/>`
    `<xsd:element name="state" type="xsd:string"/>`
    `<xsd:element name="zip" type="xsd:decimal"/>`
  `</xsd:sequence>`
`<xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>`
`</xsd:complexType>`
	`USAd-`
	`dress.java`

`<xsd:complexType name="Items">`
  `<xsd:sequence>`
    `<xsd:element name="item" minOccurs="1" maxOc-`
`curs="unbounded">`
	`Items.java`

`<xsd:complexType>`
        `<xsd:sequence>`
         `<xsd:element name="productName" type="xsd:string"/>`
         `<xsd:element name="quantity">`
           `<xsd:simpleType>`
             `<xsd:restriction base="xsd:positiveInteger">`
               `<xsd:maxExclusive value="100"/>`
             `</xsd:restriction>`
           `</xsd:simpleType>`
         `</xsd:element>`
         `<xsd:element name="USPrice" type="xsd:decimal"/>`
         `<xsd:element ref="comment" minOccurs="0"/>`
`<xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>`
        `</xsd:sequence>`
   `<xsd:attribute name="partNum" type="SKU" use="required"/>`
      `</xsd:complexType>`
	`Items.Item-`
	`Type`

`</xsd:element>`
  `</xsd:sequence>`
`</xsd:complexType>`	

`<!-- Stock Keeping Unit, a code for identifying products -->`	

Table 10–4 Schema-to-Java Bindings for Sample Applications 1 Through 5

XML Schema	JAXB Binding

`<xsd:simpleType name="SKU">`
`  <xsd:restriction base="xsd:string">`
`    <xsd:pattern value="\d{3}-[A-Z]{2}"/>`
`  </xsd:restriction>`
`</xsd:simpleType>`	

`</xsd:schema>`	

Schema-Derived JAXB Classes

The code for the individual classes generated by the JAXB binding compiler for Sample Applications 1 through 5 is listed below, followed by brief explanations of its functions. The classes listed here are:

- `Comment.java`
- `Items.java`
- `ObjectFactory.java`
- `PurchaseOrder.java`
- `PurchaseOrderType.java`
- `USAddress.java`

Comment.java

In `Comment.java`:

- The `Comment.java` class is part of the `primer.po` package.
- `Comment` is a public interface that extends `javax.xml.bind.Element`.
- Content in instantiations of this class bind to the XML schema element named `comment`.
- The `getValue()` and `setValue()` methods are used to get and set strings representing XML `comment` elements in the Java content tree.

The `Comment.java` code looks like this:

```
package primer.po;

public interface Comment
    extends javax.xml.bind.Element
{

    String getValue();
    void setValue(String value);
}
```

Items.java

In `Items.java`, below:

- The `Items.java` class is part of the `primer.po` package.
- The class provides public interfaces for `Items` and `ItemType`.
- Content in instantiations of this class bind to the XML ComplexTypes `Items` and its child element `ItemType`.
- `Item` provides the `getItem()` method.
- `ItemType` provides methods for:
  - `getPartNum();`
  - `setPartNum(String value);`
  - `getComment();`
  - `setComment(java.lang.String value);`
  - `getUSPrice();`
  - `setUSPrice(java.math.BigDecimal value);`
  - `getProductName();`
  - `setProductName(String value);`
  - `getShipDate();`
  - `setShipDate(java.util.Calendar value);`
  - `getQuantity();`
  - `setQuantity(java.math.BigInteger value);`

The `Items.java` code looks like this:

```
package primer.po;

public interface Items {
   java.util.List getItem();

   public interface ItemType {
        String getPartNum();
        void setPartNum(String value);
        java.lang.String getComment();
        void setComment(java.lang.String value);
        java.math.BigDecimal getUSPrice();
        void setUSPrice(java.math.BigDecimal value);
        String getProductName();
        void setProductName(String value);
        java.util.Calendar getShipDate();
        void setShipDate(java.util.Calendar value);
        java.math.BigInteger getQuantity();
        void setQuantity(java.math.BigInteger value);
    }
}
```

ObjectFactory.java

In `ObjectFactory.java`, below:

- The `ObjectFactory` class is part of the `primer.po` package.
- `ObjectFactory` provides factory methods for instantiating Java interfaces representing XML content in the Java content tree.
- Method names are generated by concatenating:
  - The string constant `create`
  - If the Java content interface is nested within another interface, then the concatenation of all outer Java class names
  - The name of the Java content interface
  - JAXB implementation-specific code was removed in this example to make it easier to read.

For example, in this case, for the Java interface `primer.po.Items.ItemType`, `ObjectFactory` creates the method `createItemsItemType()`.

The ObjectFactory.java code looks like this:

```
package primer.po;

public class ObjectFactory
    extends com.sun.xml.bind.DefaultJAXBContextImpl {

    /**
     * Create a new ObjectFactory that can be used to create
     * new instances of schema derived classes for package:
     * primer.po
     */
    public ObjectFactory() {
        super(new primer.po.ObjectFactory.GrammarInfoImpl());
    }

    /**
     * Create an instance of the specified Java content
     * interface.
     */
    public Object newInstance(Class javaContentInterface)
        throws javax.xml.bind.JAXBException
    {
        return super.newInstance(javaContentInterface);
    }

    /**
     * Get the specified property. This method can only be
     * used to get provider specific properties.
     * Attempting to get an undefined property will result
     * in a PropertyException being thrown.
     */
    public Object getProperty(String name)
        throws javax.xml.bind.PropertyException
    {
        return super.getProperty(name);
    }

    /**
     * Set the specified property. This method can only be
     * used to set provider specific properties.
     * Attempting to set an undefined property will result
     * in a PropertyException being thrown.
     */
    public void setProperty(String name, Object value)
        throws javax.xml.bind.PropertyException
    {
        super.setProperty(name, value);
```

```
}

/**
 * Create an instance of PurchaseOrder
 */
public primer.po.PurchaseOrder createPurchaseOrder()
    throws javax.xml.bind.JAXBException
{
    return ((primer.po.PurchaseOrder)
        newInstance((primer.po.PurchaseOrder.class)));
}

/**
 * Create an instance of ItemsItemType
 */
public primer.po.Items.ItemType createItemsItemType()
    throws javax.xml.bind.JAXBException
{
    return ((primer.po.Items.ItemType)
        newInstance((primer.po.Items.ItemType.class)));
}

/**
 * Create an instance of USAddress
 */
public primer.po.USAddress createUSAddress()
    throws javax.xml.bind.JAXBException
{
    return ((primer.po.USAddress)
        newInstance((primer.po.USAddress.class)));
}

/**
 * Create an instance of Comment
 */
public primer.po.Comment createComment()
    throws javax.xml.bind.JAXBException
{
    return ((primer.po.Comment)
        newInstance((primer.po.Comment.class)));
}

/**
 * Create an instance of Comment
 */
public primer.po.Comment createComment(String value)
    throws javax.xml.bind.JAXBException
{
```

```
            return new primer.po.impl.CommentImpl(value);
        }

        /**
         * Create an instance of Items
         */
        public primer.po.Items createItems()
            throws javax.xml.bind.JAXBException
        {
            return ((primer.po.Items)
                newInstance((primer.po.Items.class)));
        }

        /**
         * Create an instance of PurchaseOrderType
         */
        public primer.po.PurchaseOrderType
    createPurchaseOrderType()
            throws javax.xml.bind.JAXBException
        {
            return ((primer.po.PurchaseOrderType)
                newInstance((primer.po.PurchaseOrderType.class)));
        }
    }
```

PurchaseOrder.java

In `PurchaseOrder.java`, below:

- The `PurchaseOrder` class is part of the `primer.po` package.
- `PurchaseOrder` is a public interface that extends `javax.xml.bind.Element` and `primer.po.PurchaseOrderType`.
- Content in instantiations of this class bind to the XML schema element named `purchaseOrder`.

The `PurchaseOrder.java` code looks like this:

```
package primer.po;

public interface PurchaseOrder
    extends javax.xml.bind.Element, primer.po.PurchaseOrderType
{
}
```

PurchaseOrderType.java

In PurchaseOrderType.java, below:

- The PurchaseOrderType class is part of the primer.po package.
- Content in instantiations of this class bind to the XML schema child element named PurchaseOrderType.
- PurchaseOrderType is a public interface that provides the following methods:
    - getItems();
    - setItems(primer.po.Items value);
    - getOrderDate();
    - setOrderDate(java.util.Calendar value);
    - getComment();
    - setComment(java.lang.String value);
    - getBillTo();
    - setBillTo(primer.po.USAddress value);
    - getShipTo();
    - setShipTo(primer.po.USAddress value);

The PurchaseOrderType.java code looks like this:

```
package primer.po;

public interface PurchaseOrderType {
    primer.po.Items getItems();
    void setItems(primer.po.Items value);
    java.util.Calendar getOrderDate();
    void setOrderDate(java.util.Calendar value);
    java.lang.String getComment();
    void setComment(java.lang.String value);
    primer.po.USAddress getBillTo();
    void setBillTo(primer.po.USAddress value);
    primer.po.USAddress getShipTo();
    void setShipTo(primer.po.USAddress value);
}
```

USAddress.java

In `USAddress.java`, below:

- The `USAddress` class is part of the `primer.po` package.
- Content in instantiations of this class bind to the XML schema element named `USAddress`.
- `USAddress` is a public interface that provides the following methods:
    - `getState();`
    - `setState(String value);`
    - `getZip();`
    - `setZip(java.math.BigDecimal value);`
    - `getCountry();`
    - `setCountry(String value);`
    - `getCity();`
    - `setCity(String value);`
    - `getStreet();`
    - `setStreet(String value);`
    - `getName();`
    - `setName(String value);`

The `USAddress.java` code looks like this:

```
package primer.po;

public interface USAddress {
    String getState();
    void setState(String value);
    java.math.BigDecimal getZip();
    void setZip(java.math.BigDecimal value);
    String getCountry();
    void setCountry(String value);
    String getCity();
    void setCity(String value);
    String getStreet();
    void setStreet(String value);
    String getName();
    void setName(String value);
}
```

Basic Sample Applications

This section describes five basic sample applications that demonstrate how to:

- Unmarshal an XML document into a Java content tree and access the data contained within it
- Modify a Java content tree
- Use the `ObjectFactory` class to create a Java content tree from scratch and then marshal it to XML data
- Perform validation during unmarshalling
- Validate a Java content tree at runtime

Sample Application 1

The purpose of Sample Application 1 is to demonstrate how to unmarshal an XML document into a Java content tree and access the data contained within it.

1. The `<JWSDP_HOME>/jaxb-1.0/examples/users-guide/SampleApp1/Main.java` class declares imports for four standard Java classes plus three JAXB binding framework classes and the `primer.po` package:

```
import java.io.FileInputStream
import java.io.IOException
import java.util.Iterator
import java.util.List
import javax.xml.bind.JAXBContext
import javax.xml.bind.JAXBException
import javax.xml.bind.Unmarshaller
import primer.po.*;
```

2. A `JAXBContext` instance is created for handling classes generated in `primer.po`.

```
JAXBContext jc = JAXBContext.newInstance( "primer.po" );
```

3. An `Unmarshaller` instance is created.

```
Unmarshaller u = jc.createUnmarshaller();
```

4. po.xml is unmarshalled into a Java content tree comprising objects generated by the JAXB binding compiler into the primer.po package.

```
PurchaseOrder po =
    (PurchaseOrder)u.unmarshal(
        new FileInputStream( "po.xml" ) );
```

5. A simple string is printed to system.out to provide a heading for the purchase order invoice.

```
System.out.println( "Ship the following items to: " );
```

6. get and display methods are used to parse XML content in preparation for output.

```
USAddress address = po.getShipTo();
    displayAddress( address );
    Items items = po.getItems();
    displayItems( items );
```

7. Basic error handling is implemented.

```
} catch( JAXBException je ) {
    je.printStackTrace();
} catch( IOException ioe ) {
    ioe.printStackTrace();
```

8. The USAddress branch of the Java tree is walked, and address information is printed to system.out.

```
public static void displayAddress( USAddress address ) {
    // display the address
    System.out.println( "\t" + address.getName() );
    System.out.println( "\t" + address.getStreet() );
    System.out.println( "\t" + address.getCity() +
        ", " + address.getState() +
        " "  + address.getZip() );
    System.out.println( "\t" + address.getCountry() +
        "\n");
}
```

9. The `Items` list branch is walked, and item information is printed to `system.out`.

```
public static void displayItems( Items items ) {
    // the items object contains a List of
    //primer.po.ItemType objects
    List itemTypeList = items.getItem();
```

10. Walking of the `Items` branch is iterated until all items have been printed.

```
for( Iterator iter = itemTypeList.iterator(); iter.hasNext(); )
{
     Items.ItemType item = (Items.ItemType)iter.next();
     System.out.println( "\t" + item.getQuantity() +
        " copies of \"" + item.getProductName() +
        "\"" );
}
```

Sample Output

Running `java Main` for this sample application produces the following output:

```
Ship the following items to:
   Alice Smith
   123 Maple Street
   Cambridge, MA 12345
   US

   5 copies of "Nosferatu - Special Edition (1929)"
   3 copies of "The Mummy (1959)"
   3 copies of "Godzilla and Mothra: Battle for Earth/Godzilla
     vs. King Ghidora"
```

Sample Application 2

The purpose of Sample Application 2 is to demonstrate how to modify a Java content tree.

1. The `<JWSDP_HOME>/jaxb-1.0/examples/users-guide/SampleApp2/Main.java` class declares imports for three standard

Java classes plus four JAXB binding framework classes and `primer.po` package:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.math.BigDecimal;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import primer.po.*;
```

2. A `JAXBContext` instance is created for handling classes generated in `primer.po`.

```
JAXBContext jc = JAXBContext.newInstance( "primer.po" );
```

3. An `Unmarshaller` instance is created, and `po.xml` is unmarshalled.

```
Unmarshaller u = jc.createUnmarshaller();
PurchaseOrder po =
    (PurchaseOrder)u.unmarshal(
        new FileInputStream( "po.xml" ) );
```

4. `set` methods are used to modify information in the `address` branch of the content tree.

```
USAddress address = po.getBillTo();
address.setName( "John Bob" );
address.setStreet( "242 Main Street" );
address.setCity( "Beverly Hills" );
address.setState( "CA" );
address.setZip( new BigDecimal( "90210" ) );
```

5. A `Marshaller` instance is created, and the updated XML content is marshalled to `system.out`. The `setProperty` API is used to specify output encoding; in this case formatted (human readable) XML format.

```
Marshaller m = jc.createMarshaller();
m.setProperty( Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE
);
m.marshal( po, System.out );
```

Sample Output

Running `java Main` for this sample application produces the following output:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<purchaseOrder orderDate="1999-10-20-05:00">
<shipTo country="US">
<name>Alice Smith</name>
<street>123 Maple Street</street>
<city>Cambridge</city>
<state>MA</state>
<zip>12345</zip>
</shipTo>
<billTo country="US">
<name>John Bob</name>
<street>242 Main Street</street>
<city>Beverly Hills</city>
<state>CA</state>
<zip>90210</zip>
</billTo>
<items>
<item partNum="242-NO">
<productName>Nosferatu - Special Edition (1929)</productName>
<quantity>5</quantity>
<USPrice>19.99</USPrice>
</item>
<item partNum="242-MU">
<productName>The Mummy (1959)</productName>
<quantity>3</quantity>
<USPrice>19.98</USPrice>
</item>
<item partNum="242-GZ">
<productName>Godzilla and Mothra: Battle for Earth/Godzilla vs.
   King Ghidora</productName>
<quantity>3</quantity>
<USPrice>27.95</USPrice>
</item>
</items>
</purchaseOrder>
```

Sample Application 3

The purpose of Sample Application 3 is to demonstrate how to use the `ObjectFactory` class to create a Java content tree from scratch and then marshal it to XML data.

1. The `<JWSDP_HOME>/jaxb-1.0/examples/users-guide/SampleApp3/Main.java` class declares imports for four standard Java classes plus three JAXB binding framework classes and the `primer.po` package:

```
import java.math.BigDecimal;
import java.math.BigInteger;
import java.util.Calendar;
import java.util.List;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import primer.po.*;
```

2. A `JAXBContext` instance is created for handling classes generated in `primer.po`.

```
JAXBContext jc = JAXBContext.newInstance( "primer.po" );
```

3. The `ObjectFactory` class is used to instantiate a new empty `PurchaseOrder` object.

```
// creating the ObjectFactory
ObjectFactory objFactory = new ObjectFactory();

// create an empty PurchaseOrder
PurchaseOrder po = objFactory.createPurchaseOrder();
```

4. Per the constraints in the `po.xsd` schema, the `PurchaseOrder` object requires a value for the `orderDate` attribute. To satisfy this constraint, the `orderDate` is set using the standard `Calendar.getInstance()` method from `java.util.Calendar`.

```
po.setOrderDate( Calendar.getInstance() );
```

5. The `ObjectFactory` is used to instantiate new empty `USAddress` objects, and the required attributes are set.

```
USAddress shipTo = createUSAddress( "Alice Smith",
                                    "123 Maple Street",
                                    "Cambridge",
                                    "MA",
                                    "12345" );
  po.setShipTo( shipTo );

USAddress billTo = createUSAddress( "Robert Smith",
                                    "8 Oak Avenue",
                                    "Cambridge",
                                    "MA",
                                    "12345" );
po.setBillTo( billTo );
```

6. The `ObjectFactory` class is used to instantiate a new empty `Items` object.

```
Items items = objFactory.createItems();
```

7. A `get` method is used to get a reference to the `ItemType` list.

```
List itemList = items.getItem();
```

8. `ItemType` objects are created and added to the `Items` list.

```
itemList.add( createItemType(
    "Nosferatu - Special Edition (1929)",
                              new BigInteger( "5" ),
                              new BigDecimal( "19.99" ),
                              null,
                              null,
                              "242-NO" ) );
itemList.add( createItemType( "The Mummy (1959)",
                              new BigInteger( "3" ),
                              new BigDecimal( "19.98" ),
                              null,
                              null,
                              "242-MU" ) );
itemList.add( createItemType(
    "Godzilla and Mothra: Battle for Earth/Godzilla vs. King
Ghidora",
                              new BigInteger( "3" ),
                                        new BigDecimal( "27.95" ),
                                        null,
                                        null,
                                        "242-GZ" ) );
```

9. The `items` object now contains a list of `ItemType` objects and can be added to the po object.

```
po.setItems( items );
```

10. A `Marshaller` instance is created, and the updated XML content is marshalled to `system.out`. The `setProperty` API is used to specify output encoding; in this case formatted (human readable) XML format.

```
Marshaller m = jc.createMarshaller();
m.setProperty( Marshaller.JAXB_FORMATTED_OUTPUT,
    Boolean.TRUE );
m.marshal( po, System.out );
```

11. An empty `USAddress` object is created and its properties set to comply with the schema constraints.

```
public static USAddress createUSAddress(
                                 ObjectFactory objFactory,
                                 String name, String street,
                                 String city,
                                 String state,
                                 String zip )
        throws JAXBException {

        // create an empty USAddress objects
        USAddress address = objFactory.createUSAddress();

        // set properties on it
        address.setName( name );
        address.setStreet( street );
        address.setCity( city );
        address.setState( state );
        address.setZip( new BigDecimal( zip ) );

        // return it
        return address;
    }
```

12.Similar to the previous step, an empty `ItemType` object is created and its properties set to comply with the schema constraints.

```
public static Items.ItemType createItemType( ObjectFactory
objFactory,
                                        String productName,
                                        BigInteger quantity,
                                        BigDecimal price,
                                        String comment,
                                        Calendar shipDate,
                                        String partNum )
        throws JAXBException {

        // create an empty ItemType object
        Items.ItemType itemType =
            objFactory.createItemsItemType();

        // set properties on it
        itemType.setProductName( productName );
        itemType.setQuantity( quantity );
        itemType.setUSPrice( price );
        itemType.setComment( comment );
        itemType.setShipDate( shipDate );
        itemType.setPartNum( partNum );

        // return it
        return itemType;
    }
```

Sample Output

Running `java Main` for this sample application produces the following output:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<purchaseOrder orderDate="2002-09-24-05:00">
<shipTo>
<name>Alice Smith</name>
<street>123 Maple Street</street>
<city>Cambridge</city>
<state>MA</state>
<zip>12345</zip>
</shipTo>
<billTo>
<name>Robert Smith</name>
<street>8 Oak Avenue</street>
<city>Cambridge</city>
<state>MA</state>
```

```
<zip>12345</zip>
</billTo>
<items>
<item partNum="242-NO">
<productName>Nosferatu - Special Edition (1929)</productName>
<quantity>5</quantity
<USPrice>19.99</USPrice>
</item>
<item partNum="242-MU">
<productName>The Mummy (1959)</productName>
<quantity>3</quantity>
<USPrice>19.98</USPrice>
</item>
<item partNum="242-GZ">
<productName>Godzilla and Mothra: Battle for Earth/Godzilla vs.
King Ghidora</productName>
<quantity>3</quantity>
<USPrice>27.95</USPrice>
</item>
</items>
</purchaseOrder>
```

Sample Application 4

The purpose of Sample Application 4 is to demonstrate how to enable validation during unmarshalling (*Unmarshal-Time Validation*). Note that JAXB provides functions for validation during unmarshalling but not during marshalling. Validation is explained in more detail in More About Validation (page 374).

1. The `<JWSDP_HOME>/jaxb-1.0/examples/users-guide/SampleApp4/Main.java` class declares imports for three standard Java classes plus seven JAXB binding framework classes and the `primer.po` package:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.math.BigDecimal;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.UnmarshalException;
import javax.xml.bind.Unmarshaller;
import javax.xml.bind.ValidationEvent;
import javax.xml.bind.util.ValidationEventCollector;
import primer.po.*;
```

2. A `JAXBContext` instance is created for handling classes generated in `primer.po`.

```
JAXBContext jc = JAXBContext.newInstance( "primer.po" );
```

3. An `Unmarshaller` instance is created.

```
Unmarshaller u = jc.createUnmarshaller();
```

4. The default JAXB Unmarshaller `ValidationEventHandler` is enabled to send to validation warnings and errors to `system.out`. The default configuration causes the unmarshal operation to fail upon encountering the first validation error.

```
u.setValidating( true );
```

5. An attempt is made to unmarshal `po.xml` into a Java content tree. For the purposes of this example, the `po.xml` contains a deliberate error.

```
PurchaseOrder po =
     (PurchaseOrder)u.unmarshal( new FileInputStream( "po.xml"
) );
```

6. The default validation event handler processes a validation error, generates output to `system.out`, and then an exception is thrown.

```
} catch( UnmarshalException ue ) {
System.out.println( "Caught UnmarshalException" );
   } catch( JAXBException je ) {
      je.printStackTrace();
   } catch( IOException ioe ) {
      ioe.printStackTrace();
```

Sample Output

Running `java Main` for this sample application produces the following output:

```
DefaultValidationEventHandler: [ERROR]: "-1" does not satisfy
the "positiveInteger" type
Caught UnmarshalException
```

Sample Application 5

The purpose of Sample Application 5 is to demonstrate how to validate a Java content tree at runtime (*On-Demand Validation*). At any point, client applications can call the `Validator.validate` method on the Java content tree (or any subtree of it). All JAXB Providers are required to support this operation. Validation is explained in more detail in More About Validation (page 374).

1. The `<JWSDP_HOME>/jaxb-1.0/examples/users-guide/SampleApp5/Main.java` class declares imports for five standard Java classes plus nine JAXB Java classes and the `primer.po` package:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.math.BigDecimal;
import java.math.BigInteger;
import java.util.List;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.UnmarshalException;
import javax.xml.bind.Unmarshaller;
import javax.xml.bind.ValidationEvent;
import javax.xml.bind.ValidationException;
import javax.xml.bind.Validator;
import javax.xml.bind.util.ValidationEventCollector;
import primer.po.*;
```

2. A `JAXBContext` instance is created for handling classes generated in `primer.po`.

```
JAXBContext jc = JAXBContext.newInstance( "primer.po" );
```

3. An `Unmarshaller` instance is created, and a valid `po.xml` document is unmarshalled into a Java content tree. Note that `po.xml` is valid at this point; invalid data will be added later in this example.

```
Unmarshaller u = jc.createUnmarshaller();
PurchaseOrder po =
   (PurchaseOrder)u.unmarshal( new FileInputStream( "po.xml" )
);
```

4. A reference is obtained for the first item in the purchase order.

```
Items items = po.getItems();
List itemTypeList = items.getItem();
Items.ItemType item = (Items.ItemType)itemTypeList.get( 0 );
```

5. Next, the item quantity is set to an invalid number. When validation is enabled later in this example, this invalid quantity will throw an exception.

```
item.setQuantity( new BigInteger( "-5" ) );
```

Note: If `@enableFailFastCheck` was `"true"` and the optional `FailFast` validation method was supported by an implementation, a `TypeConstraintException` would be thrown here. Note that the JAXB implementation does not support the `FailFast` feature. Refer to the *JAXB Specification* for more information about `FailFast` validation.

---

6. A `Validator` instance is created, and the content tree is validated. Note that the `Validator` class is responsible for managing On-Demand validation, whereas the `Unmarshaller` class is responsible for managing Unmarshal-Time validation during unmarshal operations.

```
Validator v = jc.createValidator();
boolean valid = v.validateRoot( po );
System.out.println( valid );
```

7. The default validation event handler processes a validation error, generates output to `system.out`, and then an exception is thrown.

```
} catch( ValidationException ue ) {
   System.out.println( "Caught ValidationException" );
} catch( JAXBException je ) {
   je.printStackTrace();
} catch( IOException ioe ) {
   ioe.printStackTrace();
}
```

Sample Output

Running `java Main` for this sample application produces the following output:

```
DefaultValidationEventHandler: [ERROR]: "-5" does not satisfy
the "positiveInteger" type
Caught ValidationException
```

Customizing JAXB Bindings

The remainder of this chapter describes several sample applications that build on the concepts demonstrated in Sample Applications 1, 2, 3, 4, and 5, above.

The goal of this section is to illustrate how to customize JAXB bindings by means of custom binding declarations made in either of two ways:

- As annotations made inline in an XML schema
- As statements in an external file passed to the JAXB binding compiler

Unlike the examples in Basic Sample Applications (page 409), which focus on the Java code in the respective `Main.java` class files, the sample applications here focus on customizations made to the XML schema *before* generating the schema-derived Java binding classes.

Note: Although JAXB binding customizations must currently be made by hand, it is envisioned that a tool/wizard may eventually be written by Sun or a third party to make this process more automatic and easier in general. One of the goals of the JAXB technology is to standardize the format of binding declarations, thereby making it possible to create customization tools and to provide a standard interchange format between JAXB implementations.

This section just begins to scratch the surface of customizations you can make to JAXB bindings and validation methods. For more information, please refer to the *JAXB Specification* (`http://java.sun.com/xml/downloads/jaxb.html`).

Why Customize?

In most cases, the default bindings generated by the JAXB binding compiler will be sufficient to meet your needs. There are cases, however, in which you may want to modify the default bindings. Some of these include:

- Creating API documentation for the schema-derived JAXB packages, classes, methods and constants; by adding custom Javadoc tool annotations to your schemas, you can explain concepts, guidelines, and rules specific to your implementation.
- Providing semantically meaningful customized names for cases that the default XML name-to-Java identifier mapping cannot handle automatically; for example:
  - To resolve name collisions (as described in Appendix C.2.1 of the *JAXB Specification*). Note that the JAXB binding compiler detects and reports all name conflicts.
  - To provide names for typesafe enumeration constants that are not legal Java identifiers; for example, enumeration over integer values.
  - To provide better names for the Java representation of unnamed model groups when they are bound to a Java property or class.
  - To provide more meaningful package names than can be derived by default from the target namespace URI.
- Overriding default bindings; for example:
  - Specify that a model group should be bound to a class rather than a list.
  - Specify that a fixed attribute can be bound to a Java constant.
  - Override the specified default binding of XML Schema built-in datatypes to Java datatypes. In some cases, you might want to introduce an alternative Java class that can represent additional characteristics of the built-in XML Schema datatype.

Customization Overview

This section explains some core JAXB customization concepts:

- Inline and External Customizations
- Scope, Inheritance, and Precedence
- Customization Syntax
- Customization Namespace Prefix

Inline and External Customizations

Customizations to the default JAXB bindings are made in the form of *binding declarations* passed to the JAXB binding compiler. These binding declarations can be made in either of two ways:

- As inline annotations in a source XML schema
- As declarations in an external binding customizations file

For some people, using inline customizations is easier because you can see your customizations in the context of the schema to which they apply. Conversely, using an external binding customization file enables you to customize JAXB bindings without having to modify the source schema, and enables you to easily apply customizations to several schema files at once.

Note: You can combine the two types of customizations—for example, you could include a reference to an external binding customizations file in an inline annotation—but you cannot declare both an inline and external customization on the same schema element.

Each of these types of customization is described in more detail below.

Inline Customizations

Customizations to JAXB bindings made by means of inline *binding declarations* in an XML schema file take the form of `<xsd:appinfo>` elements embedded in schema `<xsd:annotation>` elements (`xsd:` is the XML schema namespace prefix, as defined in W3C *XML Schema Part 1: Structures*). The general form for inline customizations is shown below.

```
<xs:annotation>
   <xs:appinfo>
      .
      .
      binding declarations
      .
      .
   </xs:appinfo>
</xs:annotation>
```

Customizations are applied at the location at which they are declared in the schema. For example, a declaration at the level of a particular element would apply to that element only. Note that the XMLSchema namespace prefix must be used with the `<annotation>` and `<appinfo>` declaration tags. In the example above, `xs:` is used as the namespace prefix, so the declarations are tagged `<xs:annotation>` and `<xs:appinfo>`.

External Binding Customization Files

Customizations to JAXB bindings made by means of an external file containing binding declarations take the general form shown below.

```
<jxb:bindings schemaLocation = "xs:anyURI">
   <jxb:bindings node = "xs:string">*
      <binding declaration>
   <jxb:bindings>
</jxb:bindings>
```

- `schemaLocation` is a URI reference to the remote schema
- `node` is an XPath 1.0 expression that identifies the schema node within `schemaLocation` to which the given binding declaration is associated.

For example, the first `schemaLocation`/`node` declaration in a JAXB binding declarations file specifies the schema name and the root schema node:

```
<jxb:bindings schemaLocation="po.xsd" node="/xs:schema">
```

A subsequent `schemaLocation/node` declaration, say for a `simpleType` element named `ZipCodeType` in the above schema, would take the form:

```
<jxb:bindings node=”//xs:simpleType[@name=’ZipCodeType’]”>
```

Binding Customization File Format

Binding customization files should be straight ASCII text. The name or extension does not matter, although a typical extension, used in this chapter, is `.xjb`.

Passing Customization Files to the JAXB Binding Compiler

Customization files containing binding declarations are passed to the JAXB Binding compiler, `xjc`, using the following syntax:

```
xjc -b <file> <schema>
```

where *`<file>`* is the name of binding customization file, and *`<schema>`* is the name of the schema(s) you want to pass to the binding compiler.

You can have a single binding file that contains customizations for multiple schemas, or you can break the customizations into multiple bindings files; for example:

```
xjc schema1.xsd schema2.xsd schema3.xsd -b bindings123.xjb

xjc schema1.xsd schema2.xsd schema3.xsd -b bindings1.xjb -b
bindings2.xjb -b bindings3.xjb
```

Note that the ordering of schema files and binding files on the command line does not matter, although each binding customization file must be preceded by its own `-b` switch on the command line.

For more information about `xjc` compiler options in general, see JAXB Compiler Options (page 396).

Restrictions for External Binding Customizations

There are several rules that apply to binding declarations made in an external binding customization file that do not apply to similar declarations made inline in a source schema:

- The binding customization file must begin with the `jxb:bindings version` attribute, plus attributes for the JAXB and XMLSchema namespaces:

```
<jxb:bindings version="1.0"
      xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
      xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

- The remote schema to which the binding declaration applies must be identified explicitly in XPath notation by means of a `jxb:bindings` declaration specifying `schemaLocation` and `node` attributes:
  - `schemaLocation` – URI reference to the remote schema
  - `node` – XPath 1.0 expression that identifies the schema node within `schemaLocation` to which the given binding declaration is associated; in the case of the initial `jxb:bindings` declaration in the binding customization file, this node is typically `"/xs:schema"`

For information about XPath syntax, see *XML Path Language*, James Clark and Steve DeRose, eds., W3C, 16 November 1999. Available at `http://www.w3.org/TR/1999/REC-xpath-19991116`.

- Similarly, individual nodes within the schema to which customizations are to be applied must be specified using XPath notation; for example:

  ```
  <jxb:bindings node="//xs:complexType[@name='USAddress']">
  ```

  In such cases, the customization is applied to the node by the binding compiler as if the declaration was embedded inline in the node's `<xs:appinfo>` element.

To summarize these rules, the external binding element `<jxb:bindings>` is only recognized for processing by a JAXB binding compiler in three cases:

- When its parent is an `<xs:appinfo>` element
- When it is an ancestor of another `<jxb:bindings>` element
- When it is root element of a document—an XML document that has a `<jxb:bindings>` element as its root is referred to as an external binding declaration file

Scope, Inheritance, and Precedence

Default JAXB bindings can be customized or overridden at four different levels, or *scopes*, as described in Table 9–4.

Figure 10–1 illustrates the inheritance and precedence of customization declarations. Specifically, declarations towards the top of the pyramid inherit and supersede declarations below them. For example, Component declarations inherit from and supersede Definition declarations; Definition declarations inherit and supersede Schema declarations; and Schema declarations inherit and supersede Global declarations.

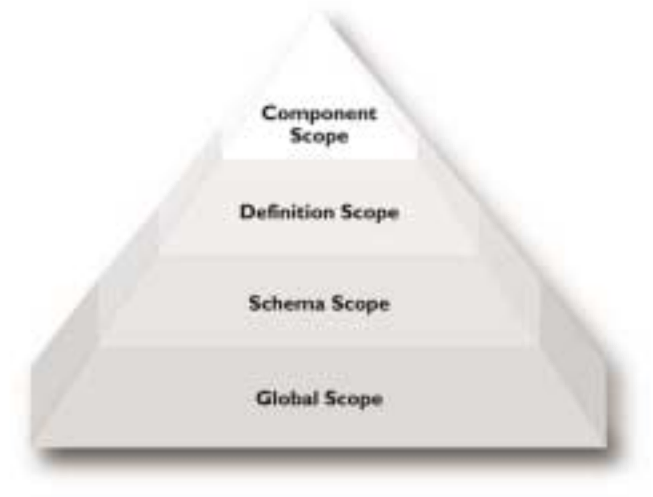


Figure 10–1 Customization Scope Inheritance and Precedence

Customization Syntax

The syntax for the four types of JAXB binding declarations, as well as the syntax for the XML-to-Java datatype binding declarations and the customization namespace prefix are described below.

- Global Binding Declarations
- Schema Binding Declarations
- Class Binding Declarations
- Property Binding Declarations
- <javaType> Binding Declarations
- Typesafe Enumeration Binding Declarations
- <javadoc> Binding Declarations
- Customization Namespace Prefix

Global Binding Declarations

Global scope customizations are declared with `<globalBindings>`. The syntax for global scope customizations is as follows:

```
<globalBindings>
  [ collectionType = "collectionType" ]
  [ fixedAttributeAsConstantProperty= "true" | "false" | "1" | "0" ]
  [ generateIsSetMethod= "true" | "false" | "1" | "0" ]
  [ enableFailFastCheck = "true" | "false" | "1" | "0" ]
  [ choiceContentProperty = "true" | "false" | "1" | "0" ]
  [ underscoreBinding  = "asWordSeparator" | "asCharInWord" ]
  [ typesafeEnumBase = "typesafeEnumBase" ]
  [ typesafeEnumMemberName = "generateName" | "generateError" ]
  [ enableJavaNamingConventions = "true" | "false" | "1" | "0" ]
  [ bindingStyle = "elementBinding" | "modelGroupBinding" ]
  [ <javaType> ... </javaType> ]*
</globalBindings>
```

- `collectionType` can be either `indexed` or any fully qualified class name that implements `java.util.List`.
- `fixedAttributeAsConstantProperty` can be either `true`, `false`, `1`, or `0`. The default value is `false`.
- `generateIsSetMethod` can be either `true`, `false`, `1`, or `0`. The default value is `false`.
- `enableFailFastCheck` can be either `true`, `false`, `1`, or `0`. If `enableFailFastCheck` is `true` or `1` and the JAXB implementation supports this optional checking, type constraint checking is performed when setting a property. The default value is `false`. Please note that the JAXB implementation does not support failfast validation.
- `choiceContentProperty` can be either `true`, `false`, `1`, or `0`. The default value is `false`. `choiceContentProperty` is not relevant when the `bindingStyle` is `elementBinding`. Therefore, if `bindingStyle` is specified as `elementBinding`, then the `choiceContentProperty` must result in an invalid customization.
- `underscoreBinding` can be either `asWordSeparator` or `asCharInWord`. The default value is `asWordSeparator`.
- `enableJavaNamingConventions` can be either `true`, `false`, `1`, or `0`. The default value is `true`.
- `typesafeEnumBase` can be a list of QNames, each of which must resolve to a simple type definition. The default value is `xs:NCName`. See Typesafe Enumeration Binding Declarations (page 434) for information about

localized mapping of `simpleType` definitions to Java `typesafe enum` classes.

- `typesafeEnumMemberName` can be either `generateError` or `generateName`. The default value is `generateError`.
- `bindingStyle` can be either `elementBinding`, or `modelGroupBinding`. The default value is `elementBinding`.
- `<javaType>` can be zero or more javaType binding declarations. See <javaType> Binding Declarations (page 432) for more information.

`<globalBindings>` declarations are only valid in the `annotation` element of the top-level `schema` element. There can only be a single instance of a `<globalBindings>` declaration in any given schema or binding declarations file. If one source schema includes or imports a second source schema, the `<globalBindings>` declaration must be declared in the first source schema.

Schema Binding Declarations

Schema scope customizations are declared with `<schemaBindings>`. The syntax for schema scope customizations is:

```
<schemaBindings>
  [ <package> package </package> ]
  [ <nameXmlTransform> ... </nameXmlTransform> ]*
</schemaBindings>

<package [ name = "packageName" ]
  [ <javadoc> ... </javadoc> ]
</package>

<nameXmlTransform>
  [ <typeName [ suffix="suffix" ]
              [ prefix="prefix" ] /> ]
  [ <elementName [ suffix="suffix" ]
                 [ prefix="prefix" ] /> ]
  [ <modelGroupName [ suffix="suffix" ]
                    [ prefix="prefix" ] /> ]
  [ <anonymousTypeName [ suffix="suffix" ]
                       [ prefix="prefix" ] /> ]
</nameXmlTransform>
```

As shown above, `<schemaBinding>` declarations include two subcomponents:

- `<package>...</package>` specifies the name of the package and, if desired, the location of the API documentation for the schema-derived classes.

- `<nameXmlTransform>...</nameXmlTransform>` specifies customizations to be applied.

Class Binding Declarations

The `<class>` binding declaration enables you to customize the binding of a schema element to a Java content interface or a Java `Element` interface. `<class>` declarations can be used to customize:

- A name for a schema-derived Java interface
- An implementation class for a schema-derived Java content interface.

The syntax for `<class>` customizations is:

```
<class [ name = "className"]
   [ implClass= "implClass" ] >
   [ <javadoc> ... </javadoc> ]
</class>
```

- `name` is the name of the derived Java interface. It must be a legal Java interface name and must not contain a package prefix. The package prefix is inherited from the current value of package.
- `implClass` is the name of the implementation class for `className` and must include the complete package name.
- The `<javadoc>` element specifies the Javadoc tool annotations for the schema-derived Java interface. The string entered here must use `CDATA` or < to escape embedded HTML tags.

Property Binding Declarations

The `<property>` binding declaration enables you to customize the binding of an XML schema element to its Java representation as a property. The scope of customization can either be at the definition level or component level depending upon where the `<property>` binding declaration is specified.

The syntax for `<property>` customizations is:

```
<property[ name = "propertyName"]
   [ collectionType = "propertyCollectionType" ]
   [ fixedAttributeAsConstantProperty= "true" | "false" | "1" | "0" ]
   [ generateIsSetMethod= "true" | "false" | "1" | "0" ]
   [ enableFailFastCheck="true" | "false" | "1" | "0" ]
   [ <baseType> ... </baseType> ]
   [ <javadoc> ... </javadoc> ]
</property>
```

```
<baseType>
   <javaType> ... </javaType>
</baseType>
```

- `name` defines the customization value `propertyName`; it must be a legal Java identifier.
- `collectionType` defines the customization value `propertyCollection-Type`, which is the collection type for the property. `propertyCollection-Type` if specified, can be either `indexed` or any fully-qualified class name that implements `java.util.List`.
- `fixedAttributeAsConstantProperty` defines the customization value `fixedAttributeAsConstantProperty`. The value can be either `true`, `false`, `1`, or `0`.
- `generateIsSetMethod` defines the customization value of `generateIs-SetMethod`. The value can be either `true`, `false`, `1`, or `0`.
- `enableFailFastCheck` defines the customization value `enableFail-FastCheck`. The value can be either `true`, `false`, `1`, or `0`. Please note that the JAXB implementation does not support failfast validation.
- `<javadoc>` customizes the Javadoc tool annotations for the property's getter method.

<javaType> Binding Declarations

The `<javaType>` declaration provides a way to customize the translation of XML datatypes to and from Java datatypes. XML provides more datatypes than Java, and so the `<javaType>` declaration lets you specify custom datatype bindings when the default JAXB binding cannot sufficiently represent your schema.

The target Java datatype can be a Java built-in datatype or an application-specific Java datatype. If an application-specific datatype is used as the target, your implementation must also provide parse and print methods for unmarshalling and marshalling data. To this end, the JAXB specification supports a `parseMethod` and `printMethod`:

- The `parseMethod` is called during unmarshalling to convert a string from the input document into a value of the target Java datatype.
- The `printMethod` is called during marshalling to convert a value of the target type into a lexical representation.

If you prefer to define your own datatype conversions, JAXB defines a static class, `DatatypeConverter`, to assist in the parsing and printing of valid lexical representations of the XML Schema built-in datatypes.

The syntax for the `<javaType>` customization is:

```
<javaType name=" javaType"
      [ xmlType=" xmlType" ]
      [ hasNsContext = “true” | “false” ]
      [ parseMethod=" parseMethod" ]
      [ printMethod=" printMethod" ]>
```

- `name` is the Java datatype to which `xmlType` is to be bound.
- `xmlType` is the name of the XML Schema datatype to which `javaType` is to bound; this attribute is required when the parent of the `<javaType>` declaration is `<globalBindings>`.
- `parseMethod` is the name of the parse method to be called during unmarshalling.
- `printMethod` is the name of the print method to be called during marshalling.
- `hasNsContext` allows a namespace context to be specified as a second parameter to a print or a parse method; can be either `true`, `false`, `1`, or `0`. By default, this attribute is `false`, and in most cases you will not need to change it.

The `<javaType>` declaration can be used in:

- A `<globalBindings>` declaration
- An annotation element for simple type definitions, `GlobalBindings`, and `<basetype>` declarations.
- A `<property>` declaration.

See MyDatatypeConverter Class (page 441) for an example of how `<javaType>` declarations and the `DatatypeConverterInterface` interface are implemented in a custom datatype converter class.

Typesafe Enumeration Binding Declarations

The typesafe enumeration declarations provide a localized way to map XML `simpleType` elements to Java `typesafe enum` classes. There are two types of typesafe enumeration declarations you can make:

- `<typesafeEnumClass>` lets you map an entire `simpleType` class to `typesafe enum` classes.
- `<typesafeEnumMember>` lets you map just selected members of a `simpleType` class to `typesafe enum` classes.

In both cases, there are two primary limitations on this type of customization:

- Only `simpleType` definitions with enumeration facets can be customized using this binding declaration.
- This customization only applies to a single `simpleType` definition at a time. To map sets of similar `simpleType` definitions on a global level, use the `typesafeEnumBase` attribute in a `<globalBindings>` declaration, as described Global Binding Declarations (page 429).

The syntax for the `<typesafeEnumClass>` customization is:

```
<typesafeEnumClass[ name = "enumClassName" ]
  [ <typesafeEnumMember> ... </typesafeEnumMember> ]*
  [ <javadoc> enumClassJavadoc </javadoc> ]
</typesafeEnumClass>
```

- `name` must be a legal Java Identifier, and must not have a package prefix.
- `<javadoc>` customizes the Javadoc tool annotations for the enumeration class.
- You can have zero or more `<typesafeEnumMember>` declarations embedded in a `<typesafeEnumClass>` declaration.

The syntax for the `<typesafeEnumMember>` customization is:

```
<typesafeEnumMember name = "enumMemberName">
            [ value = "enumMemberValue" ]
  [ <javadoc> enumMemberJavadoc </javadoc> ]
</typesafeEnumMember>
```

- `name` must always be specified and must be a legal Java identifier.
- `value` must be the enumeration value specified in the source schema.
- `<javadoc>` customizes the Javadoc tool annotations for the enumeration constant.

For inline annotations, the `<typesafeEnumClass>` declaration must be specified in the annotation element of the `<simpleType>` element. The `<typesafeEnum-Member>` must be specified in the annotation element of the enumeration member. This allows the enumeration member to be customized independently from the enumeration class.

For information about typesafe enum design patterns, see the sample chapter of Joshua Bloch's *Effective Java Programming* on the Java Developer Connection.

<javadoc> Binding Declarations

The `<javadoc>` declaration lets you add custom Javadoc tool annotations to schema-derived JAXB packages, classes, interfaces, methods, and fields. Note that `<javadoc>` declarations cannot be applied globally—that is, they are only valid as a sub-elements of other binding customizations.

The syntax for the `<javadoc>` customization is:

```
<javadoc>
  Contents in &lt;b>Javadoc&lt;\b> format.
</javadoc>

or

<javadoc>
  <<![CDATA[
  Contents in <b>Javadoc<\b> format
  ]]>
</javadoc>
```

Note that documentation strings in `<javadoc>` declarations applied at the package level must contain `<body>` open and close tags; for example:

```
<jxb:package name="primer.myPo">
       <jxb:javadoc><![CDATA[<body>Package level documentation
for generated package primer.myPo.</body>]]>
</jxb:javadoc>
      </jxb:package>
```

Customization Namespace Prefix

All standard JAXB binding declarations must be preceded by a namespace prefix that maps to the JAXB namespace URI (`http://java.sun.com/xml/ns/jaxb`). For example, in this sample, `jxb:` is used. To this end, any schema you want to customize with standard JAXB binding declarations *must* include the JAXB namespace declaration and JAXB version number at the top of the schema file. For example, in `po.xsd` for Sample Application 6, the namespace declaration is as follows:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
            jxb:version="1.0">
```

A binding declaration with the `jxb` namespace prefix would then take the form:

```
<xsd:annotation>
   <xsd:appinfo>
      <jxb:globalBindings binding declarations />
      <jxb:schemaBindings>
         .
         .
         binding declarations
         .
         .
      </jxb:schemaBindings>
   </xsd:appinfo>
</xsd:annotation>
```

Note that in this example, the `globalBindings` and `schemaBindings` declarations are used to specify, respectively, global scope and schema scope customizations. These customization scopes are described in more detail in Scope, Inheritance, and Precedence (page 427).

Sample Application 6

Sample Application 6 illustrates some basic customizations made by means of inline annotations to an XML schema named `po.xsd`. In addition, this sample implements a custom datatype converter class, `MyDatatypeConverter.java`, which illustrates print and parse methods in the `<javaType>` customization for handling custom datatype conversions.

To summarize this example:

1. `po.xsd` is an XML schema containing inline binding customizations.
2. `MyDatatypeConverter.java` is a Java class file that implements print and parse methods specified by `<javaType>` customizations in `po.xsd`.

3. `Main.java` is the primary class file in Sample Application 6, which uses the schema-derived classes generated by the JAXB compiler.

Key customizations in this sample, and the custom `MyDatatypeConverter.java` class, are described in more detail below.

- Customized Schema
- Global Binding Declarations
- Global Binding Declarations
- Schema Binding Declarations
- Class Binding Declarations
- Property Binding Declarations
- MyDatatypeConverter Class

Customized Schema

The customized schema used in `SampleApp6` is in the file `<`*`JWSDP_HOME`*`>/jaxb-1.0/examples/users-guide/SampleApp6/po.xsd`. The customizations are in the `<xsd:annotation>` tags.

Global Binding Declarations

The code below shows the `globalBindings` declarations in `po.xsd`:

```
<jxb:globalBindings
        fixedAttributeAsConstantProperty="true"
        collectionType="java.util.Vector"
        typesafeEnumBase="xsd:NCName"
        choiceContentProperty="false"
        typesafeEnumMemberName="generateError"
        bindingStyle="elementBinding"
        enableFailFastCheck="false"
        generateIsSetMethod="false"
        underscoreBinding="asCharInWord"/>
```

In this example, all values are set to the defaults except for `collectionType`.

- Setting `collectionType` to `java.util.Vector` specifies that all lists in the generated implementation classes should be represented internally as vectors. Note that the class name you specify for `collectionType` must implement `java.util.List` and be callable by `newInstance`.

- Setting fixedAttributeAsConstantProperty to true indicates that all fixed attributes should be bound to Java constants. By default, fixed attributes are just mapped to either simple or collection property, which ever is more appropriate.
- Please note that the JAXB implementation does not support the enableFailFastCheck attribute.
- If typesafeEnumBase to xsd:string it would be a global way to specify that all simple type definitions deriving directly or indirectly from xsd:string and having enumeration facets should be bound by default to a typesafe enum. If typesafeEnumBase is set to an empty string, "", no simple type definitions would ever be bound to a typesafe enum class by default. The value of typesafeEnumBase can be any atomic simple type definition except xsd:boolean and both binary types.

Note: Using typesafe enums enables you to map schema enumeration values to Java constants, which in turn makes it possible to do compares on Java constants rather than string values.

Schema Binding Declarations

The following code shows the schema binding declarations in po.xsd:

```
<jxb:schemaBindings>
      <jxb:package name="primer.myPo">
         <jxb:javadoc>
   <![CDATA[<body> Package level documentation for generated
package primer.myPo.
              </body>]]>
         </jxb:javadoc>
      </jxb:package>
      <jxb:nameXmlTransform>
          <jxb:elementName suffix="Element"/>
      </jxb:nameXmlTransform>
    </jxb:schemaBindings>
```

- <jxb:package name="primer.myPo"/> specifies the primer.myPo as the package in which the schema-derived classes should be generated.
- <jxb:nameXmlTransform> specifies that all generated Java element interfaces should have Element appended to the generated names by default. For example, when the JAXB compiler is run against this schema, the ele-

ment interfaces `CommentElement` and `PurchaseOrderElement` will be generated. By contrast, without this customization, the default binding would instead generate `Comment` and `PurchaseOrder`.

This customization is useful if a schema uses the same name in different symbol spaces; for example, in global element and type definitions. In such cases, this customization enables you to resolve the collision with one declaration rather than having to individually resolve each collision with a separate binding declaration.

- `<jxb:javadoc>` specifies customized Javadoc tool annotations for the `primer.myPo` package. Note that, unlike the `<javadoc>` declarations at the class level, below, the opening and closing `<body>` tags must be included when the `<javadoc>` declaration is made at the package level.

Class Binding Declarations

The following code shows the class binding declarations in `po.xsd`:

```
<xsd:complexType name="PurchaseOrderType">
      <xsd:annotation>
      <xsd:appinfo>
         <jxb:class name="POType">
            <jxb:javadoc>
           A &lt;b>Purchase Order&lt;/b> consists of addresses
and items.
            </jxb:javadoc>
         </jxb:class>
      </xsd:appinfo>
      </xsd:annotation>
      .
      .
      .
</xsd:complexType>
```

The Javadoc tool annotations for the schema-derived `POType` class will contain the description `"A &lt;b>Purchase Order&lt;/b> consists of addresses and items."` The `&lt;` is used to escape the opening bracket on the `<b>` HTML tags.

Note: When a `<class>` customization is specified in the `appinfo` element of a `complexType` definition, as it is here, the `complexType` definition is bound to a Java content interface.

Later in `po.xsd`, another `<javadoc>` customization is declared at this class level, but this time the HTML string is escaped with `CDATA`:

```
<xsd:annotation>
 <xsd:appinfo>
    <jxb:class>
      <jxb:javadoc>
     <![CDATA[ First line of documentation for a
<b>USAddress</b>.]]>
       </jxb:javadoc>
    </jxb:class>
  </xsd:appinfo>
  </xsd:annotation>
```

Note: If you want to include HTML markup tags in a `<jaxb:javadoc>` customization, you must enclose the data within a `CDATA` section or escape all left angle brackets using `&lt;`. See *XML 1.0 2nd Edition* for more information (`http://www.w3.org/TR/2000/REC-xml-20001006#sec-cdata-sect`).

Property Binding Declarations

Of particular interest here is the `generateIsSetMethod` customization, which causes two additional property methods, `isSetQuantity` and `unsetQuantity`, to be generated. These methods enable a client application to distinguish between schema default values and values occurring explicitly within an instance document.

For example, in `po.xsd`:

```
<xsd:complexType name="Items">
   <xsd:sequence>
      <xsd:element name="item" minOccurs="1"
maxOccurs="unbounded">
         <xsd:complexType>
            <xsd:sequence>
           <xsd:element name="productName" type="xsd:string"/>
            <xsd:element name="quantity" default="10">
            <xsd:annotation>
               <xsd:appinfo>
                  <jxb:property generateIsSetMethod="true"/>
               </xsd:appinfo>
            </xsd:annotation>
         .
         .
```

```
          .
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
```

The `@generateIsSetMethod` applies to the `quantity` element, which is bound to a property within the `Items.ItemType` interface. `unsetQuantity` and `isSetQuantity` methods are generated in the `Items.ItemType` interface.

MyDatatypeConverter Class

The purpose of the `<`*`JWSDP_HOME`*`>/jaxb-1.0/examples/users-guide/SampleApp6/MyDatatypeConverter` class, shown below, is to provide a way to customize the translation of XML datatypes to and from Java datatypes by means of a `<javaType>` customization.

```
package primer;
import java.math.BigInteger;
import javax.xml.bind.DatatypeConverter;

public class MyDatatypeConverter {

  public static short parseIntegerToShort(String value) {
  BigInteger result = DatatypeConverter.parseInteger(value);
  return (short)(result.intValue());
  }

  public static String printShortToInteger(short value) {
        BigInteger result = BigInteger.valueOf(value);
        return DatatypeConverter.printInteger(result);
  }

  public static int parseIntegerToInt(String value) {
  BigInteger result = DatatypeConverter.parseInteger(value);
  return result.intValue();
  }

  public static String printIntToInteger(int value) {
       BigInteger result = BigInteger.valueOf(value);
       return DatatypeConverter.printInteger(result);
  }
};
```

The following code shows how the `MyDatatypeConverter` class is referenced in a `<javaType>` declaration in `po.xsd`:

```
<xsd:simpleType name="ZipCodeType">
  <xsd:annotation>
     <xsd:appinfo>
        <jxb:javaType name="int"
parseMethod="primer.MyDatatypeConverter.parseIntegerToInt"
printMethod="primer.MyDatatypeConverter.printIntTo Integer" />
     </xsd:appinfo>
  </xsd:annotation>
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="10000"/>
    <xsd:maxInclusive value="99999"/>
  </xsd:restriction>
</xsd:simpleType>
```

In this example, the `jxb:javaType` binding declaration overrides the default JAXB binding of this type to `java.math.BigInteger`. For the purposes of Sample Application 6, the restrictions on `ZipCodeType`—specifically that legal US ZIP codes are limited to five digits—make it so all valid values can easily fit within the Java primitive datatype `int`. Note also that, because `<jxb:javaType name="int"/>` is declared within `ZipCodeType`, the customization applies to all JAXB properties that reference this `simpleType` definition, including the `getZip` and `setZip` methods.

Sample Application 7

Sample Application is very similar to Sample Application 6. As with Sample Application 6, the customizations in Sample Application 7 are made by means inline binding declarations in the XML schema for the application, `po.xsd`.

The global, schema, and package, and most of the class customizations for Sample Applications 6 and 7 are identical. Where Sample Application 7 differs from Sample Application 6 is in the `parseMethod` and `printMethod` used for converting XML data to the Java `int` datatype.

Specifically, rather than using methods in the custom `MyDataTypeConverter` class to perform these datatype conversions, Sample Application 7 uses the built-in methods provided by `javax.xml.bind.DatatypeConverter`:

```
<xsd:simpleType name="ZipCodeType">
  <xsd:annotation>
     <xsd:appinfo>
        <jxb:javaType name="int"
   parseMethod="javax.xml.bind.DatatypeConverter.parseInt"
   printMethod="javax.xml.bind.DatatypeConverter.printInt"/>
     </xsd:appinfo>
  </xsd:annotation>
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="10000"/>
    <xsd:maxInclusive value="99999"/>
  </xsd:restriction>
</xsd:simpleType>
```

Sample Application 8

Sample Application 8 is identical to Sample Application 7, except that the binding declarations in Sample Application 8 are made by means of an external binding declarations file rather than inline in the source XML schema.

The binding customization file used in Sample Application 8 is `<`*`JWSDP_HOME`*`>/jaxb-1.0/examples/users-guide/SampleApp8/binding.xjb`.

This section compares the customization declarations in `bindings.xjb` with the analogous declarations used in the XML schema, `po.xsd`, in Sample Application 7. The two sets of declarations achieve precisely the same results.

- JAXB Version, Namespace, and Schema Attributes
- Global and Schema Binding Declarations
- Class Declarations

JAXB Version, Namespace, and Schema Attributes

All JAXB binding declarations files must begin with:

- JAXB version number
- Namespace declarations
- Schema name and node

The version, namespace, and schema declarations in `bindings.xjb` are as follows:

```
<jxb:bindings version="1.0"
               xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <jxb:bindings schemaLocation="po.xsd" node="/xs:schema">
        .
        <binding_declarations>
        .
  </jxb:bindings>
<!-- schemaLocation="po.xsd" node="/xs:schema" -->
</jxb:bindings>
```

JAXB Version Number

An XML file with a root element of `<jaxb:bindings>` is considered an external binding file. The root element must specify the JAXB version attribute with which its binding declarations must comply; specifically the root `<jxb:bindings>` element must contain either a `<jxb:version>` declaration or a `version` attribute. By contrast, when making binding declarations inline, the JAXB version number is made as attribute of the `<xsd:schema>` declaration:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
            jxb:version="1.0">
```

Namespace Declarations

As shown in JAXB Version, Namespace, and Schema Attributes (page 444), the namespace declarations in the external binding declarations file include both the JAXB namespace and the XMLSchema namespace. Note that the prefixes used in this example could in fact be anything you want; the important thing is to consistently use whatever prefixes you define here in subsequent declarations in the file.

Schema Name and Schema Node

The fourth line of the code in JAXB Version, Namespace, and Schema Attributes (page 444) specifies the name of the schema to which this binding declarations file will apply, and the schema node at which the customizations will first take effect. Subsequent binding declarations in this file will reference specific nodes within the schema, but this first declaration should encompass the schema as a whole; for example, in `bindings.xjb`:

```
<jxb:bindings schemaLocation="po.xsd" node="/xs:schema">
```

Global and Schema Binding Declarations

The global schema binding declarations in `bindings.xjb` are the same as those in `po.xsd` for Sample Application 7. The only difference is that because the declarations in `po.xsd` are made inline, you need to embed them in `<xs:appinfo>` elements, which are in turn embedded in `<xs:annotation>` elements. Embedding declarations in this way is unnecessary in the external bindings file.

```
<jxb:globalBindings
    fixedAttributeAsConstantProperty="true"
    collectionType="java.util.Vector"
    typesafeEnumBase="xs:NCName"
    choiceContentProperty="false"
    typesafeEnumMemberName="generateError"
    bindingStyle="elementBinding"
    enableFailFastCheck="false"
    generateIsSetMethod="false"
    underscoreBinding="asCharInWord"/>
<jxb:schemaBindings>
    <jxb:package name="primer.myPo">
       <jxb:javadoc><![CDATA[<body>Package level documentation
for generated package primer.myPo.</body>]]>
       </jxb:javadoc>
    </jxb:package>
    <jxb:nameXmlTransform>
        <jxb:elementName suffix="Element"/>
    </jxb:nameXmlTransform>
</jxb:schemaBindings>
```

By comparison, the syntax used in po.xsd for Sample Application 7 is:

```
<xsd:annotation>
  <xsd:appinfo>
    <jxb:globalBindings
        .
        <binding_declarations>
        .
    <jxb:schemaBindings>
        .
        <binding_declarations>
        .
    </jxb:schemaBindings>
  </xsd:appinfo>
</xsd:annotation>
```

Class Declarations

The class-level binding declarations in bindings.xjb differ from the analogous declarations in po.xsd for Sample Application 7 in two ways:

- As with all other binding declarations in bindings.xjb, you do not need to embed your customizations in schema <xsd:appinfo> elements.
- You must specify the schema node to which the customization will be applied. The general syntax for this type of declaration is:

  `<jxb:bindings node="//<node_type>[@name='<node_name>']">`

For example, the following code shows binding declarations for the complexType named USAddress.

```
<jxb:bindings node="//xs:complexType[@name='USAddress']">
  <jxb:class>
    <jxb:javadoc><![CDATA[First line of documentation for a
<b>USAddress</b>.]]></jxb:javadoc>
  </jxb:class>

  <jxb:bindings node=".//xs:element[@name='name']">
    <jxb:property name="toName"/>
  </jxb:bindings>

  <jxb:bindings node=".//xs:element[@name='zip']">
    <jxb:property name="zipCode"/>
  </jxb:bindings>
</jxb:bindings>
<!-- node="//xs:complexType[@name='USAddress']" -->
```

Note in this example that `USAddress` is the parent of the child elements `name` and `zip`, and therefore a `</jxb:bindings>` tag encloses the `bindings` declarations for the child elements as well as the class-level `javadoc` declaration.

Sample Application 9

Sample Application 9 illustrates how to resolve name conflicts—that is, places in which a declaration in a source schema uses the same name as another declaration in that schema (namespace collisions), or places in which a declaration uses a name that does translate by default to a legal Java name.

Note: Many name collisions can occur because XSD Part 1 introduces six unique symbol spaces based on type, while Java only has only one. There is a symbols space for type definitions, elements, attributes, and group definitions. As a result, a valid XML schema can use the exact same name for both a type definition and a global element declaration.

For the purposes of this sample application, it is recommended that you run the `ant fail` command in the Sample Application 9 directory to display the error output generated by the `xjc` compiler. The XML schema for Sample Application 9, `example.xsd`, contains deliberate name conflicts.

In addition to illustrating name conflicts, Sample Application 9 shows how to:

- Bind a `choice` model group to its own interface
- Add elements to a `List` property using `java.util.List.add`

Like Sample Application 8, Sample Application 9 uses an external binding declarations file, `binding.xjb`, to define the JAXB binding customizations.

- The example.xsd Schema
- Looking at the Conflicts
- Output From ant fail
- The binding.xjb Declarations File
- Resolving the Conflicts in example.xsd
- Customizing a choice Model Group
- Adding Elements to a List Property

The example.xsd Schema

The XML schema, `<JWSDP_HOME>/jaxb-1.0/examples/users-guide/SampleApp9/example.xsd`, used in Sample Application 9 illustrates common name conflicts encountered when attempting to bind XML names to unique Java identifiers in a Java package. The schema declarations that result in name conflicts are highlighted in bold below.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
           jxb:version="1.0">

  <xs:element name="Class" type="xs:int"/>
  <xs:element name="FooBar" type="FooBar"/>
  <xs:complexType name="FooBar">
    <xs:sequence>
      <xs:element name="foo" type="xs:int"/>
      <xs:element ref="Class"/>
      <xs:choice>
         <xs:element name="phoneNumber" type="xs:string"/>
         <xs:element name="speedDial" type="xs:int"/>
      </xs:choice>
      <xs:element name="listOfChoices" type="ListOfChoices"/>
      <xs:element name="zip" type="xs:integer"/>
    </xs:sequence>
    <xs:attribute name="zip" type="xs:string"/>
  </xs:complexType>

  <xs:complexType name="ListOfChoices">
    <xs:choice maxOccurs="unbounded">
      <xs:element name="bool" type="xs:boolean"/>
      <xs:element name="comment" type="xs:string"/>
      <xs:element name="value" type="xs:int"/>
    </xs:choice>
  </xs:complexType>
</xs:schema>
```

Looking at the Conflicts

The first conflict in `example.xsd` is the declaration of the `element` name `Class`:

```
<xs:element name="Class" type="xs:int"/>
```

`Class` is a reserved word in Java, and while it is legal in the XML schema language, it cannot be used as a name for a schema-derived class generated by JAXB.

When this schema is run against the JAXB binding compiler with the `ant fail` command, the following error message is returned:

```
[xjc] [ERROR] Attempt to create a property having the same name
as the reserved word "Class". [xjc] line 6 of example.xsd
```

The second conflict is that there are an `element` and a `complexType` that both use the name `Foobar`:

```
<xs:element name=”FooBar” type=”FooBar”/>
<xs:complexType name=”FooBar”>
```

In this case, the error messages returned are:

```
[xjc] [ERROR] A property with the same name "Zip" is generated
from more than one schema component. [xjc] line 22 of
example.xsd
[xjc] [ERROR] (Relevant to above error) another one is generated
from this schema component. [xjc] line 20 of example.xsd
```

The third conflict is that there are an `element` and an `attribute` both named `zip`:

```
<xs:element name=”zip” type=”xs:integer”/>
<xs:attribute name=”zip” type=”xs:string”/>
```

The error messages returned here are:

```
[xjc] [ERROR] A property with the same name "Zip" is generated
from more than one schema component. [xjc] line 22 of
example.xsd
[xjc] [ERROR] (Relevant to above error) another one is generated
from this schema component. [xjc] line 20 of example.xsd
```

Output From ant fail

Here is the complete output returned by running `ant fail` in the Sample Application 9 directory:

```
[echo] Compiling the schema w/o external binding file (name
collision errors expected)...
[xjc] Compiling file:/C:/Documents and Settings/mama/
jwsdp-1.1/jaxb-1.0/examples/users-guide/SampleApp9/example.xsd
[xjc] [ERROR] Attempt to create a property having the same name
as the reserved word "Class".
[xjc]   line 6 of example.xsd
[xjc] [ERROR] A property with the same name "Zip" is generated
from more than one schema component.
[xjc]   line 22 of example.xsd
[xjc] [ERROR] (Relevant to above error) another one is generated
from this schema component.
[xjc]   line 20 of example.xsd

[xjc] [ERROR] A class/interface with the same name
"generated.FooBar" is already in use.
[xjc]   line 9 of example.xsd
[xjc] [ERROR] (Relevant to above error) another one is generated
from here.
[xjc]   line 23 of example.xsd
```

The binding.xjb Declarations File

The `<`*`JWSDP_HOME`*`>/jaxb-1.0/examples/users-guide/SampleApp9/binding.xjb` binding declarations file resolves the conflicts in examples.xsd by means of several customizations.

Resolving the Conflicts in example.xsd

The first conflict in `example.xsd`, using the Java reserved name `Class` for an element name, is resolved in `binding.xjb` with the `<class>` and `<property>` declarations on the schema element node `Class`:

```
<jxb:bindings node="//xs:element[@name='Class']">
  <jxb:class name="Clazz"/>
  <jxb:property name="Clazz"/>
</jxb:bindings>
```

The second conflict in example.xsd, the namespace collision between the element FooBar and the complexType FooBar, is resolved in binding.xjb by using a <nameXmlTransform> declaration at the <schemaBindings> level to append the suffix Element to all element definitions.

This customization handles the case where there are many name conflicts due to systemic collisions between two symbol spaces, usually named type definitions and global element declarations. By appending a suffix or prefix to every Java identifier representing a specific XML symbol space, this single customization resolves all name collisions:

```
<jxb:schemaBindings>
  <jxb:package name="example"/>
    <jxb:nameXmlTransform>
      <jxb:elementName suffix="Element"/>
    </jxb:nameXmlTransform>
</jxb:schemaBindings>
```

The third conflict in example.xsd, the namespace collision between the element zip and the attribute zip, is resolved in binding.xjb by mapping the attribute zip to property named zipAttribute:

```
<jxb:bindings node=".//xs:attribute[@name='zip']">
  <jxb:property name="zipAttribute"/>
</jxb:bindings>
```

Running ant in the SampleApp9 directory will pass the customizations in binding.xjb to the xjc binding compiler, which will then resolve the conflicts in example.xsd in the schema-derived Java classes.

Customizing a choice Model Group

The binding.xjb binding declarations file also demonstrates a way to override the default derived names for choice model groups in example.xsd by means of <jxb:class> and <jxb:property> declarations:

```
<jxb:bindings node="./xs:sequence/xs:choice">
  <jxb:class name="MyChoices"/>
    <jxb:property name="choices"/>
</jxb:bindings>

<jxb:bindings
```

```
node="//xs:complexType[@name='ListOfChoices']/xs:choice">
  <jxb:class name="MultipleChoice"/>
  <jxb:property name="ChoiceList"/>
</jxb:bindings>
```

This customization results in the `choice` model group being bound to its own content interface. For example, given the following `choice` model group:

```
<xs:choice>
  <xs:element name="bool" type="xs:boolean"/>
  <xs:element name="comment" type="xs:string"/>
  <xs:element name="value" type="xs:int'/>
</xs:choice>
```

the customization shown above causes JAXB to generate the following Java class:

```
/**
 * Java content class for model group.
 */
  public interface MultipleChoice {
        int getValue();
        void setValue(int value);

        java.lang.String getComment();
        void setComment(java.lang.String value);

        boolean isBool();
        void setBool(boolean value);

        Object getContent();
    }
```

Calling `getContent` returns the current value of the `Choice` content. The setters of this `choice` are just like radio buttons; setting one unsets the previously set one. This class represents the data representing the `choice`.

Adding Elements to a List Property

Sample Application 9 demonstrates how to use methods in `java.util.List` to add elements to an XML schema `choice` list. This is a three-step model:

1. The `choice` list is defined in an XML schema; for example, in `example.xsd`, a `complexType` named `ListOfChoices` is defined:

```
<xs:complexType name="ListOfChoices">
  <xs:choice maxOccurs="unbounded">
    <xs:element name="bool" type="xs:boolean"/>
    <xs:element name="comment" type="xs:string"/>
    <xs:element name="value" type="xs:int"/>
  </xs:choice>
</xs:complexType>
```

2. Next, binding declarations are used to customize the binding of the choice list to a Java `class` and `property`; for example in `binding.xjb`:

```
<jxb:bindings
node="//xs:complexType[@name='ListOfChoices']/xs:choice">
  <jxb:class name="MultipleChoice"/>
  <jxb:property name="ChoiceList"/>
 </jxb:bindings>
```

3. Finally, the schema-derived method for a JAXB list property results in the getter being generated. Modifications to this list are made using the standard `java.util.List` API; in this example `java.util.List.add` method is being called. For example, in `Main.java`, the main Java class file for Sample Application 9:

```
ListOfChoices loc= fb.getListOfChoices();
List lst = loc.getChoiceList();

ObjectFactory of = new example.ObjectFactory();
ListOfChoices.MultipleChoice choice =
of.createListOfChoicesMultipleChoice();

choice.setComment("This is a program added comment");
lst.add(choice);

choice = of.createListOfChoicesMultipleChoice();
choice.setBool(true);
lst.add(choice);
```

```
choice = of.createListOfChoicesMultipleChoice();
choice.setValue(100);
lst.add(choice);
```

11

Building Web Services With JAX-RPC

Dale Green

JAX-RPC stands for Java API for XML-based RPC. It's an API for building Web services and clients that used remote procedure calls (RPC) and XML. Often used in a distributed client/server model, an RPC mechanism enables clients to execute procedures on other systems.

In JAX-RPC, a remote procedure call is represented by an XML-based protocol such as SOAP. The SOAP specification defines envelope structure, encoding rules, and a convention for representing remote procedure calls and responses. These calls and responses are transmitted as SOAP messages over HTTP. In this release, JAX-RPC relies on SOAP 1.1 and HTTP 1.1.

Although JAX-RPC relies on complex protocols, the API hides this complexity from the application developer. On the server side, the developer specifies the remote procedures by defining methods in an interface written in the Java programming language. The developer also codes one or more classes that implement those methods. Client programs are also easy to code. A client creates a proxy, a local object representing the service, and then simply invokes methods on the proxy.

With JAX-RPC, clients and Web services have a big advantage—the platform independence of the Java programming language. In addition, JAX-RPC is not restrictive: a JAX-RPC client can access a Web service that is not running on the Java platform and vice versa. This flexibility is possible because JAX-RPC uses technologies defined by the World Wide Web Consortium (W3C): HTTP, SOAP, and the Web Service Description Language (WSDL). WSDL specifies an XML format for describing a service as a set of endpoints operating on messages.

If you're new to the Java API for XML-based RPC (JAX-RPC), this chapter is the place to start. After briefly describing JAX-RPC, the chapter shows you how to build a simple Web service and client. For advanced users, the chapter continues to focus on examples by presenting code listings and step-by-step instructions for creating dynamic clients.

A Simple Example: HelloWorld

This example shows you how to use JAX-RPC to create a Web service named `HelloWorld`. A remote client of the `HelloWorld` service can invoke the `sayHello` method, which accepts a string parameter and then returns a string.

HelloWorld at Runtime

Figure 11–1 shows a simplified view of the `HelloWorld` service after it's been deployed. Here's a more detailed description of what happens at runtime:

1. To call a remote procedure, the `HelloClient` program invokes a method on a stub, a local object that represents the remote service.
2. The stub invokes routines in the JAX-RPC runtime system.
3. The runtime system converts the remote method call into a SOAP message and then transmits the message as an HTTP request.
4. When the server receives the HTTP request, the JAX-RPC runtime system extracts the SOAP message from the request and translates it into a method call.
5. The JAX-RPC runtime system invokes the method on the tie object.
6. The tie object invokes the method on the implementation of the `HelloWorld` service.
7. The runtime system on the server converts the method's response into a SOAP message and then transmits the message back to the client as an HTTP response.
8. On the client, the JAX-RPC runtime system extracts the SOAP message from the HTTP response and then translates it into a method response for the `HelloClient` program.

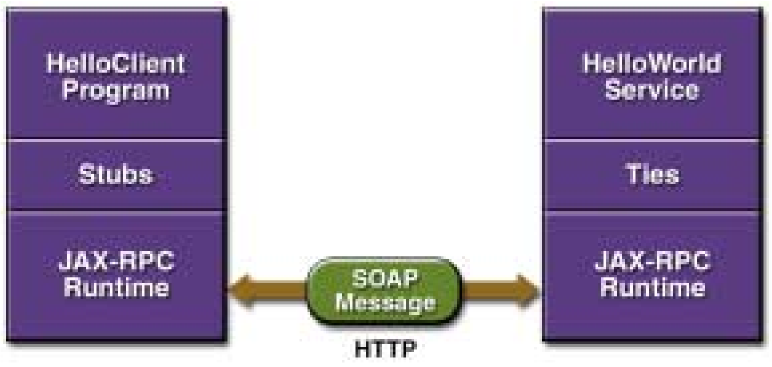


Figure 11–1 The `HelloWorld` Example at Runtime

The application developer only provides the top layers in the stacks depicted by Figure 11–1. Table 11–1 shows where the layers originate.

Table 11–1 Who (or What) Provides the Layers

Layer	**Source**
`HelloClient` Program `HelloWorld` Service (definition interface and implementation class)	Provided by the application developer
Stubs	Generated by the `wscompile` tool, which is run by the application developer
Ties	Generated by the `wsdeploy` tool, which is run by the application developer
JAX-RPC Runtime System	Included with the Java WSDP

HelloWorld Files

To create a service with JAX-RPC, an application developer needs to provide a few files. For the `HelloWorld` example, these files are in the *<JWSDP_HOME>*`/docs/tutorial/examples/jaxrpc/hello` directory:

- `HelloIF.java` - the service definition interface
- `HelloImpl.java` - the service definition implementation class, it implements the `HelloIF` interface
- `HelloClient.java` - the remote client that contacts the service and then invokes the `sayHello` method
- `config.xml` - a configuration file read by the `wscompile` tool
- `jaxrpc-ri.xml` - a configuration file read by the `wsdeploy` tool
- `web.xml` - a deployment descriptor for the Web component (a servlet) that dispatches to the service

Setting Up

First, you must set the PATH environment variable so that it includes these directories:

```
<JWSDP_HOME>/bin
<JWSDP_HOME>/jwsdp-shared/bin
<JWSDP_HOME>/jaxrpc-1.0.3/bin
<JWSDP_HOME>/jakarta-ant-1.5.1/bin
```

Next, if you haven't already done so, follow these instructions in the chapter Getting Started With Tomcat:

- Creating the Build Properties File (page 71)
- Starting Tomcat (page 80)

Building and Deploying the Service

The basic steps for developing a JAX-RPC Web service are as follows.

1. Code the service definition interface and implementation class.
2. Compile the service definition code of step 1.
3. Package the code in a WAR file.

4. Generate the ties and the WSDL file.
5. Deploy the service.

The sections that follow describe each of these steps in more detail.

Coding the Service Definition Interface and Implementation Class

A service definition interface declares the methods that a remote client may invoke on the service. The interface must conform to a few rules:

- It extends the `java.rmi.Remote` interface.
- It must not have constant declarations, such as `public final static`.
- The methods must throw the `java.rmi.RemoteException` or one of its subclasses. (The methods may also throw service-specific exceptions.)
- Method parameters and return types must be supported JAX-RPC types. See the section Types Supported By JAX-RPC (page 467).

In this example, the service definition interface is `HelloIF.java`:

```
package hello;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HelloIF extends Remote {
    public String sayHello(String s) throws RemoteException;
}
```

In addition to the interface, you'll need to code the class that implements the interface. In this example, the implementation class is called `HelloImpl`:

```
package hello;

public class HelloImpl implements HelloIF {

    public String message ="Hello";

    public String sayHello(String s) {
        return message + s;
    }
}
```

Compiling the Service Definition Code

To compile `HelloIF.java` and `HelloImpl.java`, go to the `<JWSDP_HOME>/docs/tutorial/examples/jaxrpc/hello` directory and type the following:

```
ant compile-server
```

This command places the resulting class files in the `build/shared` subdirectory.

Packaging the WAR File

To create the WAR file that contains the service code, type these commands:

```
ant setup-web-inf
ant package
```

The `setup-web-inf` target copies the class and XML files to the `build/WEB-INF` subdirectory. The `package` target runs the `jar` command and bundles the files into a WAR file named `dist/hello-portable.war`. This WAR file is not ready for deployment because it does not contain the tie classes. You'll learn how to create a deployable WAR file in the next section. The `hello-portable.war` contains the following files:

```
WEB-INF/classes/hello/HelloIF.class
WEB-INF/classes/hello/HelloImpl.class
WEB-INF/jaxrpc-ri.xml
WEB-INF/web.xml
```

The class files were created by the `compile-server` target shown in the previous section. The `web.xml` file is the deployment descriptor for the Web application that implements the service. Unlike the `web.xml` file, the `jaxrpc-ri.xml` file is not part of the specifications and is implementation-specific. The `jaxrpc-ri.xml` file for this example follows:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<webServices
    xmlns=”http://java.sun.com/xml/ns/jax-rpc/ri/dd”
    version=”1.0”
    targetNamespaceBase=”http://com.test/wsdl”
    typeNamespaceBase=”http://com.test/types”
    urlPatternBase=”/ws”>
```

```
        <endpoint
            name=”MyHello”
            displayName=”HelloWorld Service”
            description=”A simple web service”
            interface=”hello.HelloIF”
            implementation=”hello.HelloImpl”/>

        <endpointMapping
            endpointName=”MyHello”
            urlPattern=”/hello”/>

    </webServices>
```

Several of the `webServices` attributes, such as `targetNamespaceBase`, are used in the WSDL file, which you'll create in the next section. (WSDL files can be complex and are not discussed in this tutorial. See Further Information, page 481.) Note that the `urlPattern` value (`/hello`) is part of the service's URL, which is described in the section Verifying the Deployment, page 462).

For more information about the the `jaxrpc-ri.xml` file, see the section, The jaxrpc-ri.xml File (page 478). If you are an advanced user, you may want to examine the XML Schema file: *<JWSDP_HOME>*`/docs/tutorial/examples/jaxrpc/common/jax-rpc-ri-dd.xsd`.

Generating the Ties and the WSDL File

To generate the ties and the WSDL file, type the following:

```
ant process-war
```

This command runs the `wsdeploy` tool as follows:

```
wsdeploy -tmpdir build/wsdeploy-generated
-o dist/hello-deployable.war dist/hello-portable.war
```

This command runs the `wsdeploy` tool, which performs these tasks:

- Reads the `dist/hello-portable.war` file as input
- Gets information from the `jaxrpc-ri.xml` file that's inside the `hello-portable.war` file
- Generates the tie classes for the service
- Generates a WSDL file named `MyHello.wsdl`

- Packages the tie classes, the `Hello.wsdl` file, and the contents of `hello-portable.war` file into a deployable WAR file named `dist/hello-jaxrpc.war`

The `-tmpdir` option specifies the directory where `wsdeploy` stores the files that it generates, including the WSDL file, tie classes, and intermediate source code files. If you specify the `-keep` option, these files are not deleted.

There are several ways to access the WSDL file generated by `wsdeploy`:

- Run `wsdeploy` with the `-keep` option and locate the WSDL file in the directory specified by the `-tmpdir` option.
- Unpack (`jar -x`) the WAR file output by `wsdeploy` and locate the WSDL file in the `WEB-INF` directory.
- Deploy and verify the service as described in the following sections. A link to the WSDL file is on the HTML page of the URL shown in Verifying the Deployment (page 462).

Note that the `wsdeploy` tool does not deploy the service; instead, it creates a WAR file that is ready for deployment. In the next section, you will deploy the service in the `hello-jaxrpc.war` file that was created by `wsdeploy`.

For more information about `wsdeploy`, see the section, The wsdeploy Tool (page 477).

Deploying the Service

To deploy the service, type the following:

```
ant deploy
```

For subsequent deployments , run `ant redeploy` as described in the section Iterative Development (page 466).

Verifying the Deployment

To verify that the service has been successfully deployed, open a browser window and specify the service endpoint's URL:

```
http://localhost:8080/hello-jaxrpc/hello
```

The browser should display a page titled Web Services, which lists the port name `MyHello` with a status of `ACTIVE`. This page also has a URL to the service's WSDL file.

The `hello-jaxrpc` portion of the URL is the context path of the servlet that implements the `HelloWorld` service. This portion corresponds to the prefix of the `hello-jaxrpc.war` file. The `/hello` string of the URL matches the value of the `urlPattern` attribute of the `jaxrpc-ri.xml` file. Note that the forward slash in the `/hello` value of `urlPattern` is required. For a full listing of the `jaxrpc-ri.xml` file, see Packaging the WAR File (page 460).

Undeploying the Service

At this point in the tutorial, do not undeploy the service. When you are finished with this example, you can undeploy the service by typing this command:

```
ant undeploy
```

Building and Running the Client

To develop a JAX-RPC client, you follow these steps:

1. Generate the stubs.
2. Code the client.
3. Compile the client code.
4. Package the client classes into a JAR file.
5. Run the client.

The following sections describe each of these steps.

Generating the Stubs

Before generating the stubs, be sure to install the `Hello.wsdl` file according to the instructions in Deploying the Service (page 462). To create the stubs, go to the `<JWSDP_HOME>/docs/tutorial/examples/jaxrpc/hello` directory and type the following:

```
ant generate-stubs
```

This command runs the `wscompile` tool as follows:

```
wscompile -gen:client -d build/client
-classpath build/shared config.xml
```

The `-gen:client` option instructs `wscompile` to generate client-side classes such as stubs. The `-d` option specifies the destination directory of the generated files. For more information, see the section, The wscompile Tool (page 474).

The `wscompile` tool generates files based on the information it reads from the `Hello.wsdl` and `config.xml` files. The `Hello.wsdl` file was intalled on Tomcat when the service was deployed. The location of `Hello.wsdl` is specified by the <`wsdl`> element of the `config.xml` file, which follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <wsdl location=
    "http://localhost:8080/hello-jaxrpc/hello?WSDL"
     packageName="hello"/>
</configuration>
```

The tasks performed by the `wscompile` tool depend on the contents of the `config.xml` file. For more information about the `config.xml` file, see the section, Configuration File (page 476). Advanced users may want to examine the XML Schema file: *<JWSDP_HOME>*`/docs/tutorial/examples/jaxrpc/common/jax-rpc-ri-config.xsd`.

Coding the Client

`HelloClient` is a stand-alone program that calls the `sayHello` method of the `HelloWorld` service. It makes this call through a stub, a local object which acts as a proxy for the remote service. Because the stubs is created before runtime (by `wscompile`), it is usually called a *static stub*.

To create the stub, `HelloClient` invokes a private method named `createProxy`. Note that the code in this method is implementation-specific and might not be portable because it relies on the `MyHello_Impl` object. (The `MyHello_Impl` class was generated by `wscompile` in the preceding section.) After it creates the stub, the client program casts the stub to the type `HelloIF`, the service definition interface.

The source code for `HelloClient` follows:

```
package hello;

import javax.xml.rpc.Stub;

public class HelloClient {
    public static void main(String[] args) {
        try {
            Stub stub = createProxy();
            HelloIF hello = (HelloIF)stub;
            System.out.println(hello.sayHello("Duke!"));
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    private static Stub createProxy() {
        // Note: MyHello_Impl is implementation-specific.
        return (Stub)(new MyHello_Impl().getHelloIFPort());
    }
}
```

Compiling the Client Code

Because the client code refers to the stub classes, be sure to follow the instructions in Generating the Stubs (page 463) before compiling the client. To compile the client, go to the `<JWSDP_HOME>/docs/tutorial/examples/jaxrpc/hello` directory and type the following:

```
ant compile-client
```

Packaging the Client

To package the client into a JAR file, type the following command:

```
ant jar-client
```

This command creates the `dist/hello-client.jar` file.

Running the Client

To run the `HelloClient` program, type the following:

```
ant run
```

The program should display this line:

```
Hello Duke!
```

The `ant run` target executes this command:

```
java -classpath <cpath> hello.HelloClient
```

The classpath includes the `hello-client.jar` file that you created in the preceding section, as well as several JAR files that belong to the Java WSDP. In order to run the client remotely, all of these JAR files must reside on the remote client's computer.

Iterative Development

In order to show you each step of development, the previous sections instructed you to type several `ant` commands. However, it would be inconvenient to type all of those commands during iterative development. To save time, after you've initially deployed the service, you can iterate through these steps:

1. Test the application.
2. Edit the source files.
3. Execute `ant build` to create the deployable WAR file.
4. Execute `ant redeploy` to undeploy and deploy the service.
5. Execute `ant build-static` to create the JAR file for a client with static stubs.
6. Execute `ant run`.

Implementation-Specific Features

To implement the JAX-RPC Specification, the Java WSDP requires some features that are not described in the specification. These features are specific to the Java WSDP and might not be compatible with implementations from other vendors. For JAX-RPC, the implementation-specific features of the Java WSDP follow:

- `config.xml` - See Generating the Stubs (page 463) for an example.
- `jaxrpc-ri.xml` - See Packaging the WAR File (page 460) for an example.
- ties - In the preceding example, the ties are in the `hello-jaxrpc.war` file, which is implementation-specific. (The `hello-portable.war` file, however, is not implementation-specific.)
- stubs - The stubs are in the `hello-client.jar` file. Note that the `HelloClient` program instantiates `MyHelloImpl`, a static stub class that is implementation-specific. Because they do not contain static stubs, dynamic clients do not have this limitation. For more information about dynamic clients, see the sections A Dynamic Proxy Client Example (page 470) and A Dynamic Invocation Interface (DII) Client Example (page 471) .
- tools - `wsdeploy` and `wscompile`.
- support for collections - See Table 11–1.

Types Supported By JAX-RPC

Behind the scenes, JAX-RPC maps types of the Java programming language to XML/WSDL definitions. For example, JAX-RPC maps the `java.lang.String` class to the `xsd:string` XML data type. Application developers don't need to know the details of these mappings, but they should be aware that not every class in the Java 2 Platform, Standard Edition (J2SE™ platform) can be used as a method parameter or return type in JAX-RPC.

J2SE SDK Classes

JAX-RPC supports the following J2SE SDK classes:

```
java.lang.Boolean
java.lang.Byte
java.lang.Double
java.lang.Float
java.lang.Integer
java.lang.Long
java.lang.Short
java.lang.String
java.math.BigDecimal
java.math.BigInteger

java.util.Calendar
java.util.Date
```

This release of JAX-RPC also supports several implementation classes of the `java.util.Collection` interface. See Table 11–2.

Table 11–2 Supported Classes of the Java Collections Framework

`java.util.Collection` **Subinterface**	**Implementation Classes**

`List`	`ArrayList`
	`LinkedList`
	`Stack`
	`Vector`

`Map`	`HashMap`
	`Hashtable`
	`Properties`
	`TreeMap`

`Set`	`HashSet` `TreeSet`

Primitives

JAX-RPC supports the following primitive types of the Java programming language:

```
boolean
byte
double
float
int
long
short
```

Arrays

JAX-RPC also supports arrays with members of supported JAX-RPC types. Examples of supported arrays are `int[]` and `String[]`. Multidimensional arrays, such as `BigDecimal[][]`, are also supported.

Application Classes

JAX-RPC also supports classes that you've written for your applications. In an order processing application, for example, you might provide classes named `Order`, `LineItem`, and `Product`. The JAX-RPC Specification refers to such classes as *value types*, because their values (or states) may be passed between clients and remote services as method parameters or return values.

To be supported by JAX-RPC, an application class must conform to the following rules:

- It must have a public default constructor.
- It must not implement (either directly or indirectly) the `java.rmi.Remote` interface.
- Its fields must be supported JAX-RPC types.

The class may contain public, private, or protected fields. For its value to be passed (or returned) during a remote call, a field must meet these requirements:

- A public field cannot be final or transient.
- A non-public field must have corresponding getter and setter methods.

JavaBeans Components

JAX-RPC also supports JavaBeans components, which must conform to the same set of rules as application classes. In addition, a JavaBeans component must have a getter and setter method for each bean property. The type of the bean property must be a supported JAX-RPC type. For an example of a JavaBeans component, see the section JAX-RPC Distributor Service (page 749).

A Dynamic Proxy Client Example

The client in the section, A Simple Example: HelloWorld (page 456), used a static stub for the proxy. In contrast, the client example in this section calls a remote procedure through a *dynamic proxy*, a class that is created during runtime. Before creating the proxy class, the client gets information about the service by looking up its WSDL document.

Dynamic Proxy HelloClient Listing

Here is the full listing for the `HelloClient.java` file of the *<JWSDP_HOME>*`/docs/tutorial/examples/jaxrpc/proxy` directory.

```
package proxy;

import java.net.URL;
import javax.xml.rpc.Service;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceFactory;

public class HelloClient {

    public static void main(String[] args) {
        try {

            String UrlString =
                "http://localhost:8080/ProxyHelloWorld.wsdl";
            String nameSpaceUri = "http://proxy.org/wsdl";
            String serviceName = "HelloWorld";
            String portName = "HelloIFPort";

            URL helloWsdlUrl = new URL(UrlString);

            ServiceFactory serviceFactory =
                ServiceFactory.newInstance();

            Service helloService =
                serviceFactory.createService(helloWsdlUrl,
                new QName(nameSpaceUri, serviceName));

            HelloIF myProxy = (HelloIF) helloService.getPort(
                new QName(nameSpaceUri, portName),
                proxy.HelloIF.class);
                System.out.println(myProxy.sayHello("Buzz"));

            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
```

Building and Running the Dynamic Proxy Example

Perform the following steps:

1. If you haven't already done so, follow the instructions in Setting Up (page 458).
2. Go to the *<JWSDP_HOME>*/docs/tutorial/examples/jaxrpc/proxy directory.
3. Type the following commands:

```
ant build
ant deploy
ant build-dynamic
ant run
```

The client should display the following line:

```
A dynamic proxy hello to Buzz!
```

A Dynamic Invocation Interface (DII) Client Example

With the dynamic invocation interface (DII), a client can call a remote procedure even if the signature of the remote procedure or the name of the service are unknown until runtime.

Because of its flexibility, a DII client can be used in a service broker that dynamically discovers services, configures the remote calls, and executes the calls. For example, an application for an online clothing store might access a service broker that specializes in shipping. This broker would use the Java API for XML

Registries (JAXR) to locate the services of the shipping companies that meet certain criteria, such as low cost or fast delivery time. At runtime, the broker uses DII to call remote procedures on the Web services of the shipping companies. As an intermediary between the clothing store and the shipping companies, the broker offers benefits to all parties. For the clothing store, it simplifies the shipping process, and for the shipping companies, it finds customers.

DII HelloClient Listing

Here is the full listing for the `HelloClient.java` file of the *<JWSDP_HOME>*`/docs/tutorial/examples/jaxrpc/dynamic` directory.

```
package dynamic;

import javax.xml.rpc.Call;
import javax.xml.rpc.Service;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.ParameterMode;

public class HelloClient {

    private static String endpoint =
        "http://localhost:8080/dynamic-jaxrpc/dynamic";
    private static String qnameService = “Hello”;
    private static String qnamePort = “HelloIF”;

    private static String BODY_NAMESPACE_VALUE =
        “http://dynamic.org/wsdl”;
    private static String ENCODING_STYLE_PROPERTY =
        “javax.xml.rpc.encodingstyle.namespace.uri”;
    private static String NS_XSD =
        “http://www.w3.org/2001/XMLSchema”;
    private static String URI_ENCODING =
         “http://schemas.xmlsoap.org/soap/encoding/”;

    public static void main(String[] args) {
        try {

            ServiceFactory factory =
                ServiceFactory.newInstance();
            Service service =
               factory.createService(new QName(qnameService));
```

```
                QName port = new QName(qnamePort);

                Call call = service.createCall(port);
                call.setTargetEndpointAddress(endpoint);

                call.setProperty(Call.SOAPACTION_USE_PROPERTY,
                    new Boolean(true));
                call.setProperty(Call.SOAPACTION_URI_PROPERTY,““);
                call.setProperty(ENCODING_STYLE_PROPERTY,
                   URI_ENCODING);
                QName QNAME_TYPE_STRING =
                    new QName(NS_XSD, “string”);
                call.setReturnType(QNAME_TYPE_STRING);


                call.setOperationName(
                    new QName(BODY_NAMESPACE_VALUE “sayHello”));
                call.addParameter(“String_1”, QNAME_TYPE_STRING,
                    ParameterMode.IN);
                String[] params = { “Duke!” };

                String result = (String)call.invoke(params);
                System.out.println(result);

            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
```

Building and Running the DII Example

Perform the following steps:

1. If you haven’t already done so, follow the instructions in Setting Up (page 458).
2. Go to the *<JWSDP_HOME>*`/docs/tutorial/examples/jaxrpc/dynamic` directory.
3. Type the following commands:

```
ant build
ant deploy
ant build-dynamic
ant run
```

The client should display the following line:

```
A dynamic hello to Duke!
```

The wscompile Tool

The `wscompile` tool generates stubs, ties, serializers, and WSDL files used in JAX-RPC clients and services. The tool reads as input a configuration file and either a WSDL file or an RMI interface that defines the service.

Syntax

```
wscompile [options] <configuration-file>
```

By convention, the configuration file is named config.xml, but this is not a requirement. The following table lists the `wscompile` options. Note that exactly one of the -import, -define, or -gen options must be specified.

Table 11–3 `wscompile` Options

Option	**Description**
`-classpath <path>`	specify where to find input class files; on Windows, the pathnames should be enclosed in quotes, for example: -classpath “\test;\foo;\acct”
`-cp <path>`	same as -classpath <path>
`-d <directory>`	specify where to place generated output files
`-define`	read the service's RMI interface, define a service
`-f:<features>`	enable the given features (See the below below table for a list of features. When specifying multiple features, separate them with commas.)
`-features:<features>`	same as -f:<features>
`-g`	generate debugging info
`-gen`	same as -gen:client

Table 11–3 `wscompile` Options

Option	Description
`-gen:client`	generate client artifacts (stubs, etc.)
`-gen:server`	generate server artifacts (ties, etc.) and the WSDL file (If you are using `wsdeploy` you do not specify this option.)
`-gen:both`	generate both client and server artifacts
`-http-proxy:<host>:<port>`	specify a HTTP proxy server (port defaults to 8080)
`-import`	read a WSDL file, generate the service's RMI interface and a template of the class that implements the interface
`-keep`	keep generated files
`-model <file>`	write the internal model to the given file
`-nd <directory>`	specify where to place non-class generated files
`-O`	optimize generated code
`-s <directory>`	specify where to place generated source files
`-verbose`	output messages about what the compiler is doing
`-version`	print version information

The following table lists the features (delimited by commas) that may follow the -f option.

Table 11–4 `wscompile` -f Features

Feature	Description
`datahandleronly`	always map attachments to the DataHandler type
`explicitcontext`	turn on explicit service context mapping
`infix=<name>`	specify an infix to use for generated serializers
`nodatabinding`	turn off data binding for literal encoding

Table 11–4 `wscompile` -f Features

Feature	Description
`noencodedtypes`	turn off encoding type information
`nomultirefs`	turn off support for multiple references
`novalidation`	turn off full validation of imported WSDL documents
`searchschema`	search schema aggressively for subtypes
`serializeinterfaces`	turn on direct serialization of interface types

Configuration File

The `wscompile` tool reads the configuration file (config.xml), which contains information that describes the web service. The basic structure of config.xml follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
        <service> or <wsdl> or <modelfile>
</configuration>
```

The <configuration> element may contain exactly one <service>, <wsdl>, or <modelfile> element.

The <service> Element

If you specify this element, `wscompile` reads the RMI interface that describes the service and generates a WSDL file. In the <interface> subelement, the name attribute specifies the service's RMI interface, and the servantName attribute specifies the class that implements the interface. For example:

```
<service name="CollectionIF_Service"

targetNamespace="http://echoservice.org/wsdl"
      typeNamespace="http://echoservice.org/types"
      packageName=”stub_tie_generator_test”>
    <interface name=”stub_tie_generator_test.CollectionIF”
       servantName=”stub_tie_generator_test.CollectionImpl”/>
</service>
```

The <wsdl> Element

If you specify this element, `wscompile` reads the service’s WSDL file and generates the service’s RMI interface. The location attribute specifies the URL of the WSDL file, and the packageName attribute specifies the package of the classes generated by `wscompile`. For example:

```
<wsdl
    location="http://tempuri.org/sample.wsdl"
    packageName="org.tempuri.sample" />
```

The <modelfile> Element

This element is for advanced users.

If `config.xml` contains a <service> or <wsdl> element, `wscompile` generates a model file that contains the internal data structures that describe the service. If you’ve already generated a model file in this manner, then you can reuse it the next time you run `wscompile`. For example:

```
<modelfile location=”mymodel.xml.gz”/>
```

The wsdeploy Tool

The `wsdeploy` tool reads a WAR file and the `jaxrpc-ri.xml` file and then generates another WAR file that is ready for deployment. Behind the scenes, `wsdeploy` runs `wscompile` with the `-gen:server` option. The `wscompile` command generates classes and a WSDL file which `wsdeploy` includes in the generated WAR file.

Syntax

The syntax for `wsdeploy` follows:

```
wsdeploy <options> <input-war-file>
```

The following table lists the tool's options. Note that the `-o option` is required.

Table 11–5 `wsdeploy` Options

`-classpath <path>`	specify an optional classpath
`-keep`	keep temporary files
`-o <output-war-file>`	specify where to place the generated war file
`-tmpdir <directory>`	specify the temporary directory to use
`-verbose`	output messages about what the compiler is doing
`-version`	print version information

The Input WAR File

Typically, you create the input WAR file with a GUI development tool or with the ant war task. Here are the contents of a simple input WAR file:

```
META-INF/MANIFEST.MF
WEB-INF/classes/hello/HelloIF.class
WEB-INF/classes/hello/HelloImpl.class
WEB-INF/jaxrpc-ri.xml
WEB-INF/web.xml
```

In this example, HelloIF is the service's RMI interface and HelloImpl is the class that implements the interface. The web.xml file is the deployment descriptor of a web component. The jaxrpc-ri.xml file is described in the next section.

The jaxrpc-ri.xml File

The listing that follows shows a `jaxrpc-ri.xml` file for a simple HelloWorld service.

The <webServices> element must contain one or more <endpoint> elements. In this example, note that the interface and implementation attributes of <endpoint> specify the service's interface and implementation class. The <endpointMap-

ping> element associates the service port with the part of the endpoint URL path that follows the urlPatternBase.

```
<?xml version="1.0" encoding="UTF-8"?>
<webServices
       xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/dd"
       version="1.0"
       targetNamespaceBase="http://com.test/wsdl"
       typeNamespaceBase="http://com.test/types"
       urlPatternBase="/ws">
       <endpoint
           name="MyHello"
           displayName="HelloWorld Service"
           description="A simple web service"
           interface="hello.HelloIF"
           implementation="hello.HelloImpl"/>
       <endpointMapping
           endpointName="MyHello"
           urlPattern="/hello"/>
</webServices>
```

If the service has multiple endpoints, you should specify the port and WSDL for each endpoint. The following `jaxrpc-ri.xml` snippet has multiple endpoints:

```
<endpoint
   name="vendor"
   displayName=")"
   description="Vendor example endpoint"
   interface="com.buzzmurph.vendor.VendorPortType"

implementation="com.buzzmurph.act.vendor.VendorPortTypeImpl"

port="http://buzzmurph.com/preferred/Vendor.wsdl}VendorPort"
   model="/WEB-INF/vendor.xml.gz"
   wsdl="/WEB-INF/VendorImpl.wsdl"/>

<endpoint
   name="supplier"
   displayName=")"
   description="Supplier example endpoint"
   interface="com.buzzmurph.supplier.SupplierPortType"

implementation="com.buzzmurph.supplier.SupplierPortTypeImpl"
   port="http://buzzmurph.com/ord/Supplier.wsdl}SupplierPort"
   model="/WEB-INF/supplier.xml.gz"
   wsdl="/WEB-INF/SupplierImpl.wsdl"/>
```

```
<endpointMapping
   endpointName=”vendor”
   urlPattern=”/act/vendor”/>

<endpointMapping
   endpointName=”supplier”
   urlPattern=”/ord/supplier”/>
```

Advanced Topics for wscompile and wsdeploy

This section is for developers who are familiar with WSDL, SOAP, and the JAX-RPC specifications.

Namespace Mappings

Here is a schema type name example:

```
schemaType="ns1:SampleType"

xmlns:ns1="http://echoservice.org/types"
```

When generating a Java type from a schema type, `wscompile` gets the class name from the local part of the schema type name. To specify the package name of the generated Java classes, you define a mapping between the schema type namespace and the package name. You define this mapping by adding a <namespaceMappingRegistry> element to the `config.xml` file. For example:

```
<service>
    ...
    <namespaceMappingRegistry>
            <namespaceMapping
            namespace="http://echoservice.org/types"
            packageName="echoservice.org.types"/>
     </namespaceMappingRegistry>
    ...
</service>
```

Handlers

A handler accesses a SOAP message that represents an RPC request or response. A handler class must implement the javax.xml.rpc.handler interface. Because it accesses a SOAP message, a handler can manipulate the message with the APIs of the `javax.xml.soap` package.

- Examples of handler tasks:
- Encryption and decryption
- Logging and auditing
- Caching
- Application-specific SOAP header processing

A handler chain is a list of handlers. You may specify one handler chain for the client and one for the server. On the client, you include the <handlerChains> element in the jaxrpc-ri.xml file. On the server, you include this element in the config.xml file. Here is an example of the <handlerChains> element in `config.xml`:

```
<handlerChains>
  <chain runAt="server"
     roles=
      "http://acme.org/auditing
       http://acme.org/morphing"
       xmlns:ns1="http://foo/foo-1">
    <handler className="acme.MyHandler"
      headers ="ns1:foo ns1:bar"/>
      <property
        name="property" value="xyz"/>
    </handler>
  </chain>
</handlerChains>
```

For more information on handlers, see the SOAP Message Handlers chapter of the JAX-RPC specifications.

Further Information

For more information about JAX-RPC and related technologies, refer to the following:

- Java API for XML-based RPC 1.0 Specification `http://java.sun.com/xml/downloads/jaxrpc.html`

- JAX-RPC Home
  `http://java.sun.com/xml/jaxrpc/index.html`
- Simple Object Access Protocol (SOAP) 1.1 W3C Note
  `http://www.w3.org/TR/SOAP/`
- Web Services Description Language (WSDL) 1.1 W3C Note
  `http://www.w3.org/TR/wsdl`

12

Web Services Messaging with JAXM

Maydene Fisher

THE Java API for XML Messaging (JAXM) makes it possible for developers to do XML messaging using the Java platform. By simply making method calls using the JAXM API, you can create and send XML messages over the Internet. This chapter will help you learn how to use the JAXM API.

In addition to stepping you through how to use the JAXM API, this chapter gives instructions for running the sample JAXM applications included with the Java WSDP as a way to help you get started. You may prefer to go through both the overview and tutorial before running the samples to make it easier to understand what the sample applications are doing, or you may prefer to explore the samples first. The overview gives some of the conceptual background behind the JAXM API to help you understand why certain things are done the way they are. The tutorial shows you how to use the basic JAXM API, giving examples and explanations of the more commonly used features. Finally, the code examples in the last part of the tutorial show how to build an application.

The Structure of the JAXM API

The JAXM API conforms to the Simple Object Access Protocol (SOAP) 1.1 specification and the SOAP with Attachments specification. The complete JAXM API is presented in two packages:

- **`javax.xml.soap`** — the package defined in the SOAP with Attachments API for Java (SAAJ) 1.1 specification. This is the basic package for SOAP messaging, which contains the API for creating and populating a SOAP message. This package has all the API necessary for sending request-response messages. (Request-response messages are explained in SOAPConnection, page 489.)

  The current version is SAAJ 1.1.1.

- **`javax.xml.messaging`** — the package defined in the JAXM 1.1 specification. This package contains the API needed for using a messaging provider and thus for being able to send one-way messages. (One-way messages are explained in ProviderConnection, page 490.)

  The current version is JAXM 1.1.1.

Originally, both packages were defined in the JAXM 1.0 specification. The `javax.xml.soap` package was separated out and expanded into the SAAJ 1.1 specification so that now it has no dependencies on the `javax.xml.messaging` package and thus can be used independently. The SAAJ API also makes it easier to create XML fragments, which is especially helpful for developing JAX-RPC implementations.

The `javax.xml.messaging` package, defined in the JAXM 1.1 specification, maintains its dependency on the `java.xml.soap` package because the `soap` package contains the API used for creating and manipulating SOAP messages. In other words, a client sending request-response messages can use just the `javax.xml.soap` API. A Web service or client that uses one-way messaging will need to use API from both the `javax.xml.soap` and `javax.xml.messaging` packages.

Note: In this document, "JAXM 1.1.1 API" refers to the API in the `javax.xml.messaging` package; “SAAJ API” refers to the API in the `javax.xml.soap` package. “JAXM API” is a more generic term, referring to all of the API used for SOAP messaging, that is, the API in both packages.

Overview of JAXM

This overview presents a high-level view of how JAXM messaging works and explains concepts in general terms. Its goal is to give you some terminology and a framework for the explanations and code examples that are presented in the tutorial section.

The overview looks at JAXM from three perspectives:

- Messages
- Connections
- Messaging providers

Messages

JAXM messages follow SOAP standards, which prescribe the format for messages and also specify some things that are required, optional, or not allowed. With the JAXM API, you can create XML messages that conform to the SOAP specifications simply by making Java API calls.

The Structure of an XML Document

Note: For more complete information on XML documents, see Understanding XML (page 41) and Java API for XML Processing (page 115).

An XML document has a hierarchical structure with elements, subelements, subsubelements, and so on. You will notice that many of the SAAJ classes and interfaces represent XML elements in a SOAP message and have the word *element* or *SOAP* or both in their names.

An element is also referred to as a *node*. Accordingly, the SAAJ API has the interface `Node`, which is the base class for all the classes and interfaces that represent XML elements in a SOAP message. There are also methods such as `SOAPElement.addTextNode`, `Node.detachNode`, and `Node.getValue`, which you will see how to use in the tutorial section.

What Is in a Message?

The two main types of SOAP messages are those that have attachments and those that do not.

Messages with No Attachments

The following outline shows the very high-level structure of a SOAP message with no attachments. Except for the SOAP header, all the parts listed are required.

I. SOAP message

  A. SOAP part

    1. SOAP envelope

      a. SOAP header (optional)

      b. SOAP body

The SAAJ API provides the `SOAPMessage` class to represent a SOAP message, `SOAPPart` to represent the SOAP part, `SOAPEnvelope` to represent the SOAP envelope, and so on.

When you create a new `SOAPMessage` object, it will automatically have the parts that are required to be in a SOAP message. In other words, a new `SOAPMessage` object has a `SOAPPart` object that contains a `SOAPEnvelope` object. The `SOAPEnvelope` object in turn automatically contains an empty `SOAPHeader` object followed by an empty `SOAPBody` object. If you do not need the `SOAPHeader` object, which is optional, you can delete it. The rationale for having it automatically included is that more often than not you will need it, so it is more convenient to have it provided.

The `SOAPHeader` object may contain one or more headers with information about the sending and receiving parties and about intermediate destinations for the message. Headers may also do things such as correlate a message to previous messages, specify a level of service, and contain routing and delivery information. The `SOAPBody` object, which always follows the `SOAPHeader` object if there is one, provides a simple way to send mandatory information intended for the ultimate recipient. If there is a `SOAPFault` object (see SOAP Faults, page 516), it must be in the `SOAPBody` object.

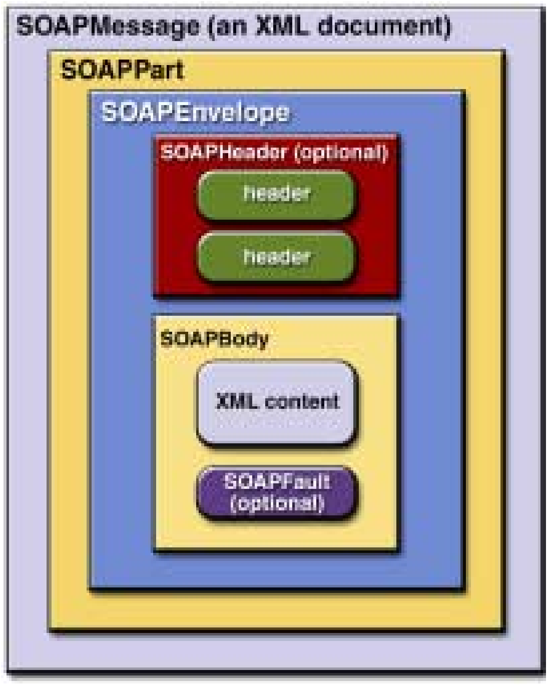


Figure 12–1 `SOAPMessage` Object with No Attachments

Messages with Attachments

A SOAP message may include one or more attachment parts in addition to the SOAP part. The SOAP part may contain only XML content; as a result, if any of the content of a message is not in XML format, it must occur in an attachment part. So, if for example, you want your message to contain an image file or plain text, your message must have an attachment part for it. Note than an attachment part can contain any kind of content, so it can contain data in XML format as well. Figure 12–2 shows the high-level structure of a SOAP message that has two attachments.

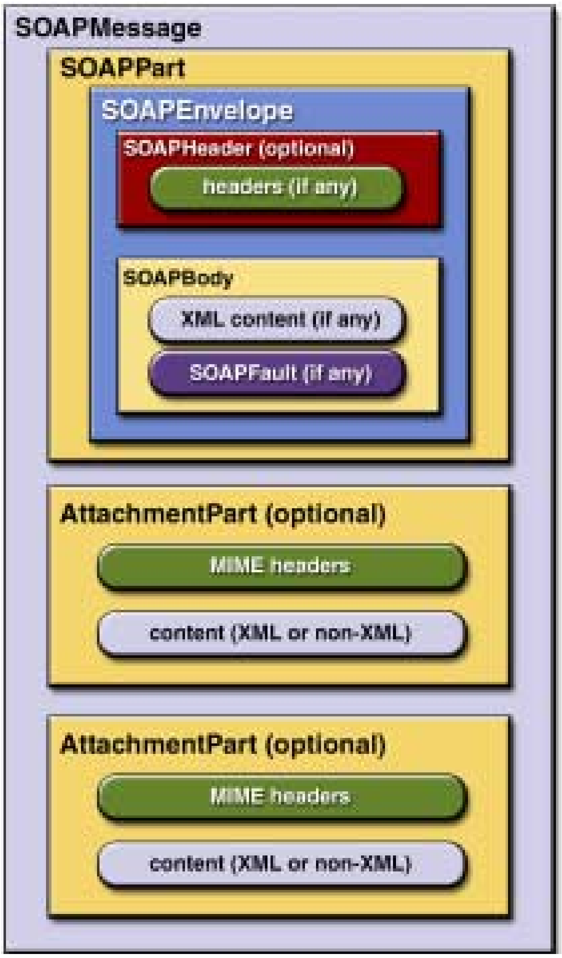


Figure 12–2 `SOAPMessage` Object with Two `AttachmentPart` Objects

The SAAJ API provides the `AttachmentPart` class to represent the attachment part of a SOAP message. A `SOAPMessage` object automatically has a `SOAPPart` object and its required subelements, but because `AttachmentPart` objects are optional, you have to create and add them yourself. The tutorial section will walk you through creating and populating messages with and without attachment parts.

A `SOAPMessage` object may have one or more attachments. Each `AttachmentPart` object has a MIME header to indicate the type of data it contains. It may also have additional MIME headers to identify it or to give its location, which can be useful when there are multiple attachments. When a `SOAPMessage` object has one or more `AttachmentPart` objects, its `SOAPPart` object may or may not contain message content.

Another way to look at SOAP messaging is from the perspective of whether or not a messaging provider is used, which is discussed at the end of the section Messaging Providers (page 492).

Connections

All SOAP messages are sent and received over a connection. The connection can go directly to a particular destination or to a messaging provider. (A messaging provider is a service that handles the transmission and routing of messages and provides features not available when you use a connection that goes directly to its ultimate destination. Messaging providers are explained in more detail later.)

The JAXM API supplies the following class and interface to represent these two kinds of connections:

1. `javax.xml.soap.SOAPConnection` — a connection from the sender directly to the receiver (a point-to-point connection)
2. `javax.xml.messaging.ProviderConnection` — a connection to a messaging provider

SOAPConnection

A `SOAPConnection` object, which represents a point-to-point connection, is simple to create and use. One reason is that you do not have to do any configuration to use a `SOAPConnection` object because it does not need to run in a servlet container (like Tomcat) or in a J2EE container. It is the only kind of connection available to a client that does not use a messaging provider.

The following code fragment creates a `SOAPConnection` object and then, after creating and populating the message, uses the connection to send the message.

The parameter *request* is the message being sent; *endpoint* represents where it is being sent.

```
SOAPConnectionFactory factory =
         SOAPConnectionFactory.newInstance();
SOAPConnection con = factory.createConnection();

. . .// create a request message and give it content

SOAPMessage response = con.call(request, endpoint);
```

When a `SOAPConnection` object is used, the only way to send a message is with the method `call`, which transmits its message and then blocks until it receives a reply. Because the method `call` requires that a response be returned to it, this type of messaging is referred to as *request-response* messaging.

A Web service implemented for request-response messaging must return a response to any message it receives. When the message is an update, the response is an acknowledgement that the update was received. Such an acknowledgement implies that the update was successful. Some messages may not require any response at all. The service that gets such a message is still required to send back a response because one is needed to unblock the `call` method. In this case, the response is not related to the content of the message; it is simply a message to unblock the `call` method.

Because the signature for the `javax.xml.soap.SOAPConnection.call` method changed in the SAAJ 1.1 specification, a JAXM implementation may elect not to implement the `call` method. To allow for this, there is a new exception on the `SOAPConnectionFactory` class stating that `SOAPConnection` is not implemented, which allows for a graceful failure.

Unlike a client with no messaging provider, which is limited to using only a `SOAPConnection` object, a client that uses a messaging provider is free to use a `SOAPConnection` object or a `ProviderConnection` object. It is expected that `ProviderConnection` objects will be used most of the time.

ProviderConnection

A `ProviderConnection` object represents a connection to a messaging provider. (The next section explains more about messaging providers.) When you send a message via a `ProviderConnection` object, the message goes to the messaging provider. The messaging provider forwards the message, following the message's routing instructions, until the message gets to the ultimate recipient's messaging provider, which in turn forwards the message to the ultimate recipient.

When an application is using a `ProviderConnection` object, it must use the method `ProviderConnection.send` to send a message. This method transmits the message one way and returns immediately, without having to block until it gets a response. The messaging provider that receives the message will forward it to the intended destination and return the response, if any, at a later time. The interval between sending a request and getting the response may be very short, or it may be measured in days. In this style of messaging, the original message is sent as a one-way message, and any response is sent subsequently as a one-way message. Not surprisingly, this style of messaging is referred to as *one-way* messaging.

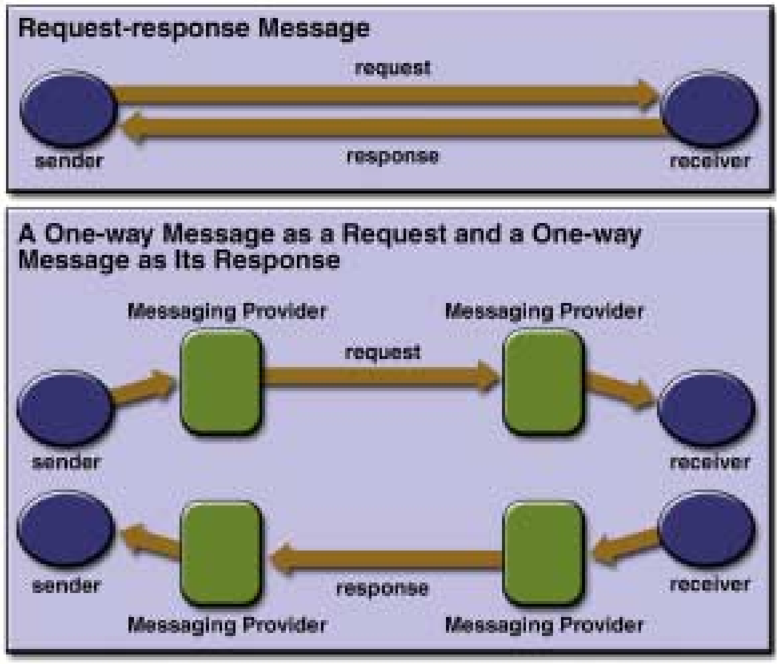


Figure 12–3 Request-response and One-way Messaging

Messaging Providers

A messaging provider is a service that handles the transmission and routing of messages. It works behind the scenes to keep track of messages and see that they are sent to the proper destination or destinations.

Transparency

One of the great features of a messaging provider is that you are not even aware of it. You just write your JAXM application, and the right things happen. For example, when you are using a messaging provider and send a message by calling the `ProviderConnection.send` method, the messaging provider receives the message and works with other parts of the communications infrastructure to perform various tasks, depending on what the message's header contains and how the messaging provider itself has been implemented. The message arrives at its final destination without your even knowing about the details involved in accomplishing the delivery.

Profiles

JAXM offers the ability to plug in additional protocols that are built on top of SOAP. A JAXM provider implementation is not required to implement features beyond what the SOAP 1.1 and SOAP with Attachments specifications require, but it is free to incorporate other standard protocols, called *profiles*, that are implemented on top of SOAP. For example, the "ebXML Message Service Specification (available at `http://www.oasis-open.org/committees/ebxml-msg/`) defines levels of service that are not included in the two SOAP specifications. A messaging provider that is implemented to include ebXML capabilities on top of SOAP capabilities is said to support an ebXML profile. A messaging provider may support multiple profiles, but an application can use only one at a time and must have a prior agreement with each of the parties to whom it sends messages about what profile is being used.

Profiles affect a message's headers. For example, depending on the profile, a new `SOAPMessage` object will come with certain headers already set. Also a profile implementation may provide API that makes it easier to create a header and set its content. The JAXM implementation includes APIs for both the ebXML and SOAP-RP profiles. The API documentation for these profiles is at `<`*`JWSDP_HOME`*`>/jaxm-1.1.1/docs/profiles/index.html`. You will find links to the API documentation for the JAXM API (the javax.xml.soap and javax.xml.messaging packages) at <*JWSDP_HOME*>/docs/api/index.html.

Continuously Active

A messaging provider works continuously. A JAXM client may make a connection with its provider, send one or more messages, and then close the connection. The provider will store the message and then send it. Depending on how the provider has been configured, it will resend a message that was not successfully delivered until it is successfully delivered or until the limit for the number of resends is reached. Also, the provider will stay in a waiting state, ready to receive any messages that are intended for the client. The provider will store incoming messages so that when the client connects with the provider again, the provider will be able to forward the messages. In addition, the provider generates error messages as needed and maintains a log where messages and their related error messages are stored.

Intermediate Destinations

When a messaging provider is used, a message can be sent to one or more intermediate destinations before going to the final recipient. These intermediate destinations, called *actors*, are specified in the message's SOAPHeader object. For example, assume that a message is an incoming Purchase Order. The header might route the message to the order input desk, the order confirmation desk, the shipping desk, and the billing department. Each of these destinations is an actor that will take the appropriate action, remove the header information relevant to it, and send the message to the next actor. The default actor is the final destination, so if no actors are specified, the message is routed to the final recipient.

The attribute actor is used to specify an intermediate recipient. A related attribute is mustUnderstand, which, when its value is true, means that an actor must understand what it is supposed to do and carry it out successfully. A SOAPHeader object uses the method addAttribute to add these attributes, and the SOAPHeaderElement interface provides methods for setting and getting the values of these attributes.

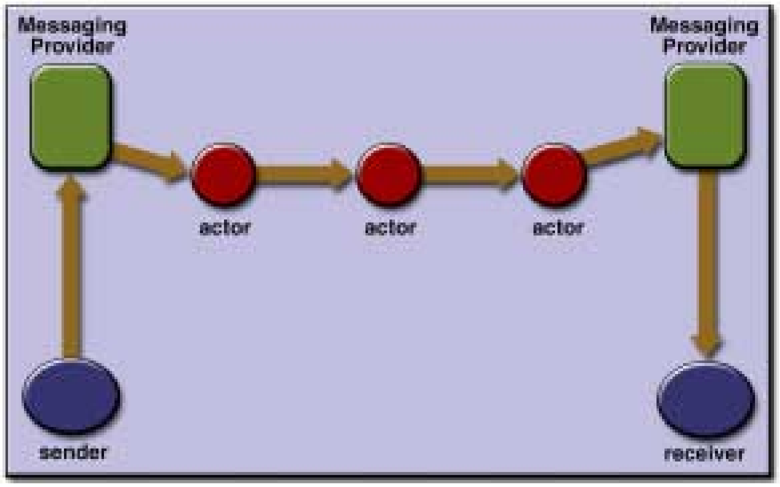


Figure 12–4 One-way Message with Intermediate Destinations

When to Use a Messaging Provider

A JAXM client may or may not use a messaging provider. Generally speaking, if you just want to be a consumer of Web services, you do not need a messaging provider. The following list shows some of the advantages of not using a messaging provider:

- The application can be written using the J2SE platform
- The application is not required to be deployed in a servlet container or a J2EE container
- No configuration is required

The limitations of not using a messaging provider are the following:

- The client can send only request-response messages
- The client can act in the client role only

It follows that if you want to provide a Web service that is able to get and save requests that are sent to you at any time, you must use a messaging provider. You will also need to run in a container, which provides the messaging infrastructure used by the provider. A messaging provider gives you the flexibility to assume both the client and service roles, and it also lets you send one-way messages. In addition, if your messaging provider supports a protocol such as ebXML or

SOAP-RP on top of SOAP, you can take advantage of the additional quality of service features that it provides.

Messaging with and without a Provider

JAXM clients can be categorized according to whether or not they use a messaging provider. Those that do not use a messaging provider can be further divided into those that run in a container and those that do not. A JAXM client that does not use a messaging provider and also does not run in a container is called a *standalone* client.

Running the Samples

The Java WSDP includes several JAXM sample applications. It also includes various implementations that make it possible for you to run the sample applications. These implementations, which constitute the JAXM implementation, are the following:

- An implementation of the JAXM API
- An implementation of a messaging provider
- Basic implementations of ebXML and SOAP-RP profiles, which run on top of SOAP

All of the sample applications use the JAXM API, of course, and some use other implementations as well. For example, the sample application Remote uses the implementations of the messaging provider and the ebXML profile; the SOAP-RP sample uses the implementations for the messaging provider and the SOAP-RP profile. The next section (The Sample Programs, page 496) gives more information about the sample applications and what they do.

Most of the samples run in a container, so before running them, you need to start Tomcat (see Starting Tomcat, page 80).

Once Tomcat is running, you can run the JAXM samples by following these steps:

1. Open a browser window and set it to

```
http://localhost:8080/index.html
```

2. On the page that comes up, click on one of the sample programs listed. Then follow the instructions in the new window that comes up.

The Sample Programs

The sample programs illustrate various kinds of applications you can write with the JAXM API. Note that the Simple, Translator, and SAAJ Simple examples log messages sent and received to the directory in your Java WSDP installation where you started Tomcat. So if, for example, you start Tomcat from the `<JWSDP_HOME>/bin` directory, that is where the messages will be logged. These messages are the XML that is sent over the wire, which you might find easier to understand after you have gone through the tutorial.

- Simple — A simple example of sending and receiving a message using the local provider. Note that a *local provider* should not be confused with a messaging provider. The local provider is simply a mechanism for returning the reply to a message that was sent using the method `SOAPConnection.call`. Note that a message sent by this method will always be a request-response message. Running this example generates the files `sent.msg` and `reply.msg`, which you will find in the directory where you started Tomcat.
- SAAJ Simple — An application similar to the Simple example except that it is written using only the SAAJ API. In SAAJ Simple, the `call` method takes a Java `Object` rather than a `URLEndpoint` object to designate the recipient, and thus uses only the `javax.xml.soap` package. Running this example generates the files `sent.msg` and `reply.msg`, which you will find in the directory where you started Tomcat.
- Translator — An application that uses a simple translation service to translate a given word into different languages. If you have given the correct proxy host and proxy port, the word you supply will be translated into French, German, and Italian. Running this example generates the files `request.msg` and `reply.msg` in the directory where you started Tomcat. Check `reply.msg` after getting the reply in the SOAP body and again after

getting the reply as an attachment to see the difference in what is sent as a reply.

- JAXM Tags — An example that uses JavaServer Pages tags to generate and consume a SOAP message
- Remote — An example of a round trip message that uses a JAXM messaging provider that supports the basic ebXML profile to send and receive a message
- SOAP-RP — An example of a round trip message that uses a JAXM messaging provider that supports the basic SOAP-RP profile to send and receive a message

There are two other sample programs, `jaxm-uddiping` and `jaxm-standalone`, that do not run in Tomcat. To run them, go to the *<JWSDP_HOME>*`/jaxm-1.1.1/samples` directory, where you will find the directories `uddiping` and `standalone`. Each directory contains a `README` file that explains what to do.

In the Examples section of the JAXM tutorial (UddiPing.java and MyUddiPing.java, page 523), you will find an application that modifies the code in `UddiPing.java` and also explains in detail how to run it. You might find it more convenient to wait until you have reached that section before trying to run the `jaxm-uddiping` and `jaxm-standalone` samples.

The preceding list presented the sample applications according to what they do. You can also look at the sample applications as examples of the three possible types of JAXM client:

- **Those that do not use a messaging provider and also do not run in a container**

  These are called *standalone* applications. The samples `jaxm-standalone` and `jaxm-uddiping` are examples of standalone clients.

- **Those that do not use a messaging provider and run in a container**

  The samples Simple, SAAJ Simple, Translator, and JAXM Tags are examples of this type.

- **Those that use a messaging provider and run in a container**

  The samples Remote and SOAP-RP are examples of this type.

Source Code for the Samples

Source code for the sample applications is in the directory

```
<JWSDP_HOME>/docs/tutorial/examples/jaxm/samples/
```

You will find six directories, one for each of the samples that runs in Tomcat. The `jaxmtags` directory contain a number of `.jsp` files. The other directories all have two files, `SendingServlet.java` and `ReceivingServlet.java`. In addition to those two files, the `translator` directory contains the file `TranslationService.java`.

If you want to see all of the files that make up a Web application, you can go to the directory `<JWSDP_HOME>/jaxm-1.1.1/webapps` and unpack the `.war` files. For example, for the Simple sample, you would do the following:

```
cd <JWSDP_HOME>/jaxm-1.1.1/webapps
jar -xvf jaxm-simple.war
```

In addition to the source files and class files for the Simple sample, you will find the files `web.xml` and `build.xml`. .

The `web.xml` file, referred to as a deployment descriptor, associates the endpoint passed to the method `SOAPConnection.call` or `ProviderConnection.send` with a particular servlet class. When the container encounters an endpoint, which is generally a URI, it uses the `web.xml` file to determine the appropriate servlet class and runs it. See the end of the section Sending the Request (page 759) for an example and explanation.

The `build.xml` file is the `Ant` file to use to run the application.

Tutorial

This section will walk you through the basics of sending a SOAP message using the JAXM API. At the end of this chapter, you will know how to do the following:

- Get a connection
- Create a message
- Add content to a message
- Send a message
- Retrieve the content from a response message
- Create and retrieve a SOAP fault element

First, we'll walk through the steps in sending a request-response message for a client that does not use a messaging provider. Then we'll do a walkthrough of a client that uses a messaging provider sending a one-way message. Both types of client may add attachments to a message, so adding attachments is covered as a separate topic. Finally, we'll see what SOAP faults are and how they work.

The section Code Examples (page 521) puts the code fragments you will produce into runnable applications, which you can test yourself. The JAXM part of the case study (JAXM Distributor Service, page 758) demonstrates how JAXM code can be used in a Web service, showing both the client and server code.

Client without a Messaging Provider

An application that does not use a messaging provider is limited to operating in a client role and can send only request-response messages. Though limited, it can make use of Web services that are implemented to do request-response messaging.

Getting a SOAPConnection Object

The first thing any JAXM client needs to do is get a connection, either a `SOAP-Connection` object or a `ProviderConnection` object. The overview section (Connections, page 489) discusses these two types of connections and how they are used.

A client that does not use a messaging provider has only one choice for creating a connection, which is to create a `SOAPConnection` object. This kind of connection is a point-to-point connection, meaning that it goes directly from the sender to the destination (usually a URL) that the sender specifies.

The first step is to obtain a SOAPConnectionFactory object that you can use to create your connection. The SAAJ API makes this easy by providing the SOAP-ConnectionFactory class with a default implementation. You can get an instance of this implementation with the following line of code.

```
SOAPConnectionFactory scFactory =
                    SOAPConnectionFactory.newInstance();
```

Notice that because newInstance is a static method, you will always use the class name SOAPConnectionFactory when you invoke its newInstance method.

Now you can use scFactory to create a SOAPConnection object.

```
SOAPConnection con = scFactory.createConnection();
```

You will use con later to send the message that is created in the next part.

Creating a Message

The next step is to create a message, which you do using a MessageFactory object. If you are a standalone client, you can use the default implementation of the MessageFactory class that the SAAJ API provides. The following code fragment illustrates getting an instance of this default message factory and then using it to create a message.

```
MessageFactory factory = MessageFactory.newInstance();
SOAPMessage message = factory.createMessage();
```

As is true of the newInstance method for SOAPConnectionFactory, the newInstance method for MessageFactory is static, so you invoke it by calling MessageFactory.newInstance. Note that it is possible to write your own implementation of a message factory and plug it in via system properties, but the default message factory is the one that will generally be used.

The other way to get a MessageFactory object is to retrieve it from a naming service where it has been registered. This way is available only to applications that use a messaging provider, and it will be covered later (in Creating a Message, page 508).

Parts of a Message

A `SOAPMessage` object is required to have certain elements, and the SAAJ API simplifies things for you by returning a new `SOAPMessage` object that already contains these elements. So `message`, which was created in the preceding line of code, automatically has the following:

I. A `SOAPPart` object that contains

   A. A `SOAPEnvelope` object that contains

      1. An empty `SOAPHeader` object

      2. An empty `SOAPBody` object

The `SOAPHeader` object, though optional, is included for convenience because most messages will use it. The `SOAPBody` object can hold the content of the message and can also contain fault messages that contain status information or details about a problem with the message. The section SOAP Faults (page 516) walks you through how to use `SOAPFault` objects.

Accessing Elements of a Message

The next step in creating a message is to access its parts so that content can be added. The `SOAPMessage` object `message`, created in the previous code fragment, is where to start. It contains a `SOAPPart` object, so you use `message` to retrieve it.

```
SOAPPart soapPart = message.getSOAPPart();
```

Next you can use `soapPart` to retrieve the `SOAPEnvelope` object that it contains.

```
SOAPEnvelope envelope = soapPart.getEnvelope();
```

You can now use `envelope` to retrieve its empty `SOAPHeader` and `SOAPBody` objects.

```
SOAPHeader header = envelope.getHeader();
SOAPBody body = envelope.getBody();
```

Our example of a standalone client does not use a SOAP header, so you can delete it. Because all `SOAPElement` objects, including `SOAPHeader` objects, are derived from the `Node` interface, you use the method `Node.detachNode` to delete `header`.

```
header.detachNode();
```

Adding Content to the Body

To add content to the body, you need to create a `SOAPBodyElement` object to hold the content. When you create any new element, you also need to create an associated `Name` object to identify it. One way to create `Name` objects is by using `SOAPEnvelope` methods, so you can use `envelope` from the previous code fragment to create the `Name` object for your new element.

Note: The SAAJ API augments the `javax.xml.soap` package by adding the `SOAPFactory` class, which lets you create `Name` objects without using a `SOAPEnvelope` object. This capability is useful for creating XML elements when you are not creating an entire message. For example, JAX-RPC implementations find this ability useful. When you are not working with a `SOAPMessage` object, you do not have access to a `SOAPEnvelope` object and thus need an alternate means of creating `Name` objects. In addition to a method for creating `Name` objects, the `SOAPFactory` class provides methods for creating `Detail` objects and SOAP fragments. You will find an explanation of `Detail` objects in the SOAP Fault sections Overview (page 516) and Creating and Populating a SOAPFault Object (page 518).

`Name` objects associated with `SOAPBody` and `SOAPHeader` objects must be fully qualified; that is, they must be created with a local name, a prefix for the namespace being used, and a URI for the namespace. Specifying a namespace for an element makes clear which one is meant if there is more than one element with the same local name.

The code fragment that follows retrieves the `SOAPBody` object `body` from `envelope`, creates a `Name` object for the element to be added, and adds a new `SOAPBodyElement` object to `body`.

```
SOAPBody body = envelope.getBody();
Name bodyName = envelope.createName("GetLastTradePrice",
            "m", "http://wombat.ztrade.com");
SOAPBodyElement gltp = body.addBodyElement(bodyName);
```

At this point, `body` contains a `SOAPBodyElement` object identified by the `Name` object `bodyName`, but there is still no content in `gltp`. Assuming that you want to get a quote for the stock of Sun Microsystems, Inc., you need to create a child element for the symbol using the method `addChildElement`. Then you need to give it the stock symbol using the method `addTextNode`. The `Name` object for the new SOAPElement object symbol is initialized with only a local name, which is allowed for child elements.

```
Name name = envelope.createName("symbol");
SOAPElement symbol = gltp.addChildElement(name);
symbol.addTextNode("SUNW");
```

You might recall that the headers and content in a SOAPPart object must be in XML format. The JAXM API takes care of this for you, building the appropriate XML constructs automatically when you call methods such as addBodyElement, addChildElement, and addTextNode. Note that you can call the method addTextNode only on an element such as bodyElement or any child elements that are added to it. You cannot call addTextNode on a SOAPHeader or SOAPBody object because they contain elements, not text.

The content that you have just added to your SOAPBody object will look like the following when it is sent over the wire:

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  <SOAP-ENV:Body>
     <m:GetLastTradePrice xmlns:m=
                 "http://wombat.ztrade.com">
        <symbol>SUNW</symbol>
     </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Let's examine this XML excerpt line by line to see how it relates to your JAXM code. Note that an XML parser does not care about indentations, but they are generally used to indicate element levels and thereby make it easier for a human reader to understand.

JAXM code:

```
SOAPPart soapPart = message.getSOAPPart();
SOAPEnvelope envelope = soapPart.getEnvelope();
```

XML it produces:

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  . . . . . . (intervening elements omitted)
</SOAP-ENV:Envelope>
```

The outermost element in this XML example is the SOAP envelope element, indicated by `SOAP-ENV:Envelope`. `Envelope` is the name of the element, and `SOAP-ENV` is the namespace prefix. The interface `SOAPEnvelope` represents a SOAP envelope.

The first line signals the beginning of the SOAP envelope element, and the last line signals the end of it; everything in between is part of the SOAP envelope. The second line has an attribute for the SOAP envelope element. `xmlns` stands for “XML namespace,” and its value is the URI of the namespace associated with `Envelope`. This attribute is automatically included for you.

JAXM code:

```
SOAPBody body = envelope.getBody();
```

XML it produces:

```
<SOAP-ENV:Body>
     . . . . . .
</SOAP-ENV:Body>
```

These two lines mark the beginning and end of the SOAP body, represented in JAXM by a `SOAPBody` object.

JAXM code:

```
Name bodyName = envelope.createName("GetLastTradePrice",
            "m", "http://wombat.ztrade.com");
SOAPBodyElement gltp = body.addBodyElement(bodyName);
```

XML it produces:

```
<m:GetLastTradePrice xmlns:m=
             "http://wombat.ztrade.com">

   . . . .
</m:GetLastTradePrice>
```

These lines are what the `SOAPBodyElement gltp` in your code represents. "GetLastTradePrice" is its local name, "m" is its namespace prefix, and "http://wombat.ztrade.com" is its namespace URI.

JAXM code:

```
Name name = envelope.createName("symbol");
SOAPElement symbol = gltp.addChildElement(name);
symbol.addTextNode("SUNW");
```

XML it produces:

```
   <symbol>SUNW</symbol>
```

The `String` `"SUNW"` is the message content that your recipient, the stock quote service, receives.

Sending a Message

A standalone client uses a `SOAPConnection` object and must therefore use the `SOAPConnection` method `call` to send a message. This method takes two arguments, the message being sent and the destination to which the message should go. This message is going to the stock quote service indicated by the `URL` object `endpoint`.

```
java.net.URL endpoint = new URL(
                        "http://wombat.ztrade.com/quotes”;

SOAPMessage response = con.call(message, endpoint);
```

Your message sent the stock symbol SUNW; the `SOAPMessage` object `response` should contain the last stock price for Sun Microsystems, which you will retrieve in the next section.

A connection uses a fair amount of resources, so it is a good idea to close a connection as soon as you are through using it.

```
con.close();
```

Getting the Content of a Message

The initial steps for retrieving a message’s content are the same as those for giving content to a message: You first access the `SOAPBody` object, using the message to get the envelope and the envelope to get the body. Then you access its `SOAPBodyElement` object because that is the element to which content was added in the example. (In a later section you will see how to add content directly to the

SOAPBody object, in which case you would not need to access the SOAPBodyElement object for adding content or for retrieving it.) To get the content, which was added with the method SOAPElement.addTextNode, you call the method Node.getValue. Note that getValue returns the value of the immediate child of the element that calls the method. Therefore, in the following code fragment, the method getValue is called on bodyElement, the element on which the method addTextNode was called.

In order to access bodyElement, you need to call the method getChildElement on body. Passing bodyName to getChildElement returns a java.util.Iterator object that contains all of the child elements identified by the Name object bodyName. You already know that there is only one, so just calling the method next on it will return the SOAPBodyElement you want. Note that the method Iterator.next returns a Java Object, so it is necessary to cast the Object it returns to a SOAPBodyElement object before assigning it to the variable bodyElement.

```
SOAPPart sp = response.getSOAPPart();
SOAPEnvelope env = sp.getEnvelope();
SOAPBody sb = env.getBody();
java.util.Iterator it = sb.getChildElements(bodyName);
SOAPBodyElement bodyElement = (SOAPBodyElement)it.next();
String lastPrice = bodyElement.getValue();
System.out.print("The last price for SUNW is ");
System.out.println(lastPrice);
```

If there were more than one element with the name bodyName, you would have had to use a while loop using the method Iterator.hasNext to make sure that you got all of them.

```
while (it.hasNext()) {
  SOAPBodyElement bodyElement = (SOAPBodyElement)it.next();
  String lastPrice = bodyElement.getValue();
  System.out.print("The last price for SUNW is ");
  System.out.println(lastPrice);
}
```

At this point, you have seen how to send a request-response message as a standalone client. You have also seen how to get the content from the response. The next part shows you how to send a message using a messaging provider.

Client with a Messaging Provider

Using a messaging provider gives you more flexibility than a standalone client has because it can take advantage of the additional functionality that a messaging provider can offer.

Getting a ProviderConnection Object

Whereas a `SOAPConnection` object is a point-to-point connection directly to a particular URL, a `ProviderConnection` object is a connection to a messaging provider. With this kind of connection, all messages that you send or receive go through the messaging provider.

As with getting a `SOAPConnection` object, the first step is to get a connection factory, but in this case, it is a `ProviderConnectionFactory` object. You can obtain a `ProviderConnectionFactory` object by retrieving it from a naming service. This is possible when your application is using a messaging provider and is deployed in a servlet or J2EE container. With a `ProviderConnection-Factory` object, you can create a connection to a particular messaging provider and thus be able to use the capabilities of a profile that the messaging provider supports.

To get a `ProviderConnectionFactory` object, you first supply the logical name of your messaging provider to the container at deployment time. This is the name associated with your messaging provider that has been registered with a naming service based on the Java Naming and Directory Interface™ (JNDI) API. You can then do a lookup using this name to obtain a `ProviderConnec-tionFactory` object that will create connections to your messaging provider. For example, if the name registered for your messaging provider is “ProviderABC”, you can do a lookup on “ProviderABC” to get a `ProviderConnectionFactory` object and use it to create a connection to your messaging provider. This is what is done in the following code fragment. The first two lines use methods from the JNDI API to retrieve the `ProviderConnectionFactory` object, and the last line uses a method from the JAXM API to create the connection to the messaging provider. Note that because the JNDI method `lookup` returns a Java `Object`, you must convert it to a ProviderConnectionFactory object before assigning it to the variable pcFactory.

```
Context ctx = new InitialContext();
ProviderConnectionFactory pcFactory =
     (ProviderConnectionFactory)ctx.lookup("ProviderABC");

ProviderConnection pcCon = pcFactory.createConnection();
```

You will use pcCon, which represents a connection to your messaging provider, to get information about your messaging provider and to send the message you will create in the next section.

Creating a Message

You create all JAXM messages by getting a MessageFactory object and using it to create the SOAPMessage object. For the standalone client example, you simply used the default MessageFactory object obtained via the method MessageFactory.newInstance. However, when you are using a messaging provider, you obtain the MessageFactory object in a different way.

Getting a MessageFactory

If you are using a messaging provider, you create a MessageFactory object by using the method ProviderConnection.createMessageFactory. In addition, you pass it a String indicating the profile you want to use. To find out which profiles your messaging provider supports, you need to get a ProviderMetaData object with information about your provider. This is done by calling the method getMetaData on the connection to your provider. Then you need to call the method getSupportedProfiles to get an array of the profiles your messaging provider supports. Supposing that you want to use the ebXML profile, you need to see if any of the profiles in the array matches "ebxml". If there is a match, that profile is assigned to the variable profile, which can then be passed to the method createMessageFactory.

```
ProviderMetaData metaData = pcCon.getMetaData();
String[] supportedProfiles = metaData.getSupportedProfiles();
String profile = null;

for (int i=0; i < supportedProfiles.length; i++) {
  if (supportedProfiles[i].equals("ebxml")) {
     profile = supportedProfiles[i];
     break;
```

```
    }
}

MessageFactory factory = pcCon.createMessageFactory(profile);
```

You can now use `factory` to create a `SOAPMessage` object that conforms to the ebXML profile. This example uses the minimal ebXML profile implementation included in the Java WSDP. Note that the following line of code uses the class `EbXMLMessageImpl`, which is defined in the ebXML profile implementation and is not part of the JAXM API.

```
EbXMLMessageImpl message = (EbXMLMessageImpl)factory.
                           createMessage();
```

For this profile, instead of using `Endpoint` objects, you indicate `Party` objects for the sender and the receiver. This information will appear in the message's header, and the messaging provider will use it to determine where to send the message. The following lines of code use the methods `setSender` and `setReceiver`, which are defined in the `EbXMLMessageImpl` implementation. These methods not only create a `SOAPHeader` object but also give it content. You can use these methods because your `SOAPMessage` object is an `EbXMLMessageImpl` object, giving you access to the methods defined in `EbXMLMessageImpl`.

```
message.setSender(new Party("http://grand.products.com"));
message.setReceiver(new Party("http://whiz.gizmos.com"));
```

You can view the API documentation for the ebXML and SOAP-RP profile implementations provided in this Java WSDP at the following location:

```
<JWSDP_HOME>/jaxm-1.1.1/docs/profiles/index.html
```

If you are not using a profile or you want to set content for a header not covered by your profile's implementation, you need to follow the steps shown in the next section.

Adding Content to the Header

To add content to the header, you need to create a `SOAPHeaderElement` object. As with all new elements, it must have an associated `Name` object, which you create using the message's `SOAPEnvelope` object.

The following code fragment retrieves the `SOAPHeader` object from `envelope` and adds a new `SOAPHeaderElement` object to it.

```
SOAPHeader header = envelope.getHeader();
Name headerName = envelope.createName("Purchase Order",
               "PO", "http://www.sonata.com/order");
SOAPHeaderElement headerElement =
               header.addHeaderElement(headerName);
```

At this point, `header` contains the `SOAPHeaderElement` object `headerElement` identified by the `Name` object `headerName`. Note that the `addHeaderElement` method both creates `headerElement` and adds it to `header`.

Now that you have identified `headerElement` with `headerName` and added it to `header`, the next step is to add content to `headerElement`, which the next line of code does with the method `addTextNode`.

```
headerElement.addTextNode("order");
```

Now you have the `SOAPHeader` object `header` that contains a `SOAPHeaderElement` object whose content is "order".

Adding Content to the SOAP Body

The process for adding content to the `SOAPBody` object is the same for clients using a messaging provider as it is for standalone clients. This is also the same as the process for adding content to the `SOAPHeader` object. You access the `SOAPBody` object, add a `SOAPBodyElement` object to it, and add text to the `SOAPBodyElement` object. It is possible to add additional `SOAPBodyElement` objects, and it is possible to add subelements to the `SOAPBodyElement` objects with the method `addChildElement`. For each element or child element, you add content with the method `addTextNode`.

The section on the standalone client demonstrated adding one `SOAPBodyElement` object, adding a child element, and giving it some text. The following example shows adding more than one `SOAPBodyElement` and adding text to each of them.

The code first creates the `SOAPBodyElement` object `purchaseLineItems`, which has a fully-qualified namespace associated with it. That is, the `Name` object for it has a local name, a namespace prefix, and a namespace URI. As you saw earlier, a `SOAPBodyElement` object is required to have a fully-qualified namespace, but child elements added to it may have `Name` objects with only the local name.

```
SOAPBody body = envelope.getBody();
Name bodyName = envelope.createName("PurchaseLineItems", "PO",
              "http://sonata.fruitsgalore.com");
SOAPBodyElement purchaseLineItems =
                 body.addBodyElement(bodyName);

Name childName = envelope.createName("Order");
SOAPElement order =
       purchaseLineItems.addChildElement(childName);

childName = envelope.createName("Product");
SOAPElement product = order.addChildElement(childName);
product.addTextNode("Apple");

childName = envelope.createName("Price");
SOAPElement price = order.addChildElement(childName);
price.addTextNode("1.56");

childName = envelope.createName("Order");
SOAPElement order2 =
       purchaseLineItems.addChildElement(childName);

childName = envelope.createName("Product");
SOAPElement product2 = order2.addChildElement(childName);
product2.addTextNode("Peach");

childName = envelope.createName("Price");
SOAPElement price2 = order2.addChildElement(childName);
price2.addTextNode("1.48");
```

The JAXM code in the preceding example produces the following XML in the SOAP body:

```
<PO:PurchaseLineItems
 xmlns:PO="http://www.sonata.fruitsgalore/order">
  <Order>
    <Product>Apple</Product>
    <Price>1.56</Price>
  </Order>

  <Order>
```

```
      <Product>Peach</Product>
      <Price>1.48</Price>
    </Order>
  </PO:PurchaseLineItems>
```

Adding Content to the SOAPPart Object

If the content you want to send is in a file, JAXM provides an easy way to add it directly to the `SOAPPart` object. This means that you do not access the `SOAPBody` object and build the XML content yourself, as you did in the previous section.

To add a file directly to the `SOAPPart` object, you use a `javax.xml.transform.Source` object from JAXP (the Java API for XML Processing). There are three types of `Source` objects: `SAXSource`, `DOMSource`, and `StreamSource`. A `StreamSource` object holds content as an XML document. `SAXSource` and `DOMSource` objects hold content along with the instructions for transforming the content into an XML document.

The following code fragment uses JAXP API to build a `DOMSource` object that is passed to the `SOAPPart.setContent` method. The first two lines of code get a `DocumentBuilderFactory` object and use it to create the `DocumentBuilder` object `builder`. Then `builder` parses the content file to produce a `Document` object, which is used to initialize a new `DOMSource` object.

```
DocumentBuilderFactory dbFactory = DocumentBuilderFactory.
                         newInstance();
DocumentBuilder builder = dbFactory.newDocumentBuilder();
Document doc = builder.parse("file:///music/order/soap.xml");
DOMSource domSource = new DOMSource(doc);
```

The following two lines of code access the `SOAPPart` object (using the `SOAPMessage` object `message`) and set the new `DOMSource` object as its content. The method `SOAPPart.setContent` not only sets content for the `SOAPBody` object but also sets the appropriate header for the `SOAPHeader` object.

```
SOAPPart soapPart = message.getSOAPPart();
soapPart.setContent(domSource);
```

You will see other ways to add content to a message in the section on `AttachmentPart` objects. One big difference to keep in mind is that a `SOAPPart` object must contain only XML data, whereas an `AttachmentPart` object may contain any type of content.

Sending the Message

When the connection is a `ProviderConnection` object, messages have to be sent using the method `ProviderConnection.send`. This method sends the message passed to it and returns immediately. Unlike the `SOAPConnection` method `call`, it does not have to block until it receives a response, which leaves the application free to do other things.

The `send` method takes only one argument, the message to be sent. It does not need to be given the destination because the messaging provider can use information in the header to figure out where the message needs to go.

```
pcCon.send(message);
pcCon.close();
```

Adding Attachments

Adding `AttachmentPart` objects to a message is the same for all clients, whether they use a messaging provider or not. As noted in earlier sections, you can put any type of content, including XML, in an `AttachmentPart` object. And because the SOAP part can contain only XML content, you must use an `AttachmentPart` object for any content that is not in XML format.

Creating an AttachmentPart Object and Adding Content

The `SOAPMessage` object creates an `AttachmentPart` object, and the message also has to add the attachment to itself after content has been added. The `SOAPMessage` class has three methods for creating an `AttachmentPart` object.

The first method creates an attachment with no content. In this case, an `AttachmentPart` method is used later to add content to the attachment.

```
AttachmentPart attachment = message.createAttachmentPart();
```

You add content to `attachment` with the `AttachmentPart` method `setContent`. This method takes two parameters, a Java `Object` for the content, and a `String` object that gives the content type. Content in the `SOAPBody` part of a message automatically has a Content-Type header with the value "text/xml" because the content has to be in XML. In contrast, the type of content in an `AttachmentPart` object has to be specified because it can be any type.

Each AttachmentPart object has one or more headers associated with it. When you specify a type to the method setContent, that type is used for the header Content-Type. Content-Type is the only header that is required. You may set other optional headers, such as Content-Id and Content-Location. For convenience, JAXM provides get and set methods for the headers Content-Type, Content-Id, and Content-Location. These headers can be helpful in accessing a particular attachment when a message has multiple attachments. For example, to access the attachments that have particular headers, you call the SOAPMessage method getAttachments and pass it the header or headers you are interested in.

The following code fragment shows one of the ways to use the method setContent. The Java Object being added is a String, which is plain text, so the second argument has to be "text/plain". The code also sets a content identifier, which can be used to identify this AttachmentPart object. After you have added content to attachment, you need to add attachment to the SOAPMessage object, which is done in the last line.

```
String stringContent = "Update address for Sunny Skies " +
  "Inc., to 10 Upbeat Street, Pleasant Grove, CA 95439";

attachment.setContent(stringContent, "text/plain");
attachment.setContentId("update_address");

message.addAttachmentPart(attachment);
```

The variable attachment now represents an AttachmentPart object that contains the String stringContent and has a header that contains the String "text/plain". It also has a Content-Id header with "update_address" as its value. And now attachment is part of message.

Let's say you also want to attach a jpeg image showing how beautiful the new location is. In this case, the second argument passed to setContent must be "image/jpeg" to match the content being added. The code for adding an image might look like the following. For the first attachment, the Object passed to the method setContent was a String. In this case, it is a stream.

```
AttachmentPart attachment2 = message.createAttachmentPart();

byte[] jpegData = . . .;
ByteArrayInputStream stream = new ByteArrayInputStream(
                        jpegData);
```

```
attachment2.setContent(stream, "image/jpeg");

message.addAttachmentPart(attachment);
```

The other two SOAPMessage.createAttachment methods create an AttachmentPart object complete with content. One is very similar to the AttachmentPart.setContent method in that it takes the same parameters and does essentially the same thing. It takes a Java Object containing the content and a String giving the content type. As with AttachmentPart.setContent, the Object may be a String, a stream, a javax.xml.transform.Source object, or a javax.activation.DataHandler object. You have already seen an example of using a Source object as content. The next example will show how to use a DataHandler object for content.

The other method for creating an AttachmentPart object with content takes a DataHandler object, which is part of the JavaBeans™ Activation Framework (JAF). Using a DataHandler object is fairly straightforward. First you create a java.net.URL object for the file you want to add as content. Then you create a DataHandler object initialized with the URL object and pass it to the method createAttachmentPart.

```
URL url = new URL("http://greatproducts.com/gizmos/img.jpg");
DataHandler dh = new DataHandler(url);
AttachmentPart attachment = message.createAttachmentPart(dh);
attachment.setContentId("gyro_image");

message.addAttachmentPart(attachment);
```

You might note two things about the previous code fragment. First, it sets a header for Content-ID with the method setContentId. This method takes a String that can be whatever you like to identify the attachment. Second, unlike the other methods for setting content, this one does not take a String for Content-Type. This method takes care of setting the Content-Type header for you, which is possible because one of the things a DataHandler object does is determine the data type of the file it contains.

Accessing an AttachmentPart Object

If you receive a message with attachments or want to change an attachment to a message you are building, you will need to access the attachment. When it is given no argument, the method SOAPMessage.getAttachments returns a java.util.Iterator object over all the AttachmentPart objects in a message.

The following code prints out the content of each `AttachmentPart` object in the `SOAPMessage` object `message`.

```
java.util.Iterator it = message.getAttachments();
while (it.hasNext()) {
  AttachmentPart attachment = (AttachmentPart)it.next();
  Object content = attachment.getContent();
  String id = attachment.getContentId();
  System.out.print("Attachment " + id + " contains: " +
                              content);
  System.out.println("");
}
```

Summary

In this section, you have been introduced to the basic JAXM API. You have seen how to create and send SOAP messages as a standalone client and as a client using a messaging provider. You have walked through adding content to a SOAP header and a SOAP body and also walked through creating attachments and giving them content. In addition, you have seen how to retrieve the content from the SOAP part and from attachments. In other words, you have walked through using the basic JAXM API.

SOAP Faults

This section expands on the basic JAXM API by showing you how to use the API for creating and accessing a SOAP Fault element in an XML message.

Overview

If you send a message that was not successful for some reason, you may get back a response containing a SOAP Fault element that gives you status information, error information, or both. There can be only one SOAP Fault element in a message, and it must be an entry in the SOAP Body. The SOAP 1.1 specification defines only one Body entry, which is the SOAP Fault element. Of course, the SOAP Body may contain other Body entries, but the SOAP Fault element is the only one that has been defined.

A `SOAPFault` object, the representation of a SOAP Fault element in the JAXM API, is similar to an `Exception` object in that it conveys information about a problem. However, a `SOAPFault` object is quite different in that it is an element in a message's SOAPBody object rather than part of the try/catch mechanism used for Exception objects. Also, as part of the SOAPBody object, which provides a simple means for sending mandatory information intended for the ultimate recipient, a SOAPFault object only reports status or error information. It does not halt the execution of an application the way an Exception object can.

Various parties may supply a SOAPFault object in a message. If you are a standalone client using the SAAJ API, and thus sending point-to-point messages, the recipient of your message may add a SOAPFault object to the response to alert you to a problem. For example, if you sent an order with an incomplete address for where to send the order, the service receiving the order might put a SOAPFault object in the return message telling you that part of the address was missing.

In another scenario, if you use the JAXM 1.1.1 API in order to use a messaging provider, the messaging provider may be the one to supply a SOAPFault object. For example, if the provider has not been able to deliver a message because a server is unavailable, the provider might send you a message with a SOAPFault object containing that information. In this case, there was nothing wrong with the message itself, so you can try sending it again later without any changes. In the previous example, however, you would need to add the missing information before sending the message again.

A SOAPFault object contains the following elements:

- a **fault code** — always required
  The SOAP 1.1 specification defines a set of fault code values in section 4.4.1, which a developer may extend to cover other problems. The default fault codes defined in the specification relate to the JAXM API as follows:
  - VersionMismatch — the namespace for a SOAPEnvelope object was invalid
  - MustUnderstand — an immediate child element of a SOAPHeader object had its mustUnderstand attribute set to "1", and the processing party did not understand the element or did not obey it
  - Client — the SOAPMessage object was not formed correctly or did not contain the information needed to succeed

    - `Server` — the `SOAPMessage` object could not be processed because of a processing error, not because of a problem with the message itself
- a **fault string** — always required
  a human readable explanation of the fault
- a **fault actor** — required if the `SOAPHeader` object contains one or more `actor` attributes; optional if no actors are specified, meaning that the only actor is the ultimate destination
  The fault actor, which is specified as a URI, identifies who caused the fault. For an explanation of what an actor is, see the section Intermediate Destinations (page 493).
- a **`Detail` object** — required if the fault is an error related to the `SOAPBody` object
  If, for example, the fault code is "Client", indicating that the message could not be processed because of a problem in the `SOAPBody` object, the `SOAP-Fault` object must contain a `Detail` object that gives details about the problem. If a `SOAPFault` object does not contain a `Detail` object, it can be assumed that the `SOAPBody` object was processed successfully.

Creating and Populating a SOAPFault Object

You have already seen how to add content to a `SOAPBody` object; this section will walk you through adding a `SOAPFault` object to a `SOAPBody` object and then adding its constituent parts.

As with adding content, the first step is to access the `SOAPBody` object.

```
SOAPEnvelope envelope =
                msg.getSOAPPart().getEnvelope();
SOAPBody body = envelope.getBody();
```

With the `SOAPBody` object *`body`* in hand, you can use it to create a `SOAPFault` object with the following line of code.

```
SOAPFault fault = body.addFault();
```

The following code uses convenience methods to add elements and their values to the `SOAPFault` object *`fault`*. For example, the method `setFaultCode` creates an element, adds it to `fault`, and adds a `Text` node with the value `"Server"`.

```
fault.setFaultCode("Server");
fault.setFaultActor("http://gizmos.com/orders");
fault.setFaultString("Server not responding");
```

The SOAPFault object *fault* created in the previous lines of code indicates that the cause of the problem is an unavailable server and that the actor at "http://gizmos.com/orders" is having the problem. If the message were being routed only to its ultimate destination, there would have been no need for setting a fault actor. Also note that *fault* does not have a Detail object because it does not relate to the SOAPBody object.

The following code fragment creates a SOAPFault object that includes a Detail object. Note that a SOAPFault object may have only one Detail object, which is simply a container for DetailEntry objects, but the Detail object may have multiple DetailEntry objects. The Detail object in the following lines of code has two DetailEntry objects added to it.

```
SOAPFault fault = body.addFault();

fault.setFaultCode("Client");
fault.setFaultString("Message does not have necessary info");

Detail detail = fault.addDetail();

Name entryName = envelope.createName("order", "PO",
                    "http://gizmos.com/orders/");
DetailEntry entry = detail.addDetailEntry(entryName);
entry.addTextNode("quantity element does not have a value");

Name entryName2 = envelope.createName("confirmation", "PO",
                      "http://gizmos.com/confirm");
DetailEntry entry2 = detail.addDetailEntry(entryName2);
entry2.addTextNode("Incomplete address: no zip code");
```

Retrieving Fault Information

Just as the SOAPFault interface provides convenience methods for adding information, it also provides convenience methods for retrieving that information. The following code fragment shows what you might write to retrieve fault information from a message you received. In the code fragment, newmsg is the SOAPMessage object that has been sent to you. Because a SOAPFault object must be part of the SOAPBody object, the first step is to access the SOAPBody object. Then the code tests to see if the SOAPBody object contains a SOAPFault object. If so, the code retrieves the SOAPFault object and uses it to retrieve its contents. The convenience methods getFaultCode, getFaultString, and getFaultActor make retrieving the values very easy.

```
SOAPBody body =
          newmsg.getSOAPPart().getEnvelope().getBody();
if ( body.hasFault() ) {
   SOAPFault newFault = body.getFault();
   String code = newFault.getFaultCode();
   String string = newFault.getFaultString();
   String actor = newFault.getFaultActor();
```

Next the code prints out the values it just retrieved. Not all messages are required to have a fault actor, so the code tests to see if there is one. Testing whether the variable actor is null works because the method getFaultActor returns null if a fault actor has not been set.

```
   System.out.println("SOAP fault contains: ");
   System.out.println("    fault code = " + code);
   System.out.println("    fault string = " + string);

   if ( actor != null ) {
      System.out.println("    fault actor = " + actor);
   }
}
```

The final task is to retrieve the Detail object and get its DetailEntry objects. The code uses the SOAPFault object newFault to retrieve the Detail object newDetail, and then it uses newDetail to call the method getDetailEntries. This method returns the java.util.Iterator object it, which contains all of the DetailEntry objects in newDetail. Not all SOAPFault objects are required to have a Detail object, so the code tests to see whether newDetail is null. If it is not, the code prints out the values of the DetailEntry object(s) as long as there are any.

```
Detail newDetail = newFault.getDetail();
if ( newDetail != null) {
   Iterator it = newDetail.getDetailEntries();
   while ( it.hasNext() ) {
      DetailEntry entry = (DetailEntry)it.next();
      String value = entry.getValue();
      System.out.println("    Detail entry = " + value);
   }
}
```

In summary, you have seen how to add a SOAPFault object and its contents to a message as well as how to retrieve the information in a SOAPFault object. A SOAPFault object, which is optional, is added to the SOAPBody object to convey status or error information. It must always have a fault code and a String explanation of the fault. A SOAPFault object must indicate the actor that is the source of the fault only when there are multiple actors; otherwise, it is optional. Similarly, the SOAPFault object must contain a Detail object with one or more DetailEntry objects only when the contents of the SOAPBody object could not be processed successfully.

Code Examples

The first part of this tutorial used code fragments to walk you through the fundamentals of using the JAXM API. In this section, you will use some of those code fragments to create applications. First, you will see the program Request.java. Then you will see how to create and run the application MyUddiPing.java. Finally, you will see how to create and run SOAPFaultTest.java.

Request.java

The class Request.java puts together the code fragments used in the section Client without a Messaging Provider (page 499) and adds what is needed to make it a complete example of a client sending a request-response message. In addition to putting all the code together, it adds import statements, a main method, and a try/catch block with exception handling. The file Request.java, shown here in its entirety, is a standalone client application that uses the SAAJ API (the `javax.xml.soap` package). It does not need to use the `javax.xml.messaging` package because it does not use a messaging provider.

```
import javax.xml.soap.*;
import java.util.*;
import java.net.URL;

public class Request {
  public static void main(String[] args){
    try {
      SOAPConnectionFactory scFactory =
           SOAPConnectionFactory.newInstance();
      SOAPConnection con = scFactory.createConnection();

      MessageFactory factory =
           MessageFactory.newInstance();
      SOAPMessage message = factory.createMessage();

      SOAPPart soapPart = message.getSOAPPart();
      SOAPEnvelope envelope = soapPart.getEnvelope();
      SOAPHeader header = envelope.getHeader();
      SOAPBody body = envelope.getBody();
      header.detachNode();

      Name bodyName = envelope.createName(
                "GetLastTradePrice", "m",
                "http://wombats.ztrade.com");
      SOAPBodyElement gltp =
                body.addBodyElement(bodyName);

      Name name = envelope.createName("symbol");
      SOAPElement symbol = gltp.addChildElement(name);
      symbol.addTextNode("SUNW");

      URL endpoint = new URL
                  ("http://wombat.ztrade.com/quotes";
      SOAPMessage response = con.call(message, endpoint);

      con.close();

      SOAPPart sp = response.getSOAPPart();
      SOAPEnvelope se = sp.getEnvelope();
      SOAPBody sb = se.getBody();

      Iterator it = sb.getChildElements(bodyName);
      SOAPBodyElement bodyElement =
                (SOAPBodyElement)it.next();
```

```
            String lastPrice = bodyElement.getValue();

            System.out.print("The last price for SUNW is ");
            System.out.println(lastPrice);

        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

In order for `Request.java` to be runnable, the second argument supplied to the method `call` has to be a valid existing URI, which is not true in this case. See the JAXM code in the case study for similar code that you can run (JAXM Client, page 759). Also, the application in the next section is one that you can run.

UddiPing.java and MyUddiPing.java

The sample program `UddiPing.java` is another example of a standalone application. A Universal Description, Discovery and Integration (UDDI) service is a business registry and repository from which you can get information about businesses that have registered themselves with the registry service. For this example, the `UddiPing` application is not actually accessing a UDDI service registry but rather a test (demo) version. Because of this, the number of businesses you can get information about is limited. Nevertheless, `UddiPing` demonstrates a request being sent and a response being received. The application prints out the complete message that is returned, that is, the complete XML document as it looks when it comes over the wire. Later in this section you will see how to rewrite `UddiPing.java` so that in addition to printing out the entire XML document, it also prints out just the text content of the response, making it much easier to see the information you want.

In order to get a better idea of how to run the `UddiPing` example, take a look at the directory `<`*`JWSDP_HOME`*`>/jaxm-1.1.1/samples/uddiping`. This directory contains the subdirectory `src` and the files `run.sh` (or `run.bat`), `uddi.properties`, `UddiPing.class`, and `README`. The `README` file tells you what you need to do to run the application, which is explained more fully here.

The `README` file directs you to modify the file `uddi.properties`, which contains the URL of the destination (the UDDI test registry) and the proxy host and proxy port of the sender. If you are in the `uddiping` directory when you call the `run.sh` (or `run.bat`) script, the information in `uddi.properties` should be correct already. If you are outside Sun Microsystem's firewall, however, you need to supply your proxy host and proxy port. If you are not sure what the values for these are, you need to consult your system administrator or other person with that information.

The main job of the `run` script is to execute `UddiPing`. Once the file `uddi.properties` has the correct proxy host and proxy port, you can call the appropriate `run` script as shown here. Note that you must supply two arguments, `uddi.properties` and the name of the business you want to look up.

Unix:

```
cd <JWSDP_HOME>/jaxm-1.1.1/samples/uddiping
run.sh uddi.properties Microsoft
```

Windows:

```
cd <JWSDP_HOME>\jaxm-1.1.1\samples\uddiping
run.bat uddi.properties Microsoft
```

What appears on your screen will look something like this (but much longer):

```
Received reply from:
http://www3.ibm.com/services/uddi/inquiryapi<?xml
version="1.0" encoding="UTF-8" ?><Envelope
xmlns="http://schemas.xmlsoap.org/soap/envelope/"><Body><busin
essList generic="1.0" xmlns="urn:uddi-org:api"
operator="www.ibm.com/services/uddi"
truncated="false"><businessInfos><businessInfo
businessKey="D7475060-BF58-11D5-A432-
0004AC49CC1E"><name>Microsoft Corporation</name><description
xml:lang="en">Computer Software and Hardware
Manufacturer</description><serviceInfos></serviceInfos></busin
essInfo></businessInfos></businessList></Body></Envelope>
```

If the business name you specified is in the test registry, the output is an XML document with the name and description of that business. However, these are embedded in the XML document, which makes them difficult to see. The next section adds code to `UddiPing.java` that extracts the content so that it is readily visible.

Creating MyUddiPing.java

To make the response to UddiPing.java easier to read, you will create a new file called MyUddiPing.java, which extracts the content and prints it out. You will see how to write the new file later in this section after setting up a new directory with the necessary subdirectories and files.

Setting Up

Because the name of the new file is MyUddiPing.java, create the directory myuddiping under the <*JWSDP_HOME*>/jaxm-1.1.1/samples directory.

```
cd <JWSDP_HOME>/jaxm-1.1.1/samples
mkdir myuddiping
```

This new myuddiping directory will be the base directory for all future commands relating to MyUddiPing.java.

In place of the run.sh or run.bat script used for running UddiPing, you will be using an Ant file, build.xml, for setting up directories and files and for running MyUddiPing. The advantage of using an Ant file is that it is cross-platform and can thus be used for both Unix and Windows platforms. Accordingly, you need to copy the build.xml file in the examples/jaxm directory of the tutorial to your new myuddiping directory. (The command for copying should be all on one line. Note that there is no space between "myuddiping/" "and "build", and there is a "." at the end of the command line.)

Unix:

```
cd myuddiping
cp <JWSDP_HOME>/docs/tutorial/examples/jaxm/myuddiping/
                                                  build.xml .
```

Windows:

```
cd myuddiping
copy <JWSDP_HOME>\docs\tutorial\examples\jaxm\myuddiping\
                                                  build.xml .
```

Once you have the file build.xml in your myuddiping directory, you can call it to do the rest of the setup and also to run MyUddiPing. An Ant build file is an XML file that is sectioned into *targets*, with each target being an element that contains attributes and one or more tasks. For example, the target element whose name attribute is prepare creates the directories build and src and copies the file MyUddiPing.java from the *<JWSDP_HOME>*/docs/tutorial/examples/jaxm/myuddiping/src directory to the new src directory. Then it copies the file uddi.properties from the uddiping directory to the myuddiping directory that you created.

To accomplish these tasks, you type the following at the command line:

```
ant prepare
```

The target named build compiles the source file MyUddiPing.java and puts the resulting .class file in the build directory. So to do these tasks, you type the following at the command line:

```
ant build
```

Now that you are set up for running MyUddiPing, let's take a closer look at the code.

Examining MyUddiPing

We will go through the file MyUddiPing.java a few lines at a time. Note that most of the class MyUddiPing.java is based on UddiPing.java. We will be adding a section at the end of MyUddiPing.java that accesses only the content you want from the response that is returned by the method call.

The first four lines of code import the packages used in the application.

```
import javax.xml.soap.*;
import javax.xml.messaging.*;
import java.util.*;
import java.io.*;
```

The next few lines begin the definition of the class MyUddiPing, which starts with the definition of its main method. The first thing it does is check to see if two arguments were supplied. If not, it prints a usage message and exits.

```
public class MyUddiPing {
  public static void main(String[] args) {
    try {
      if (args.length != 2) {
        System.err.println("Usage: MyUddiPing " +
          "properties-file business-name");
        System.exit(1);
      }
```

The following lines create a `java.util.Properties` file that contains the system properties and the properties from the file `uddi.properties` that is in the `myuddiping` directory.

```
Properties myprops = new Properties();
myprops.load(new FileInputStream(args[0]));
Properties props = System.getProperties();
Enumeration it = myprops.propertyNames();
while (it.hasMoreElements()) {
   String s = (String) it.nextElement();
   props.put(s, myprops.getProperty(s));
}
```

The next four lines create a `SOAPMessage` object. First, the code gets an instance of `SOAPConnectionFactory` and uses it to create a connection. Then it gets an instance of `MessageFactory` and uses it to create a message.

```
SOAPConnectionFactory scf =
     SOAPConnectionFactory.newInstance();
SOAPConnection connection =
     scf.createConnection();
MessageFactory msgFactory =
     MessageFactory.newInstance();
SOAPMessage msg = msgFactory.createMessage();
```

The new `SOAPMessage` object `msg` automatically contains a `SOAPPart` object that contains a `SOAPEnvelope` object. The `SOAPEnvelope` object contains a `SOAPBody` object, which is the element you want to access in order to add content to it. The next lines of code get the `SOAPPart` object, the `SOAPEnvelope` object, and the `SOAPBody` object.

```
SOAPEnvelope envelope =
       msg.getSOAPPart().getEnvelope();
SOAPBody body = envelope.getBody();
```

The following lines of code add an element with a fully-qualified name and then add two attributes to the new element. The first attribute has the name `"generic"` and the value `"2.0"`. The second attribute has the name `"maxRows"` and the value `"100"`. Then the code adds a child element with the name `name` and adds some text to it with the method addTextNode. The text added is the business name you will supply when you run the application.

```
SOAPBodyElement findBusiness =
     body.addBodyElement(
     envelope.createName("find_business",
     "", "urn:uddi-org:api_v2"));
findBusiness.addAttribute(
     envelope.createName("generic", "2.0");
findBusiness.addAttribute(
     envelope.createName("maxRows", "100");
SOAPElement businessName =
     findBusiness.addChildElement(
     envelope.createName("name"));
businessName.addTextNode(args[1]);
```

The next line of code creates the Java Object that represents the destination for this message. It gets the value of the property named "URL" from the system property file.

```
Object endpoint =
   System.getProperties().getProperty("URL");
```

The following line of code saves the changes that have been made to the message. This method will be called automatically when the message is sent, but it does not hurt to call it explicitly.

```
msg.saveChanges();
```

Next the message msg is sent to the destination that endpoint represents, which is the test UDDI registry. The method call will block until it gets a SOAPMessage object back, at which point it returns the reply.

```
SOAPMessage reply = connection.call(msg, endpoint);
```

In the next two lines, the first prints out a line giving the URL of the sender (the test registry), and the second prints out the returned message as an XML document.

```
System.out.println("Received reply from: " +endpoint);
reply.writeTo(System.out);
```

The code thus far has been based on UddiPing.java. The next section adds code to create MyUddiPing.java.

Adding New Code

The code we are going to add to `UddiPing` will make the reply more user-friendly. It will get the content from certain elements rather than printing out the whole XML document as it was sent over the wire. Because the content is in the `SOAPBody` object, the first thing you need to do is access it, as shown in the following line of code. You can access each element in separate method calls, as was done in earlier examples, or you can access the `SOAPBody` object using this shorthand version.

```
SOAPBody replyBody =
    reply.getSOAPPart().getEnvelope().getBody();
```

Next you might print out two blank lines to separate your results from the raw XML message and a third line that describes the text that follows.

```
        System.out.println("");
        System.out.println("");
        System.out.print(
         "Content extracted from the reply message: ");
```

Now you can begin the process of getting all of the child elements from an element, getting the child elements from each of those, and so on, until you arrive at a text element that you can print out. Unfortunately, the registry used for this example code, being just a test registry, is not always consistent. The number of subelements sometimes varies, making it difficult to know how many levels down the code needs to go. And in some cases, there are multiple entries for the same company name. Note that by contrast, the entries in a standard valid registry will be consistent.

The code you will be adding drills down through the subelements within the SOAP body and retrieves the name and description of the business. The method you use to retrieve child elements is the `SOAPElement` method `getChildElements`. When you give this method no arguments, it retrieves all of the child elements of the element on which it is called. If you know the `Name` object used to name an element, you can supply that to `getChildElements` and retrieve only the children with that name. In this example, however, you need to retrieve all elements and keep drilling down until you get to the elements that contain text content.

Here is the basic pattern that is repeated for drilling down:

```
Iterator iter1 = replyBody.getChildElements();
while (iter1.hasNext()) {
   SOAPBodyElement bodyElement =
         (SOAPBodyElement)iter1.next();
   Iterator iter2 =
         bodyElement.getChildElements();
   while (iter2.hasNext()) {
```

The method `getChildElements` returns the elements in the form of a `java.util.Iterator` object. You access the child elements by calling the method `next` on the `Iterator` object. The method `Iterator.hasNext` can be used in a `while` loop because it returns `true` as long as the next call to the method `next` will return a child element. The loop ends when there are no more child elements to retrieve.

An immediate child of a `SOAPBody` object is a `SOAPBodyElement` object, which is why calling `iter1.next` returns a `SOAPBodyElement` object. Children of `SOAP-BodyElement` objects and all child elements from there down are `SOAPElement` objects. For example, the call `iter2.next` returns the `SOAPElement` object `child2`. Note that the method `Iterator.next` returns an `Object`, which has to be narrowed (cast) to the specific kind of object you are retrieving. Thus, the result of calling `iter1.next` is cast to a `SOAPBodyElement` object, whereas the results of calling `iter2.next`, `iter3.next`, and so on, are `all` cast to a `SOAPElement` object.

Here is the code you add to access and print out the business name and description:

```
Iterator iter1 = replyBody.getChildElements();
while (iter1.hasNext()) {
   SOAPBodyElement bodyElement =
         (SOAPBodyElement)iter1.next();
   Iterator iter2 =
         bodyElement.getChildElements();
   while (iter2.hasNext()) {
      SOAPElement child2 =
         (SOAPElement)iter2.next();
      Iterator iter3 =
         child2.getChildElements();
      String content = child2.getValue();
      System.out.println(content);
      while (iter3.hasNext()) {
         SOAPElement child3 =
            (SOAPElement)iter3.next();
```

```
                  Iterator iter4 =
                       child3.getChildElements();
                  content = child3.getValue();
                  System.out.println(content);
                  while (iter4.hasNext()) {
                     SOAPElement child4 =
                        (SOAPElement)iter4.next();
                     content = child4.getValue();
                     System.out.println(content);
                  }
               }
            }
         }
         connection.close();
      } catch (Exception ex) {
         ex.printStackTrace();
      }
   }
}
```

You have already compiled MyUddiPing.java by calling the following at the command line:

```
ant build
```

With the code compiled, you are ready to run MyUddiPing. The following command will call java on the .class file for MyUddiPing, which takes two arguments. The first argument is the file uddi.properties, which is supplied by a property set in build.xml. The second argument is the name of the business for which you want to get a description, and you need to supply this argument on the command line. Note that any property set on the command line overrides the value set for that property in the build.xml file. The last argument supplied to Ant is always the target, which in this case is run.

```
cd <JWSDP_HOME>/jaxm-1.1.1/samples/myuddiping
ant -Dbusiness-name=”Oracle” run
```

Here is the output that will appear after the full XML message. It is produced by the code added in MyUddiPing.java.

```
Content extracted from the reply message:

Oracle Corporation
Oracle Corporation provides the software and services for e-
business.
```

```
Oracle JDeveloper Web Services
Oracle9i JDeveloper provides end-to-end support for web
services and UDDI

Oracle Sample Web services
Business Account established to showcase Oracle’s Web services.
```

There may be some occurrences of “null” in the output.

SOAPFaultTest.java

The code `SOAPFaultTest.java`, based on the code fragments in a preceding section (SOAP Faults, page 516) creates a message with a `SOAPFault` object. It then retrieves the contents of the `SOAPFault` object and prints them out. You will find the code for `SOAPFaultTest` in the following directory:

```
<JWSDP_HOME>/docs/tutorial/examples/jaxm/fault/src
```

Here is the file `SOAPFaultTest.java`.

```
import javax.xml.soap.*;
import java.util.*;

public class SOAPFaultTest {

  public static void main(String[] args) {
    try {
      MessageFactory msgFactory =
                             MessageFactory.newInstance();
      SOAPMessage msg = msgFactory.createMessage();
      SOAPEnvelope envelope =
                          msg.getSOAPPart().getEnvelope();
      SOAPBody body = envelope.getBody();
      SOAPFault fault = body.addFault();

      fault.setFaultCode("Client");
      fault.setFaultString(
                "Message does not have necessary info");
      fault.setFaultActor("http://gizmos.com/order");

      Detail detail = fault.addDetail();

      Name entryName = envelope.createName("order", "PO",
                        "http://gizmos.com/orders/");
      DetailEntry entry = detail.addDetailEntry(entryName);
```

```
        entry.addTextNode(
             "quantity element does not have a value");

        Name entryName2 = envelope.createName("confirmation",
                    "PO", "http://gizmos.com/confirm");
        DetailEntry entry2 = detail.addDetailEntry(entryName2);
        entry2.addTextNode("Incomplete address: no zip code");

        msg.saveChanges();

        // Now retrieve the SOAPFault object and its contents
        //after checking to see that there is one

        if ( body.hasFault() ) {
           fault = body.getFault();
           String code = fault.getFaultCode();
           String string = fault.getFaultString();
           String actor = fault.getFaultActor();

           System.out.println("SOAP fault contains: ");
           System.out.println("    fault code = " + code);
           System.out.println("    fault string = " + string);
           if ( actor != null) {
              System.out.println("    fault actor = " + actor);
           }

           detail = fault.getDetail();
           if ( detail != null) {
              Iterator it = detail.getDetailEntries();
              while ( it.hasNext() ) {
                 entry = (DetailEntry)it.next();
                 String value = entry.getValue();
                 System.out.println(
                    "    Detail entry = " + value);
              }
           }
        }
     } catch (Exception ex) {
        ex.printStackTrace();
     }
   }
}
```

Running SOAPFaultTest

To run `SOAPFaultTest`, you use the Ant file `build.xml` that is in the directory `<`*`JWSDP_HOME`*`>/docs/tutorial/examples/jaxm/fault`.

This Ant file does many things for you, including creating a `build` directory where class files will go, creating the classpath needed to run `SOAPFaultTest`, compiling `SOAPFaulTest.java`, putting the resulting `.class` file in the `build` directory, and running `SOAPFaultTest`.

To run `SOAPFaultTest`, do the following:

1. Go to the directory where the appropriate `build.xml` file is located.

   ```
   cd <JWSDP_HOME>/docs/tutorial/examples/jaxm/fault
   ```

2. At the command line, type the following:

   ```
   ant prepare
   ```

   This will create the `build` directory, the directory where class files will be put.

3. At the command line, type

   ```
   ant build
   ```

   This will run `javac` on `SOAPFaultTest.java` using the classpath that has been set up in the `build.xml` file. The resulting `.class` file will be put in the `build` directory created by the prepare target.

4. At the command line, type

   ```
   ant run
   ```

   This will execute the command `java SOAPFaultTest`.

Note that as a shortcut, you can simply type `ant run`. The necessary targets will be executed in the proper order because if a target indicates that it depends on one or more other targets, those will be executed before the specified target is executed. In this case, the `run` target depends on the `build` target, which in turn depends on the `prepare` target, so the `prepare`, `build`, and `run` targets will be executed in that order. As an even faster shortcut, you can type just `ant`. The default target for this `build.xml file` is `run`, so it has the same effect as typing `ant run`.

If you want to run `SOAPFaultTest` again, it is a good idea to start over by deleting the `build` directory and the `.class` file it contains. You can do this by typing the following at the command line:

```
ant clean
```

After running `SOAPFaultTest`, you will see something like this:

```
Here is what the XML message looks like:

<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/
soap/envelope/"><soap-env:Header/><soap-env:Body><soap-env:
Fault><soap-env:faultcode>Client</soap-env:faultcode><soap-
env:faultstring>Message does not have necessary info</soap-
env:faultstring><soap-env:faultactor>http://gizmos.com/order
</soap-env:faultactor><soap-env:Detail><PO:order xmlns:PO=
"http://gizmos.com/orders/">quantity element does not have a
value</PO:order><PO:confirmation xmlns:PO="http://gizmos.com/
confirm">Incomplete address: no zip code</PO:confirmation>
</soap-env:Detail></soap-env:Fault></soap-env:Body></soap-env:
Envelope>

Here is what the SOAP fault contains:
  fault code = Client
  fault string = Message does not have necessary info
  fault actor = http://gizmos.com/order
  Detail entry = quantity element does not have a value
  Detail entry = Incomplete address: no zip code
```

Conclusion

JAXM provides a Java API that simplifies writing and sending XML messages. You have seen how to use this API to write client code for JAXM request-response messages and one-way messages. You have also seen how to get the content from a reply message. This knowledge was applied in writing and running the `MyUddiPing` and `SOAPFaultTest` examples. In addition, the case study (The Coffee Break Application, page 747) provides detailed examples of JAXM code for both the client and server.

You now have first-hand experience of how JAXM makes it easier to do XML messaging.

Further Information

You can find additional information about JAXM from the following:

- Documents bundled with the JAXM implementation at

`<*JWSDP_HOME*>/jaxm-1.1.1/docs`

- SAAJ 1.1 specification, available from
  `http://java.sun.com/xml/downloads/saaj.html`
- JAXM 1.1 specification, available from
  `http://java.sun.com/xml/downloads/jaxm.html`
- JAXM website at
  `http://java.sun.com/xml/jaxm/`
- JAXM sample applications (see Running the Samples, page 495)

13

Publishing and Discovering Web Services with JAXR

Kim Haase

THE Java API for XML Registries (JAXR) provides a uniform and standard Java API for accessing different kinds of XML registries.

The implementation of JAXR that is part of the Java Web Services Developer Pack (Java WSDP) includes several sample programs as well as a Registry Browser tool that also illustrates how to write a JAXR client program. See Registry Browser (page 839) for information about this tool.

After providing a brief overview of JAXR, this chapter describes how to implement a JAXR client to publish an organization and its web services to a registry and to query a registry to find organizations and services. Finally, it explains how to run the examples provided with this tutorial and offers links to more information on JAXR.

Overview of JAXR

This section covers the following topics:

- What is a registry?
- What is JAXR?
- JAXR architecture

What Is a Registry?

An XML *registry* is an infrastructure that enables the building, deployment, and discovery of Web services. It is a neutral third party that facilitates dynamic and loosely coupled business-to-business (B2B) interactions. A registry is available to organizations as a shared resource, often in the form of a Web-based service.

Currently there are a variety of specifications for XML registries. These include

- The ebXML Registry and Repository standard, which is sponsored by the Organization for the Advancement of Structured Information Standards (OASIS) and the United Nations Centre for the Facilitation of Procedures and Practices in Administration, Commerce and Transport (U.N./CEFACT)
- The Universal Description, Discovery, and Integration (UDDI) project, which is being developed by a vendor consortium

A *registry provider* is an implementation of a business registry that conforms to a specification for XML registries.

What Is JAXR?

JAXR enables Java software programmers to use a single, easy-to-use abstraction API to access a variety of XML registries. A unified JAXR information model describes content and metadata within XML registries.

JAXR gives developers the ability to write registry client programs that are portable across different target registries. JAXR also enables value-added capabilities beyond those of the underlying registries.

The current version of the JAXR specification includes detailed bindings between the JAXR information model and both the ebXML Registry and the

UDDI version 2 specifications. You can find the latest version of the specification at

```
http://java.sun.com/xml/downloads/jaxr.html
```

At this release of the Java WSDP, JAXR implements the level 0 capability profile defined by the JAXR specification. This level allows access to both UDDI and ebXML registries at a basic level. At this release, JAXR supports access only to UDDI version 2 registries.

Currently several UDDI version 2 registries exist. The Java WSDP Registry Server provides a UDDI version 2 registry that you can use to test your JAXR applications in a private environment. See The Java WSDP Registry Server (page 829) for details.

Several ebXML registries are under development, and one is available at the Center for E-Commerce Infrastructure Development (CECID), Department of Computer Science Information Systems, The University of Hong Kong (HKU). For information, see `http://www.cecid.hku.hk/Release/PR09APR2002.html`.

A JAXR provider for ebXML registries is available in open source at `http://ebxmlrr.sourceforge.net`.

JAXR Architecture

The high-level architecture of JAXR consists of the following parts:

- A *JAXR client*: a client program that uses the JAXR API to access a business registry via a JAXR provider.
- A *JAXR provider*: an implementation of the JAXR API that provides access to a specific registry provider or to a class of registry providers that are based on a common specification.

A JAXR provider implements two main packages:

- `javax.xml.registry`, which consists of the API interfaces and classes that define the registry access interface.
- `javax.xml.registry.infomodel`, which consists of interfaces that define the information model for JAXR. These interfaces define the types of objects that reside in a registry and how they relate to each other. The basic interface in this package is the `RegistryObject` interface. Its subinterfaces include `Organization`, `Service`, and `ServiceBinding`.

The most basic interfaces in the `javax.xml.registry` package are

- `Connection`. The `Connection` interface represents a client session with a registry provider. The client must create a connection with the JAXR provider in order to use a registry.
- `RegistryService`. The client obtains a `RegistryService` object from its connection. The `RegistryService` object in turn enables the client to obtain the interfaces it uses to access the registry.

The primary interfaces, also part of the `javax.xml.registry` package, are

- `BusinessQueryManager`, which allows the client to search a registry for information in accordance with the `javax.xml.registry.infomodel` interfaces. An optional interface, `DeclarativeQueryManager`, allows the client to use SQL syntax for queries. (The implementation of JAXR in the Java WSDP does not implement `DeclarativeQueryManager`.)
- `BusinessLifeCycleManager`, which allows the client to modify the information in a registry by either saving it (updating it) or deleting it.

When an error occurs, JAXR API methods throw a `JAXRException` or one of its subclasses.

Many methods in the JAXR API use a `Collection` object as an argument or a returned value. Using a `Collection` object allows operations on several registry objects at a time.

Figure 13–1 illustrates the architecture of JAXR. In the Java WSDP, a JAXR client uses the capability level 0 interfaces of the JAXR API to access the JAXR provider. The JAXR provider in turn accesses a registry. The Java WSDP supplies a JAXR provider for UDDI registries.

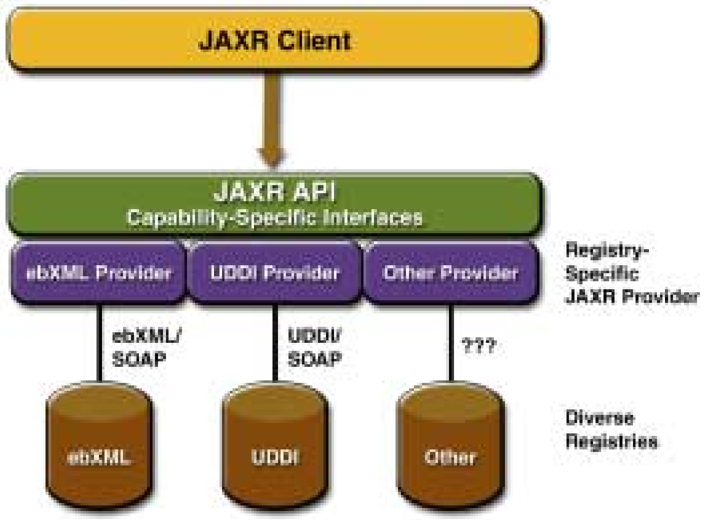


Figure 13–1 JAXR Architecture

Implementing a JAXR Client

This section describes the basic steps to follow in order to implement a JAXR client that can perform queries and updates to a UDDI registry. A JAXR client is a client program that can access registries using the JAXR API.

This tutorial does not describe how to implement a JAXR provider. A JAXR provider provides an implementation of the JAXR specification that allows access to an existing registry provider, such as a UDDI or ebXML registry. The implementation of JAXR in the Java WSDP itself is an example of a JAXR provider.

This tutorial includes several client examples, which are described in Running the Client Examples (page 562).

The JAXR release also includes several sample JAXR clients, the most complete of which is a Registry Browser that includes a graphical user interface (GUI). For details on using this browser, see Registry Browser (page 839).

Establishing a Connection

The first task a JAXR client must complete is to establish a connection to a registry.

Preliminaries: Getting Access to a Registry

Any user of a JAXR client may perform queries on a registry. In order to add data to the registry or to update registry data, however, a user must obtain permission from the registry to access it. To register with one of the public UDDI version 2 registries, go to one of the following Web sites and follow the instructions:

- `http://uddi.microsoft.com/` (Microsoft)
- `http://uddi.ibm.com/testregistry/registry.html` (IBM)
- `http://udditest.sap.com/` (SAP)

These UDDI version 2 registries are intended for testing purposes. When you register, you will obtain a user name and password. You will specify this user name and password for some of the JAXR client example programs.

Note: The JAXR API has been tested with the Microsoft and IBM registries, but not with the SAP registry.

Creating or Looking Up a Connection Factory

A client creates a connection from a connection factory. A JAXR provider may supply one or more preconfigured connection factories that clients can obtain by looking them up using the Java Naming and Directory Interface™ (JNDI) API.

At this release of the Java WSDP, JAXR does not supply preconfigured connection factories. Instead, a client creates an instance of the abstract class `ConnectionFactory`:

```
import javax.xml.registry.*;
...
ConnectionFactory connFactory =
    ConnectionFactory.newInstance();
```

Creating a Connection

To create a connection, a client first creates a set of properties that specify the URL or URLs of the registry or registries being accessed. For example, the following code provides the URLs of the query service and publishing service for the IBM test registry. (There should be no line break in the strings.)

```
Properties props = new Properties();
props.setProperty("javax.xml.registry.queryManagerURL",
    "http://uddi.ibm.com/testregistry/inquiryapi");
props.setProperty("javax.xml.registry.lifeCycleManagerURL",
    "https://uddi.ibm.com/testregistry/protect/publishapi");
```

With the Java WSDP implementation of JAXR, if the client is accessing a registry that is outside a firewall, it must also specify proxy host and port information for the network on which it is running. For queries it may need to specify only the HTTP proxy host and port; for updates it must specify the HTTPS proxy host and port.

```
props.setProperty("com.sun.xml.registry.http.proxyHost",
    "myhost.mydomain");
props.setProperty("com.sun.xml.registry.http.proxyPort",
    "8080");
props.setProperty("com.sun.xml.registry.https.proxyHost",
    "myhost.mydomain");
props.setProperty("com.sun.xml.registry.https.proxyPort",
    "8080");
```

The client then sets the properties for the connection factory and creates the connection:

```
connFactory.setProperties(props);
Connection connection = connFactory.createConnection();
```

The `makeConnection` method in the sample programs shows the steps used to create a JAXR connection.

Setting Connection Properties

The implementation of JAXR in the Java WSDP allows you to set a number of properties on a JAXR connection. Some of these are standard properties defined in the JAXR specification. Other properties are specific to the implementation of

JAXR in the Java WSDP. Table 13–1 and Table 13–2 list and describe these properties.

Table 13–1 Standard JAXR Connection Properties

Property Name and Description	Data Type	Default Value
`javax.xml.registry.queryManagerURL` Specifies the URL of the query manager service within the target registry provider	String	None
`javax.xml.registry.lifeCycleManagerURL` Specifies the URL of the life cycle manager service within the target registry provider (for registry updates)	String	Same as the specified `queryManagerURL` value
`javax.xml.registry.semanticEquivalences` Specifies semantic equivalences of concepts as one or more tuples of the ID values of two equivalent concepts separated by a comma; the tuples are separated by vertical bars: `id1,id2|id3,id4`	String	None
`javax.xml.registry.security.authenticationMethod` Provides a hint to the JAXR provider on the authentication method to be used for authenticating with the registry provider	String	None; `UDDI_GET_AUTHTOKEN` is the only supported value
`javax.xml.registry.uddi.maxRows` The maximum number of rows to be returned by find operations. Specific to UDDI providers	Integer	None
`javax.xml.registry.postalAddressScheme` The ID of a `ClassificationScheme` to be used as the default postal address scheme. See Specifying Postal Addresses (page 560) for an example	String	None

Table 13–2 Implementation-Specific JAXR Connection Properties

Property Name and Description	Data Type	Default Value
`com.sun.xml.registry.http.proxyHost` Specifies the HTTP proxy host to be used for accessing external registries. If you specified a proxy host and port when you installed the Java WSDP, the values you specified are in the file `<`*`JWSDP_HOME`*`>/conf/jwsdp.properties`.	String	Proxy host value specified in `<`*`JWSDP_HOME`*`>/conf/jwsdp.properties`
`com.sun.xml.registry.http.proxyPort` Specifies the HTTP proxy port to be used for accessing external registries; usually 8080	String	Proxy port value specified in `<`*`JWSDP_HOME`*`>/conf/jwsdp.properties`
`com.sun.xml.registry.https.proxyHost` Specifies the HTTPS proxy host to be used for accessing external registries	String	Same as HTTP proxy host value
`com.sun.xml.registry.https.proxyPort` Specifies the HTTPS proxy port to be used for accessing external registries; usually 8080	String	Same as HTTP proxy port value
`com.sun.xml.registry.http.proxyUserName` Specifies the user name for the proxy host for HTTP proxy authentication, if one is required	String	None
`com.sun.xml.registry.http.proxyPassword` Specifies the password for the proxy host for HTTP proxy authentication, if one is required	String	None
`com.sun.xml.registry.useCache` Tells the JAXR implementation to look for registry objects in the cache first and then to look in the registry if not found	Boolean, passed in as String	True

Table 13–2 Implementation-Specific JAXR Connection Properties

Property Name and Description	Data Type	Default Value
`com.sun.xml.registry.useSOAP` Tells the JAXR implementation to use Apache SOAP rather than the Java API for XML Messaging; may be useful for debugging	Boolean, passed in as String	False

You can set these properties as follows:

- Most of these properties must be set in a JAXR client program. For example:

```
Properties props = new Properties();
props.setProperty("javax.xml.registry.queryManagerURL",
    "http://uddi.ibm.com/testregistry/inquiryapi");
props.setProperty("javax.xml.registry.lifeCycleManagerURL",
    "https://uddi.ibm.com/testregistry/protect/publishapi");
ConnectionFactory factory = ConnectionFactory.newInstance();
factory.setProperties(props);
connection = factory.createConnection();
```

- The `postalAddressScheme`, `useCache`, and `useSOAP` properties may be set in a `<sysproperty>` tag in a `build.xml` file for the `Ant` tool. For example:

  ```
  <sysproperty key="useSOAP" value="true"/>
  ```

  These properties may also be set with the `-D` option on the `java` command line.

An additional system property specific to the implementation of JAXR in the Java WSDP is `com.sun.xml.registry.userTaxonomyFilenames`. For details on using this property, see Defining a Taxonomy (page 557).

Obtaining and Using a RegistryService Object

After creating the connection, the client uses the connection to obtain a RegistryService object and then the interface or interfaces it will use:

```
RegistryService rs = connection.getRegistryService();
BusinessQueryManager bqm = rs.getBusinessQueryManager();
BusinessLifeCycleManager blcm =
    rs.getBusinessLifeCycleManager();
```

Typically, a client obtains both a BusinessQueryManager object and a BusinessLifeCycleManager object from the RegistryService object. If it is using the registry for simple queries only, it may need to obtain only a BusinessQueryManager object.

Querying a Registry

The simplest way for a client to use a registry is to query it for information about the organizations that have submitted data to it. The BusinessQueryManager interface supports a number of find methods that allow clients to search for data using the JAXR information model. Many of these methods return a BulkResponse (a collection of objects) that meets a set of criteria specified in the method arguments. The most useful of these methods are:

- findOrganizations, which returns a list of organizations that meet the specified criteria—often a name pattern or a classification within a classification scheme
- findServices, which returns a set of services offered by a specified organization
- findServiceBindings, which returns the service bindings (information about how to access the service) that are supported by a specified service

The JAXRQuery program illustrates how to query a registry by organization name and display the data returned. The JAXRQueryByNAICSClassification and JAXRQueryByWSDLClassification programs illustrate how to query a registry using classifications. All JAXR providers support at least the following taxonomies for classifications:

- The North American Industry Classification System (NAICS). See http://www.census.gov/epcd/www/naics.html for details.
- The Universal Standard Products and Services Classification (UNSPSC). See http://www.eccma.org/unspsc/ for details.

- The ISO 3166 country codes classification system maintained by the International Organization for Standardization (ISO). See `http://www.iso.org/iso/en/prods-services/iso3166ma/index.html` for details.

The following sections describe how to perform some common queries.

Finding Organizations by Name

To search for organizations by name, you normally use a combination of find qualifiers (which affect sorting and pattern matching) and name patterns (which specify the strings to be searched). The `findOrganizations` method takes a collection of `findQualifier` objects as its first argument and a collection of `namePattern` objects as its second argument. The following fragment shows how to find all the organizations in the registry whose names begin with a specified string, `qString`, and to sort them in alphabetical order.

```
// Define find qualifiers and name patterns
Collection findQualifiers = new ArrayList();
findQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);
Collection namePatterns = new ArrayList();
namePatterns.add(qString);

// Find using the name
BulkResponse response =
    bqm.findOrganizations(findQualifiers,
        namePatterns, null, null, null, null);
Collection orgs = response.getCollection();
```

A client can use percent signs (%) to specify that the query string can occur anywhere within the organization name. For example, the following code fragment performs a case-sensitive search for organizations whose names contain `qString`:

```
Collection findQualifiers = new ArrayList();
findQualifiers.add(FindQualifier.CASE_SENSITIVE_MATCH);
Collection namePatterns = new ArrayList();
namePatterns.add("%" + qString + "%");

// Find orgs with name containing qString
BulkResponse response =
    bqm.findOrganizations(findQualifiers, namePatterns, null,
        null, null, null);
Collection orgs = response.getCollection();
```

Finding Organizations by Classification

To find organizations by classification, you need to establish the classification within a particular classification scheme and then specify the classification as an argument to the findOrganizations method.

The following code fragment finds all organizations that correspond to a particular classification within the NAICS taxonomy. (You can find the NAICS codes at http://www.census.gov/epcd/naics/naicscod.txt and also in the file <*JWSDP_HOME*>/docs/jaxr/taxonomies/naics.xml.)

```
ClassificationScheme cScheme =
    bqm.findClassificationSchemeByName(null,
        "ntis-gov:naics");
Classification classification =
    blcm.createClassification(cScheme,
        "Snack and Nonalcoholic Beverage Bars", "722213");
Collection classifications = new ArrayList();
classifications.add(classification);
// make JAXR request
BulkResponse response = bqm.findOrganizations(null,
    null, classifications, null, null, null);
Collection orgs = response.getCollection();
```

You can also use classifications to find organizations that offer services based on technical specifications that take the form of WSDL (Web Services Description Language) documents. In JAXR, a concept is used as a proxy to hold the information about a specification. The steps are a little more complicated than in the previous example, because the client must find the specification concepts first, then the organizations that use those concepts.

The following code fragment finds all the WSDL specification instances used within a given registry. You can see that the code is similar to the NAICS query code except that it ends with a call to findConcepts instead of findOrganizations.

```
String schemeName = "uddi-org:types";
ClassificationScheme uddiOrgTypes =
    bqm.findClassificationSchemeByName(null, schemeName);

/*
 * Create a classification, specifying the scheme
 *  and the taxonomy name and value defined for WSDL
 *  documents by the UDDI specification.
 */
```

```
Classification wsdlSpecClassification =
blcm.createClassification(uddiOrgTypes,
    "wsdlSpec", "wsdlSpec");

Collection classifications = new ArrayList();
classifications.add(wsdlSpecClassification);

// Find concepts
BulkResponse br = bqm.findConcepts(null, null,
    classifications, null, null);
```

To narrow the search, you could use other arguments of the `findConcepts` method (search qualifiers, names, external identifiers, or external links).

The next step is to go through the concepts, find the WSDL documents they correspond to, and display the organizations that use each document:

```
// Display information about the concepts found
Collection specConcepts = br.getCollection();
Iterator iter = specConcepts.iterator();
if (!iter.hasNext()) {
   System.out.println("No WSDL specification concepts found");
} else {
    while (iter.hasNext()) {
    Concept concept = (Concept) iter.next();

    String name = getName(concept);

    Collection links = concept.getExternalLinks();
    System.out.println("\nSpecification Concept:\n\tName: " +
        name + "\n\tKey: " +
        concept.getKey().getId() +
        "\n\tDescription: " +
        getDescription(concept));
    if (links.size() > 0) {
        ExternalLink link =
            (ExternalLink) links.iterator().next();
        System.out.println("\tURL of WSDL document: '" +
        link.getExternalURI() + "'");
    }

    // Find organizations that use this concept
    Collection specConcepts1 = new ArrayList();
    specConcepts1.add(concept);
    br = bqm.findOrganizations(null, null, null,
        specConcepts1, null, null);
        // Display information about organizations
        ...
    }
```

If you find an organization that offers a service you wish to use, you can invoke the service using the JAX-RPC API.

Finding Services and ServiceBindings

After a client has located an organization, it can find that organization's services and the service bindings associated with those services.

```
    Iterator orgIter = orgs.iterator();
    while (orgIter.hasNext()) {
        Organization org = (Organization) orgIter.next();
        Collection services = org.getServices();
        Iterator svcIter = services.iterator();
        while (svcIter.hasNext()) {
            Service svc = (Service) svcIter.next();
            Collection serviceBindings =
                svc.getServiceBindings();
            Iterator sbIter = serviceBindings.iterator();
            while (sbIter.hasNext()) {
                ServiceBinding sb =
                    (ServiceBinding) sbIter.next();
            }
        }
    }
```

Managing Registry Data

If a client has authorization to do so, it can submit data to a registry, modify it, and remove it. It uses the `BusinessLifeCycleManager` interface to perform these tasks.

Registries usually allow a client to modify or remove data only if the data is being modified or removed by the same user who first submitted the data.

Getting Authorization from the Registry

Before it can submit data, the client must send its user name and password to the registry in a set of credentials. The following code fragment shows how to do this.

```
String username = "myUserName";
String password = "myPassword";

// Get authorization from the registry
PasswordAuthentication passwdAuth =
    new PasswordAuthentication(username,
        password.toCharArray());

Set creds = new HashSet();
creds.add(passwdAuth);
connection.setCredentials(creds);
```

Creating an Organization

The client creates the organization and populates it with data before saving it.

An `Organization` object is one of the more complex data items in the JAXR API. It normally includes the following:

- A `Name` object
- A `Description` object
- A `Key` object, representing the ID by which the organization is known to the registry. This key is created by the registry, not by the user, and is returned after the organization is submitted to the registry.
- A `PrimaryContact` object, which is a `User` object that refers to an authorized user of the registry. A `User` object normally includes a `PersonName` object and collections of `TelephoneNumber`, `EmailAddress`, and/or `PostalAddress` objects.
- A collection of `Classification` objects
- `Service` objects and their associated `ServiceBinding` objects

For example, the following code fragment creates an organization and specifies its name, description, and primary contact. When a client creates an organization, it does not include a key; the registry returns the new key when it accepts the newly created organization. The `blcm` object in this code fragment is the `BusinessLifeCycleManager` object returned in Obtaining and Using a Registry-

Service Object (page 547). An `InternationalString` object is used for string values that may need to be localized.

```
// Create organization name and description
Organization org =
    blcm.createOrganization("The Coffee Break");
InternationalString s =
    blcm.createInternationalString("Purveyor of " +
        "the finest coffees. Established 1895");
org.setDescription(s);

// Create primary contact, set name
User primaryContact = blcm.createUser();
PersonName pName = blcm.createPersonName("Jane Doe");
primaryContact.setPersonName(pName);

// Set primary contact phone number
TelephoneNumber tNum = blcm.createTelephoneNumber();
tNum.setNumber("(800) 555-1212");
Collection phoneNums = new ArrayList();
phoneNums.add(tNum);
primaryContact.setTelephoneNumbers(phoneNums);

// Set primary contact email address
EmailAddress emailAddress =
    blcm.createEmailAddress("jane.doe@TheCoffeeBreak.com");
Collection emailAddresses = new ArrayList();
emailAddresses.add(emailAddress);
primaryContact.setEmailAddresses(emailAddresses);

// Set primary contact for organization
org.setPrimaryContact(primaryContact);
```

Adding Classifications

Organizations commonly belong to one or more classifications based on one or more classification schemes (taxonomies). To establish a classification for an organization using a taxonomy, the client first locates the taxonomy it wants to use. It uses the `BusinessQueryManager` to find the taxonomy. The `findClassificationSchemeByName` method takes a set of `FindQualifier` objects as its first argument, but this argument can be null.

```
// Set classification scheme to NAICS
ClassificationScheme cScheme =
   bqm.findClassificationSchemeByName(null, "ntis-gov:naics");
```

The client then creates a classification using the classification scheme and a concept (a taxonomy element) within the classification scheme. For example, the following code sets up a classification for the organization within the NAICS taxonomy. The second and third arguments of the `createClassification` method are the name and value of the concept.

```
// Create and add classification
Classification classification =
    blcm.createClassification(cScheme,
        "Snack and Nonalcoholic Beverage Bars", "722213");
Collection classifications = new ArrayList();
classifications.add(classification);
org.addClassifications(classifications);
```

Services also use classifications, so you can use similar code to add a classification to a `Service` object.

Adding Services and Service Bindings to an Organization

Most organizations add themselves to a registry in order to offer services, so the JAXR API has facilities to add services and service bindings to an organization.

Like an `Organization` object, a `Service` object has a name and a description. Also like an `Organization` object, it has a unique key that is generated by the registry when the service is registered. It may also have classifications associated with it.

A service also commonly has service bindings, which provide information about how to access the service. A `ServiceBinding` object normally has a description, an access URI, and a specification link, which provides the linkage between a service binding and a technical specification that describes how to use the service using the service binding.

The following code fragment shows how to create a collection of services, add service bindings to a service, then add the services to the organization. It specifies an access URI but not a specification link. Because the access URI is not real and because JAXR by default checks for the validity of any published URI, the binding sets its `validateURI` property to false.

```
// Create services and service
Collection services = new ArrayList();
Service service = blcm.createService("My Service Name");
InternationalString is =
  blcm.createInternationalString("My Service Description");
service.setDescription(is);

// Create service bindings
Collection serviceBindings = new ArrayList();
ServiceBinding binding = blcm.createServiceBinding();
is = blcm.createInternationalString("My Service Binding " +
    "Description");
binding.setDescription(is);
// allow us to publish a bogus URL without an error
binding.setValidateURI(false);
binding.setAccessURI("http://TheCoffeeBreak.com:8080/sb/");
serviceBindings.add(binding);

// Add service bindings to service
service.addServiceBindings(serviceBindings);

// Add service to services, then add services to organization
services.add(service);
org.addServices(services);
```

Saving an Organization

The primary method a client uses to add or modify organization data is the `saveOrganizations` method, which creates one or more new organizations in a registry if they did not exist previously. If one of the organizations exists but some of the data have changed, the `saveOrganizations` method updates and replaces the data.

After a client populates an organization with the information it wants to make public, it saves the organization. The registry returns the key in its response, and the client retrieves it.

```
// Add organization and submit to registry
// Retrieve key if successful
Collection orgs = new ArrayList();
orgs.add(org);
BulkResponse response = blcm.saveOrganizations(orgs);
Collection exceptions = response.getException();
```

```
if (exceptions == null) {
    System.out.println("Organization saved");

    Collection keys = response.getCollection();
    Iterator keyIter = keys.iterator();
    if (keyIter.hasNext()) {
         javax.xml.registry.infomodel.Key orgKey =
            (javax.xml.registry.infomodel.Key) keyIter.next();
        String id = orgKey.getId();
        System.out.println("Organization key is " + id);
        org.setKey(orgKey);
    }
}
```

Removing Data from the Registry

A registry allows you to remove from the registry any data that you have submitted to it. You use the key returned by the registry as an argument to one of the `BusinessLifeCycleManager` delete methods: `deleteOrganizations`, `deleteServices`, `deleteServiceBindings`, and others.

The `JAXRDelete` sample program deletes the organization created by the `JAXRPublish` program. It deletes the organization that corresponds to a specified key string and then displays the key again so that the user can confirm that it has deleted the correct one.

```
String id = key.getId();
System.out.println("Deleting organization with id " + id);
Collection keys = new ArrayList();
keys.add(key);
BulkResponse response = blcm.deleteOrganizations(keys);
Collection exceptions = response.getException();
if (exceptions == null) {
    System.out.println("Organization deleted");
    Collection retKeys = response.getCollection();
    Iterator keyIter = retKeys.iterator();
    javax.xml.registry.infomodel.Key orgKey = null;
    if (keyIter.hasNext()) {
        orgKey =
            (javax.xml.registry.infomodel.Key) keyIter.next();
        id = orgKey.getId();
        System.out.println("Organization key was " + id);
    }
}
```

A client can use a similar mechanism to delete services and service bindings.

Using Taxonomies in JAXR Clients

In the JAXR API, a taxonomy is represented by a `ClassificationScheme` object.

This section describes how to use the implementation of JAXR in the Java WSDP:

- To define your own taxonomies
- To specify postal addresses for an organization

Defining a Taxonomy

The JAXR specification requires a JAXR provider to be able to add user-defined taxonomies for use by JAXR clients. The mechanisms clients use to add and administer these taxonomies are implementation-specific.

The implementation of JAXR in the Java WSDP uses a simple file-based approach to provide taxonomies to the JAXR client. These files are read at run time, when the JAXR provider starts up.

The taxonomy structure for the Java WSDP is defined by the JAXR Predefined Concepts DTD, which is declared both in the file `jaxrconcepts.dtd` and, in XML schema form, in the file `jaxrconcepts.xsd`. The file `jaxrconcepts.xml` contains the taxonomies for the implementation of JAXR in the Java WSDP. All these files are contained in the `<`*`JWSDP_HOME`*`>/common/lib/jaxr-ri.jar` file, but you can find copies of them in the directory `<`*`JWSDP_HOME`*`>/docs/jaxr/taxonomies`. This directory also contains copies of the XML files that the implementation of JAXR in the Java WSDP uses to define the well-known taxonomies that it uses: `naics.xml`, `iso3166.xml`, and `unspsc.xml`. You may use all of these as examples of how to construct a taxonomy XML file.

The entries in the `jaxrconcepts.xml` file look like this:

```
<PredefinedConcepts>
<JAXRClassificationScheme id="schId" name="schName">
<JAXRConcept id="schId/conCode" name="conName"
parent="parentId" code="conCode"></JAXRConcept>
...
</JAXRClassificationScheme>
</PredefinedConcepts>
```

The taxonomy structure is a containment-based structure. The element `PredefinedConcepts` is the root of the structure and must be present. The `JAXRClassificationScheme` element is the parent of the structure, and the `JAXRConcept` elements are children and grandchildren. A `JAXRConcept` element may have children, but it is not required to do so.

In all element definitions, attribute order and case are significant.

To add a user-defined taxonomy, follow these steps.

1. Publish the `JAXRClassificationScheme` element for the taxonomy as a `ClassificationScheme` object in the registry that you will be accessing. For example, you can publish the `ClassificationScheme` object to the Java WSDP Registry Server. In order to publish a `ClassificationScheme` object, you must set its name. You also give the scheme a classification within a known classification scheme such as `uddi-org:types`. In the following code fragment, the name is the first argument of the `LifeCycleManager.createClassificationScheme` method call.

```
ClassificationScheme cScheme =
    blcm.createClassificationScheme("MyScheme",
        "A Classification Scheme");
ClassificationScheme uddiOrgTypes =
    bqm.findClassificationSchemeByName(null,
        "uddi-org:types");
if (uddiOrgTypes != null) {
    Classification classification =
        blcm.createClassification(uddiOrgTypes,
            "postalAddress", "categorization" );
    postalScheme.addClassification(classification);
    ExternalLink externalLink =
        blcm.createExternalLink(
            "http://www.mycom.com/myscheme.html",
            "My Scheme");
    postalScheme.addExternalLink(externalLink);
    Collection schemes = new ArrayList();
    schemes.add(cScheme);
    BulkResponse br =
        blcm.saveClassificationSchemes(schemes);
}
```

The `BulkResponse` object returned by the `saveClassificationSchemes` method contains the key for the classification scheme, which you need to retrieve:

```
if (br.getStatus() == JAXRResponse.STATUS_SUCCESS) {
    System.out.println("Saved ClassificationScheme");
    Collection schemeKeys = br.getCollection();
    Iterator keysIter = schemeKeys.iterator();
    while (keysIter.hasNext()) {
        javax.xml.registry.infomodel.Key key =
            (javax.xml.registry.infomodel.Key)
                keysIter.next();
        System.out.println("The postalScheme key is " +
            key.getId());
        System.out.println("Use this key as the scheme“ +
            “ uuid in the taxonomy file");
    }
}
```

2. In an XML file, define a taxonomy structure that is compliant with the JAXR Predefined Concepts DTD. Enter the `ClassificationScheme` element in your taxonomy XML file by specifying the returned key ID value as the `id` attribute and the name as the `name` attribute. For the code fragment above, for example, the opening tag for the `JAXRClassificationScheme` element looks something like this (all on one line):

   ```
   <JAXRClassificationScheme
   id="uuid:nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn"
   name="MyScheme">
   ```

   The `ClassificationScheme id` must be a UUID.

3. Enter each `JAXRConcept` element in your taxonomy XML file by specifying the following four attributes, in this order:

   a. `id` is the `JAXRClassificationScheme id` value, followed by a `/` separator, followed by the code of the `JAXRConcept` element

   b. `name` is the name of the `JAXRConcept` element

   c. `parent` is the immediate parent `id` (either the `ClassificationScheme id` or that of the parent `JAXRConcept`)

   d. `code` is the `JAXRConcept` element code value

   The first `JAXRConcept` element in the `naics.xml` file looks like this (all on one line):

   ```
   <JAXRConcept
   id="uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2/11"
   name="Agriculture, Forestry, Fishing and Hunting"
   ```

```
parent="uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"
code="11"></JAXRConcept>
```

4. To add the user-defined taxonomy structure to the JAXR provider, specify the system property `com.sun.xml.registry.userTaxonomyFilenames` when you run your client program. The command line (all on one line) would look like this. A vertical bar (`|`) is the file separator.

   ```
   java myProgram -DuserTaxonomyFilenames=
   c:\myfile\xxx.xml|c:\myfile\xxx2.xml
   ```

   You can use a `<sysproperty>` tag to set this property in a `build.xml` file. Or, in your program, you can set the property as follows:

   ```
   System.setProperty
   ("com.sun.xml.registry.userTaxonomyFilenames",
       "c:\myfile\xxx.xml|c:\myfile\xxx2.xml");
   ```

Specifying Postal Addresses

The JAXR specification defines a postal address as a structured interface with attributes for street, city, country, and so on. The UDDI specification, on the other hand, defines a postal address as a free-form collection of address lines, each of which may also be assigned a meaning. To map the JAXR `PostalAddress` format to a known UDDI address format, you specify the UDDI format as a `ClassificationScheme` object and then specify the semantic equivalences between the concepts in the UDDI format classification scheme and the comments in the JAXR `PostalAddress` classification scheme. The JAXR `PostalAddress` classification scheme is provided by the implementation of JAXR in the Java WSDP.

In the JAXR API, a `PostalAddress` object has the fields `streetNumber`, `street`, `city`, `state`, `postalCode` and `country`. In the implementation of JAXR in the Java WSDP, these are predefined concepts in the `jaxrconcepts.xml` file, within the `ClassificationScheme` named `PostalAddressAttributes`.

To specify the mapping between the JAXR postal address format and another format, you need to set two connection properties:

- The `javax.xml.registry.postalAddressScheme` property, which specifies a postal address classification scheme for the connection
- The `javax.xml.registry.semanticEquivalences` property, which specifies the semantic equivalences between the JAXR format and the other format

For example, suppose you want to use a scheme that has been published to the IBM registry with the known UUID `uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b`. This scheme already exists in the `jaxrconcepts.xml` file under the name `IBMDefaultPostalAddressAttributes`.

```
<JAXRClassificationScheme
id="uuid:6EAF4B50-4196-11D6-9E2B-000629DC0A2B"
name="IBMDefaultPostalAddressAttributes">
```

First, you specify the postal address scheme using the `id` value from the `JAXR-ClassificationScheme` element (the UUID). Case does not matter:

```
props.setProperty("javax.xml.registry.postalAddressScheme",
  "uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b");
```

Next, you specify the mapping from the `id` of each `JAXRConcept` element in the default JAXR postal address scheme to the `id` of its counterpart in the IBM scheme:

```
props.setProperty("javax.xml.registry.semanticEquivalences",
  "urn:uuid:PostalAddressAttributes/StreetNumber," +
  "urn:uuid:6eaf4b50-4196-11d6-9e2b-
000629dc0a2b/StreetAddressNumber|" +
  "urn:uuid:PostalAddressAttributes/Street," +
  "urn:uuid:6eaf4b50-4196-11d6-9e2b-
000629dc0a2b/StreetAddress|" +
  "urn:uuid:PostalAddressAttributes/City," +
  "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/City|" +
  "urn:uuid:PostalAddressAttributes/State," +
  "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/State|" +
  "urn:uuid:PostalAddressAttributes/PostalCode," +
  "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/ZipCode|" +
  "urn:uuid:PostalAddressAttributes/Country," +
  "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/Country");
```

After you create the connection using these properties, you can create a postal address and assign it to the primary contact of the organization before you publish the organization:

```
String streetNumber = "99";
String street = "Imaginary Ave. Suite 33";
String city = "Imaginary City";
String state = "NY";
String country = "USA";
String postalCode = "00000";
String type = "";
PostalAddress postAddr =
  blcm.createPostalAddress(streetNumber, street, city, state,
    country, postalCode, type);
Collection postalAddresses = new ArrayList();
postalAddresses.add(postAddr);
primaryContact.setPostalAddresses(postalAddresses);
```

A JAXR query can then retrieve the postal address using PostalAddress methods, if the postal address scheme and semantic equivalences for the query are the same as those specified for the publication. To retrieve postal addresses when you do not know what postal address scheme was used to publish them, you can retrieve them as a collection of Slot objects. The JAXRQueryPostal.java sample program shows how to do this.

In general, you can create a user-defined postal address taxonomy for any postalAddress tModels that use the well-known categorization in the uddi-org:types taxonomy, which has the tModel UUID uuid:c1acf26d-9672-4404-9d70-39b756e62ab4 with a value of postalAddress. You can retrieve the tModel overviewDoc, which points to the technical detail for the specification of the scheme, where the taxonomy structure definition can be found. (The JAXR equivalent of an overviewDoc is an ExternalLink.)

Running the Client Examples

The simple client programs provided with this tutorial can be run from the command line. You can modify them to suit your needs. They allow you to specify the IBM registry, the Microsoft registry, or the Registry Server for queries and updates; you can specify any other UDDI version 2 registry.

The client examples, in the <*JWSDP_HOME*>/docs/tutorial/examples/jaxr directory, are as follows:

- JAXRQuery.java shows how to search a registry for organizations
- JAXRQueryByNAICSClassification.java shows how to search a registry using a common classification scheme
- JAXRQueryByWSDLClassification.java shows how to search a registry for Web services that describe themselves by means of a WSDL document
- JAXRPublish.java shows how to publish an organization to a registry
- JAXRDelete.java shows how to remove an organization from a registry
- JAXRSaveClassificationScheme.java shows how to publish a classification scheme (specifically, a postal address scheme) to a registry
- JAXRPublishPostal.java shows how to publish an organization with a postal address for its primary contact
- JAXRQueryPostal.java shows how to retrieve postal address data from an organization
- JAXRDeleteScheme.java shows how to delete a classification scheme from a registry
- JAXRGetMyObjects.java lists all the objects that you own in a registry

The <*JWSDP_HOME*>/docs/tutorial/examples/jaxr directory also contains:

- A build.xml file for the examples
- A JAXRExamples.properties file that supplies string values used by the sample programs
- A file called postalconcepts.xml that you use with the postal address examples

Before You Compile the Examples

Before you compile the examples, edit the file JAXRExamples.properties as follows. (See Using JAXR to Access the Registry Server, page 830 for details on editing this file to access the Registry Server.)

1. Edit the following lines in the JAXRExamples.properties file to specify the registry you wish to access. For both the queryURL and the publishURL assignments, comment out all but the registry you wish to access. The default is the Registry Server, so if you will be using the Registry Server you do not need to change this section.

```
## Uncomment one pair of query and publish URLs.
## IBM:
#query.url=http://uddi.ibm.com/testregistry/inquiryapi
#publish.url=https://uddi.ibm.com/testregistry/protect/
publishapi
## Microsoft:
#query.url=http://uddi.microsoft.com/inquire
#publish.url=https://uddi.microsoft.com/publish
## Registry Server:
query.url=http://localhost:8080/registry-server/
RegistryServerServlet
publish.url=http://localhost:8080/registry-server/
RegistryServerServlet
```

The IBM and Microsoft registries both have a considerable amount of data in them that you can perform queries on. Moreover, you do not have to register if you are only going to perform queries.

We have not included the URL of the SAP registry; feel free to add it.

If you want to publish to any of the public registries, the registration process for obtaining access to them is not difficult (see Preliminaries: Getting Access to a Registry, page 542). Each of them, however, allows you to have only one organization registered at a time. If you publish an organization to one of them, you must delete it before you can publish another. Since the organization that the `JAXRPublish` example publishes is fictitious, you will want to delete it immediately anyway. (It is particularly important to delete such organizations promptly, because the public registries replicate each other's data, and your fictitious organization may appear in a registry that is not the one you published it to and from which you therefore cannot delete it.)

The Registry Server gives you more freedom to experiment with JAXR. You can publish as many organizations to it as you wish. However, this registry comes with an empty database, so you must publish organizations to it yourself before you can perform queries on the data.

2. Edit the following lines in the `JAXRExamples.properties` file to specify the user name and password you obtained when you registered with the registry. The default is the Registry Server default password.

```
## Specify username and password if needed
## testuser/testuser are defaults for Registry Server
```

```
registry.username=testuser
registry.password=testuser
```

3. If you will be using a public registry, edit the following lines in the `JAXR-Examples.properties` file, which contain empty strings for the proxy hosts, to specify your own proxy settings. The proxy host is the system on your network through which you access the Internet; you usually specify it in your Internet browser settings. You can leave this value empty to use the Registry Server.

   ```
   ## HTTP and HTTPS proxy host and port;
   ##   ignored by Registry Server
   http.proxyHost=
   http.proxyPort=8080
   https.proxyHost=
   https.proxyPort=8080
   ```

   The proxy ports have the value 8080, which is the usual one; change this string if your proxy uses a different port.

   For a public registry, your entries usually follow this pattern:

   ```
   http.proxyHost=proxyhost.mydomain
   http.proxyPort=8080
   https.proxyHost=proxyhost.mydomain
   https.proxyPort=8080
   ```

4. Feel free to change any of the organization data in the remainder of the file. This data is used by the publishing and postal address examples.

You can edit the `JAXRExamples.properties` file at any time. When you run the client examples, they use the latest version of the file.

Compiling the Examples

To compile the programs, go to the `<`*`JWSDP_HOME`*`>/docs/tutorial/examples/jaxr` directory. A `build.xml` file allows you to use the command

```
ant build
```

to compile all the examples. The `Ant` tool creates a subdirectory called `build` and places the class files there.

You will notice that the classpath setting in the `build.xml` file includes the contents of several directories. All JAXR client examples require this classpath setting.

Running the Examples

Some of the `build.xml` targets for running the examples contain commented-out `<sysproperty>` tags that set the JAXR logging level to debug and set other connection properties. These tags are provided to illustrate how to specify connection properties. Feel free to modify or delete these tags.

If you are running the examples with the Registry Server, start Tomcat. See Starting the Registry Server (page 830) for details. You do not need to start Tomcat in order to run the examples against public registries.

Running the JAXRPublish Example

To run the `JAXRPublish` program, use the `run-publish` target with no command line arguments:

```
ant run-publish
```

The program output displays the string value of the key of the new organization, which is named “The Coffee Break.”

After you run the `JAXRPublish` program but before you run `JAXRDelete`, you can run `JAXRQuery` to look up the organization you published. You can also use the Registry Browser to search for it.

Running the JAXRQuery Example

To run the `JAXRQuery` example, use the `Ant` target `run-query`. Specify a `query-string` argument on the command line to search the registry for organizations whose names contain that string. For example, the following command line searches for organizations whose names contain the string “coff” (searching is not case-sensitive):

```
ant run-query -Dquery-string=coff
```

Running the JAXRQueryByNAICSClassification Example

After you run the JAXRPublish program, you can also run the JAXRQueryByNAICSClassification example, which looks for organizations that use the “Snack and Nonalcoholic Beverage Bars” classification, the same one used for the organization created by JAXRPublish. To do so, use the Ant target run-query-naics:

```
ant run-query-naics
```

Running the JAXRDelete Example

To run the JAXRDelete program, specify the key string returned by the JAXRPublish program as input to the run-delete target:

```
ant run-delete -Dkey-string=keyString
```

Running the JAXRQueryByWSDLClassification Example

You can run the JAXRQueryByWSDLClassification example at any time. Use the Ant target run-query-wsdl:

```
ant run-query-wsdl
```

This example returns many results from the public registries and is likely to run for several minutes.

Publishing a Classification Scheme

In order to publish organizations with postal addresses to public registries, you must publish a classification scheme for the postal address first.

To run the JAXRSaveClassificationScheme program, use the target run-save-scheme:

```
ant run-save-scheme
```

The program returns a UUID string, which you will use in the next section.

You do not have to run this program if you are using the Registry Server, because it does not validate these objects.

The public registries allow you to own more than one classification scheme at a time (the limit is usually a total of about 10 classification schemes and concepts put together).

Running the Postal Address Examples

Before you run the postal address examples, open the file `postalconcepts.xml` in an editor. Wherever you see the string `uuid-from-save`, replace it with the UUID string returned by the `run-save-scheme` target. For the registry server, you may use any string that is formatted as a UUID.

For a given registry, you only need to save the classification scheme and edit `postalconcepts.xml` once. After you perform those two steps, you can run the `JAXRPublishPostal` and `JAXRQueryPostal` programs multiple times.

1. Run the `JAXRPublishPostal` program. Notice that in the `build.xml` file, the `run-publish-postal` target contains a `<sysproperty>` tag that sets the `userTaxonomyFilenames` property to the location of the `postalconcepts.xml` file in the current directory:

   ```
   <sysproperty
      key="com.sun.xml.registry.userTaxonomyFilenames"
      value="postalconcepts.xml"/>
   ```

   Specify the string you entered in the `postalconcepts.xml` file as input to the `run-publish-postal` target:

   ```
   ant run-publish-postal -Duuid-string=uuidstring
   ```

   The program output displays the string value of the key of the new organization.

2. Run the `JAXRQueryPostal` program. The `run-query-postal` target contains the same `<sysproperty>` tag as the `run-publish-postal` target.

   As input to the `run-query-postal` target, specify both a `query-string` argument and a `uuid-string` argument on the command line to search the registry for the organization published by the `run-publish-postal` target:

   ```
   ant run-query-postal -Dquery-string=coffee
   -Duuid-string=uuidstring
   ```

The postal address for the primary contact will appear correctly with the JAXR `PostalAddress` methods. Any postal addresses found that use other postal address schemes will appear as `Slot` lines.

3. If you are using a public registry, make sure to follow the instructions in Running the JAXRDelete Example (page 567) to delete the organization you published.

Deleting a Classification Scheme

To delete the classification scheme you published after you have finished using it, run the `JAXRDeleteScheme` program using the `run-delete-scheme` target:

```
ant run-delete-scheme -Duuid-string=uuidstring
```

For a UDDI registry, deleting a classification scheme removes it from the registry logically but not physically. You can no longer use the classification scheme, but it will still be visible if, for example, you call the method `QueryManager.getRegisteredObjects`. Since the public registries allow you to own up to 10 of these objects, this is not likely to be a problem.

Getting a List of Your Registry Objects

To get a list of the objects you own in the registry, both organizations and classification schemes, run the `JAXRGetMyObjects` program by using the `run-get-objects` target:

```
ant run-get-objects
```

Other Targets

To remove the `build` directory and class files, use the command

```
ant clean
```

To obtain a syntax reminder for the targets, use the command

```
ant -projecthelp
```

Further Information

For more information about JAXR, registries, and Web services, see the following:

- Java Specification Request (JSR) 93: JAXR 1.0:
  `http://jcp.org/jsr/detail/093.jsp`
- JAXR home page:
  `http://java.sun.com/xml/jaxr/index.html`
- Universal Description, Discovery, and Integration (UDDI) project:
  `http://www.uddi.org/`
- ebXML:
  `http://www.ebxml.org/`
- Open Source JAXR Provider for ebXML Registries:
  `https://sourceforge.net/forum/forum.php?forum_id=197238`
- Java Web Services Developer Pack (Java WSDP):
  `http://java.sun.com/webservices/webservicespack.html`
- Java Technology and XML:
  `http://java.sun.com/xml/`
- Java Technology & Web Services:
  `http://java.sun.com/webservices/index.html`

14

Java Servlet Technology

Stephanie Bodoff

AS soon as the Web began to be used for delivering services, service providers recognized the need for dynamic content. Applets, one of the earliest attempts toward this goal, focused on using the client platform to deliver dynamic user experiences. At the same time, developers also investigated using the server platform for this purpose. Initially, Common Gateway Interface (CGI) scripts were the main technology used to generate dynamic content. Though widely used, CGI scripting technology has a number of shortcomings, including platform dependence and lack of scalability. To address these limitations, Java Servlet technology was created as a portable way to provide dynamic, user-oriented content.

What is a Servlet?

A *servlet* is a Java programming language class used to extend the capabilities of servers that host applications accessed via a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by Web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes.

The `javax.servlet` and `javax.servlet.http` packages provide interfaces and classes for writing servlets. All servlets must implement the `Servlet` interface, which defines life-cycle methods.

When implementing a generic service, you can use or extend the `GenericServlet` class provided with the Java Servlet API. The `HttpServlet` class provides methods, such as `doGet` and `doPost`, for handling HTTP-specific services.

This chapter focuses on writing servlets that generate responses to HTTP requests. Some knowledge of the HTTP protocol is assumed; if you are unfamiliar with this protocol, you can get a brief introduction to HTTP in HTTP Overview (page 849).

The Example Servlets

This chapter uses the Duke's Bookstore application to illustrate the tasks involved in programming servlets. Table 14–1 lists the servlets that handle each bookstore function. Each programming task is illustrated by one or more servlets. For example, `BookDetailsServlet` illustrates how to handle HTTP `GET` requests, `BookDetailsServlet` and `CatalogServlet` show how to construct responses, and `CatalogServlet` illustrates how to track session information.

Table 14–1 Duke's Bookstore Example Servlets

Function	**Servlet**
Enter the bookstore	`BookStoreServlet`
Create the bookstore banner	`BannerServlet`
Browse the bookstore catalog	`CatalogServlet`
Put a book in a shopping cart	`CatalogServlet`, `BookDetailsServlet`
Get detailed information on a specific book	`BookDetailsServlet`
Display the shopping cart	`ShowCartServlet`
Remove one or more books from the shopping cart	`ShowCartServlet`
Buy the books in the shopping cart	`CashierServlet`

Table 14–1 Duke's Bookstore Example Servlets (Continued)

Function	**Servlet**
Receive an acknowledgement for the purchase	`ReceiptServlet`

The data for the bookstore application is maintained in a database and accessed through the helper class `database.BookDB`. The `database` package also contains the class `BookDetails`, which represents a book. The shopping cart and shopping cart items are represented by the classes `cart.ShoppingCart` and `cart.ShoppingCartItem`, respectively.

The source code for the bookstore application is located in the `<`*`JWSDP_HOME`*`>/docs/tutorial/examples/web/bookstore1` directory created when you unzip the tutorial bundle (see Running the Examples, page xiii).

To build, install, and run the example:

1. In a terminal window, go to `<`*`JWSDP_HOME`*`>/docs/tutorial/examples/web/bookstore1`.
2. Run `build`. The `build` target will spawn any necessary compilations and copy files to the `<`*`JWSDP_HOME`*`>/docs/tutorial/examples/web/bookstore1/build` directory.
3. Make sure Tomcat is started.
4. Run `ant install`. The `install` target notifies Tomcat that the new context is available.
5. Start the PointBase database server and populate the database if you have not done so already (see Accessing Databases from Web Applications, page 109).
6. To run the application, open the bookstore URL `http://localhost:8080/bookstore1/enter`.

To deploy the application:

1. Run `ant package`. The package task creates a WAR file containing the application classes in `WEB-INF/classes` and the `context.xml` file in `META-INF`.
2. Make sure Tomcat is started.
3. Run `ant deploy`. The `deploy` target copies the WAR to Tomcat and notifies Tomcat that the new context is available.

Troubleshooting

Common Problems and Their Solutions (page 87) lists some reasons why a Web client can fail. In addition, Duke's Bookstore returns the following exceptions:

- `BookNotFoundException`—Returned if a book can't be located in the bookstore database. This will occur if you haven't loaded the bookstore database with data by running `ant create-book-db` or if the database server hasn't been started or it has crashed.
- `BooksNotFoundException`—Returned if the bookstore data can't be retrieved. This will occur if you haven't loaded the bookstore database with data by running `ant create-book-db` or if the database server hasn't been started or it has crashed.
- `UnavailableException`—Returned if a servlet can't retrieve the Web context attribute representing the bookstore. This will occur if you haven't copied the PointBase client library `<`*`PB_HOME`*`>/lib/pbclient45.jar` to `<`*`JWSDP_HOME`*`>/common/lib`, if the PointBase server hasn't been started, or if you have not defined a data source in Tomcat that references the PointBase database (see Defining a Data Source in Tomcat, page 112).

Because we have specified an error page, you will see the message `The application is unavailable. Please try later`. If you don't specify an error page, the Web container generates a default page containing the message `A Servlet Exception Has Occurred` and a stack trace that can help diagnose the cause of the exception. If you use the `errorpage.html`, you will have to look in the Web container's log to determine the cause of the exception. Web log files reside in the directory `<`*`JWSDP_HOME`*`>/logs` and are named `jwsdp_log.<`*`date`*`>.txt`.

Servlet Life Cycle

The life cycle of a servlet is controlled by the container in which the servlet has been deployed. When a request is mapped to a servlet, the container performs the following steps.

1. If an instance of the servlet does not exist, the Web container
   a. Loads the servlet class.
   b. Creates an instance of the servlet class.
   c. Initializes the servlet instance by calling the `init` method. Initialization is covered in Initializing a Servlet (page 581).

2. Invokes the `service` method, passing a request and response object. Service methods are discussed in Writing Service Methods (page 582).

If the container needs to remove the servlet, it finalizes the servlet by calling the servlet's `destroy` method. Finalization is discussed in Finalizing a Servlet (page 602).

Handling Servlet Life Cycle Events

You can monitor and react to events in a servlet's life cycle by defining listener objects whose methods get invoked when life cycle events occur. To use these listener objects you must define the listener class and specify the listener class.

Defining The Listener Class

You define a listener class as an implementation of a listener interface. Servlet Life Cycle Events (page 575) lists the events that can be monitored and the corresponding interface that must be implemented. When a listener method is invoked, it is passed an event that contains information appropriate to the event. For example, the methods in the `HttpSessionListener` interface are passed an `HttpSessionEvent`, which contains an `HttpSession`.

Table 14–2 Servlet Life Cycle Events

Object	**Event**	**Listener Interface and Event Class**
Web context (See Accessing the Web Context, page 598)	Initialization and destruction	`javax.servlet.ServletContextListener` and `ServletContextEvent`
	Attribute added, removed, or replaced	`javax.servlet.ServletContextAttributeListener` and `ServletContextAttributeEvent`
Session (See Maintaining Client State, page 599)	Creation, invalidation, and timeout	`javax.servlet.http.HttpSessionListener` and `HttpSessionEvent`
	Attribute added, removed, or replaced	`javax.servlet.http.HttpSessionAttributeListener` and `HttpSessionBindingEvent`

The `listeners.ContextListener` class creates and removes the database helper and counter objects used in the Duke's Bookstore application. The methods retrieve the Web context object from `ServletContextEvent` and then store (and remove) the objects as servlet context attributes.

```
import database.BookDB;
import javax.servlet.*;
import util.Counter;

public final class ContextListener
  implements ServletContextListener {
  private ServletContext context = null;
  public void contextInitialized(ServletContextEvent event) {
    context = event.getServletContext();
    try {
      BookDB bookDB = new BookDB();
      context.setAttribute("bookDB", bookDB);
    } catch (Exception ex) {
      System.out.println(
        "Couldn't create database: "
        + ex.getMessage());
    }
    Counter counter = new Counter();
    context.setAttribute("hitCounter", counter);
    context.log("Created hitCounter"
      + counter.getCounter());
    counter = new Counter();
    context.setAttribute("orderCounter", counter);
    context.log("Created orderCounter"
      + counter.getCounter());
  }

  public void contextDestroyed(ServletContextEvent event) {
    context = event.getServletContext();
    BookDB bookDB = context.getAttribute(
      "bookDB");
    bookDB.remove();
    context.removeAttribute("bookDB");
    context.removeAttribute("hitCounter");
    context.removeAttribute("orderCounter");
  }
}
```

Specifying Event Listener Classes

To specify an event listener class, you add a `listener` element to the Web application deployment descriptor. Here is the `listener` element for the Duke's Bookstore application:

```
<listener>
  <listener-class>listeners.ContextListener</listener-class>
</listener>
```

Handling Errors

Any number of exceptions can occur when a servlet is executed. The Web container will generate a default page containing the message `A Servlet Exception Has Occurred` when an exception occurs, but you can also specify that the container should return a specific error page for a given exception. To specify such a page, you add an `error-page` element to the Web application deployment descriptor. These elements map the exceptions returned by the Duke's Bookstore application to `errorpage.html`:

```
<error-page>
  <exception-type>
    exception.BookNotFoundException
  </exception-type>
  <location>/errorpage.html</location>
</error-page>
<error-page>
  <exception-type>
    exception.BooksNotFoundException
  </exception-type>
  <location>/errorpage.html</location>
</error-page>
<error-page>
  <exception-type>exception.OrderException</exception-type>
  <location>/errorpage.html</location>
</error-page>
```

Sharing Information

Web components, like most objects, usually work with other objects to accomplish their tasks. There are several ways they can do this. They can use private helper objects (for example, JavaBeans components), they can share objects that are attributes of a public scope, they can use a database, and they can invoke other Web resources. The Java Servlet technology mechanisms that allow a Web component to invoke other Web resources are described in Invoking Other Web Resources (page 594).

Using Scope Objects

Collaborating Web components share information via objects maintained as attributes of four scope objects. These attributes are accessed with the `[get|set]Attribute` methods of the class representing the scope. Table 14–3 lists the scope objects.

Table 14–3 Scope Objects

Scope Object	Class	Accessible From
Web context	`javax.servlet.ServletContext`	Web components within a Web context. See Accessing the Web Context (page 598).
session	`javax.servlet.http.HttpSession`	Web components handling a request that belongs to the session. See Maintaining Client State (page 599).
request	subtype of `javax.servlet.ServletRequest`	Web components handling the request.
page	`javax.servlet.jsp.PageContext`	The JSP page that creates the object. See Implicit Objects (page 616).

Figure 14–1 shows the scoped attributes maintained by the Duke's Bookstore application.

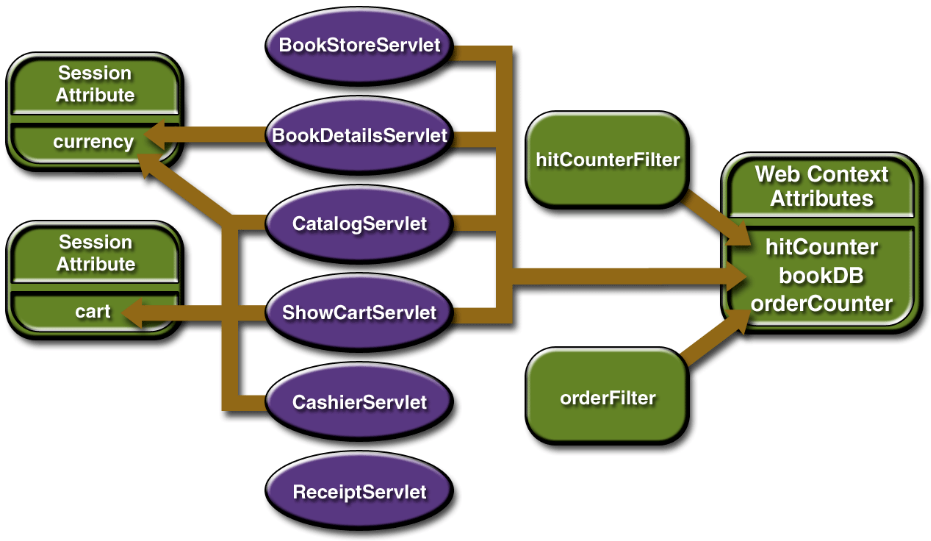


Figure 14–1 Duke's Bookstore Scoped Attributes

Controlling Concurrent Access to Shared Resources

In a multithreaded server, it is possible for shared resources to be accessed concurrently. Besides scope object attributes, shared resources include in-memory data such as instance or class variables, and external objects such as files, database connections, and network connections. Concurrent access can arise in several situations:

- Multiple Web components accessing objects stored in the Web context
- Multiple Web components accessing objects stored in a session
- Multiple threads within a Web component accessing instance variables. A Web container will typically create a thread to handle each request. If you want to ensure that a servlet instance handles only one request at a time, a servlet can implement the `SingleThreadModel` interface. If a servlet implements this interface, you are guaranteed that no two threads will execute concurrently in the servlet's service method. A Web container can

implement this guarantee by synchronizing access to a single instance of the servlet, or by maintaining a pool of Web component instances and dispatching each new request to a free instance. This interface does not prevent synchronization problems that result from Web components accessing shared resources such as static class variables or external objects.

When resources can be accessed concurrently, they can be used in an inconsistent fashion. To prevent this, you must control the access using the synchronization techniques described in the Threads lesson in *The Java Tutorial*.

In the previous section we showed five scoped attributes shared by more than one servlet: `bookDB`, `cart`, `currency`, `hitCounter`, and `orderCounter`. The `bookDB` attribute is discussed in the next section. The cart, currency, and counters can be set and read by multiple multithreaded servlets. To prevent these objects from being used inconsistently, access is controlled by synchronized methods. For example, here is the `util.Counter` class:

```
public class Counter {
  private int counter;
  public Counter() {
    counter = 0;
  }
  public synchronized int getCounter() {
    return counter;
  }
  public synchronized int setCounter(int c) {
    counter = c;
    return counter;
  }
  public synchronized int incCounter() {
    return(++counter);
  }
}
```

Accessing Databases

Data that is shared between Web components and is persistent between invocations of a Web application is usually maintained by a database. Web components use the JDBC 2.0 API to access relational databases. The data for the bookstore application is maintained in a database and accessed through the helper class `database.BookDB`. For example, `ReceiptServlet` invokes the `BookDB.buyBooks` method to update the book inventory when a user makes a purchase. The buyBooks method invokes buyBook for each book contained in the shopping cart. To ensure the order is processed in its entirety, the calls to buyBook are wrapped in a single JDBC transaction. The use of the shared database connection is synchronized via the [get|release]Connection methods.

```
public void buyBooks(ShoppingCart cart) throws OrderException {
  Collection items = cart.getItems();
  Iterator i = items.iterator();
  try {
     getConnection();
     con.setAutoCommit(false);
     while (i.hasNext()) {
        ShoppingCartItem sci = (ShoppingCartItem)i.next();
        BookDetails bd = (BookDetails)sci.getItem();
        String id = bd.getBookId();
        int quantity = sci.getQuantity();
        buyBook(id, quantity);
     }
     con.commit();
     con.setAutoCommit(true);
     releaseConnection();
  } catch (Exception ex) {
     try {
     con.rollback();
     releaseConnection();
     throw new OrderException("Transaction failed: " +
        ex.getMessage());
     } catch (SQLException sqx) {
        releaseConnection();
        throw new OrderException("Rollback failed: " +
           sqx.getMessage());
     }
  }
}
```

Initializing a Servlet

After the Web container loads and instantiates the servlet class and before it delivers requests from clients, the Web container initializes the servlet. You can customize this process to allow the servlet to read persistent configuration data, initialize resources, and perform any other one-time activities by overriding the init method of the Servlet interface. A servlet that cannot complete its initialization process should throw UnavailableException.

All the servlets that access the bookstore database (BookStoreServlet, CatalogServlet, BookDetailsServlet, and ShowCartServlet) initialize a variable in their init method that points to the database helper object created by the Web context listener:

```
public class CatalogServlet extends HttpServlet {
  private BookDB bookDB;
  public void init() throws ServletException {
     bookDB = (BookDB)getServletContext().
        getAttribute("bookDB");
     if (bookDB == null) throw new
        UnavailableException("Couldn't get database.");
  }
}
```

Writing Service Methods

The service provided by a servlet is implemented in the service method of a GenericServlet, the do*Method* methods (where *Method* can take the value Get, Delete, Options, Post, Put, Trace) of an HttpServlet, or any other protocol-specific methods defined by a class that implements the Servlet interface. In the rest of this chapter, the term *service method* will be used for any method in a servlet class that provides a service to a client.

The general pattern for a service method is to extract information from the request, access external resources, and then populate the response based on that information.

For HTTP servlets, the correct procedure for populating the response is to first fill in the response headers, then retrieve an output stream from the response, and finally write any body content to the output stream. Response headers must always be set before a PrintWriter or ServletOutputStream is retrieved because the HTTP protocol expects to receive all headers before body content. The next two sections describe how to get information from requests and generate responses.

Getting Information from Requests

A request contains data passed between a client and the servlet. All requests implement the `ServletRequest` interface. This interface defines methods for accessing the following information:

- Parameters, which are typically used to convey information between clients and servlets
- Object-valued attributes, which are typically used to pass information between the servlet container and a servlet or between collaborating servlets
- Information about the protocol used to communicate the request and the client and server involved in the request
- Information relevant to localization

For example, in `CatalogServlet` the identifier of the book that a customer wishes to purchase is included as a parameter to the request. The following code fragment illustrates how to use the `getParameter` method to extract the identifier:

```
String bookId = request.getParameter("Add");
if (bookId != null) {
  BookDetails book = bookDB.getBookDetails(bookId);
```

You can also retrieve an input stream from the request and manually parse the data. To read character data, use the `BufferedReader` object returned by the request's `getReader` method. To read binary data, use the `ServletInputStream` returned by `getInputStream`.

HTTP servlets are passed an HTTP request object, `HttpServletRequest`, which contains the request URL, HTTP headers, query string, and so on.

An HTTP request URL contains the following parts:

```
http://[host]:[port][request path]?[query string]
```

The request path is further composed of the following elements:

- **Context path:** A concatenation of a forward slash `/` with the context root of the servlet's Web application.
- **Servlet path:** The path section that corresponds to the component alias that activated this request. This path starts with a forward slash `/`.

- **Path info:** The part of the request path that is not part of the context path or the servlet path.

If the context path is `/catalog` and for the aliases listed in Table 14–4, Table 14–5 gives some examples of how the URL will be broken down.

Table 14–4 Aliases

Pattern	Servlet
`/lawn/*`	`LawnServlet`
`/*.jsp`	`JSPServlet`

Table 14–5 Request Path Elements

Request Path	Servlet Path	Path Info
`/catalog/lawn/index.html`	`/lawn`	`/index.html`
`/catalog/help/feedback.jsp`	`/help/feedback.jsp`	`null`

Query strings are composed of a set of parameters and values. Individual parameters are retrieved from a request with the `getParameter` method. There are two ways to generate query strings:

- A query string can explicitly appear in a Web page. For example, an HTML page generated by the `CatalogServlet` could contain the link `<a href="/bookstore1/catalog?Add=101">Add To Cart</a>`. `CatalogServlet` extracts the parameter named `Add` as follows:

  ```
  String bookId = request.getParameter("Add");
  ```

- A query string is appended to a URL when a form with a `GET` HTTP method is submitted. In the Duke's Bookstore application, `CashierServlet` generates a form, then a user name input to the form is appended to the URL that maps to `ReceiptServlet`, and finally `ReceiptServlet` extracts the user name using the `getParameter` method.

Constructing Responses

A response contains data passed between a server and the client. All responses implement the `ServletResponse` interface. This interface defines methods that allow you to do the following:

- Retrieve an output stream to use to send data to the client. To send character data, use the `PrintWriter` returned by the response's `getWriter` method. To send binary data in a MIME body response, use the `ServletOutputStream` returned by `getOutputStream`. To mix binary and text data, for example, to create a multipart response, use a `ServletOutputStream` and manage the character sections manually.
- Indicate the content type (for example, `text/html`), being returned by the response. A registry of content type names is kept by the Internet Assigned Numbers Authority (IANA) at:

  `ftp://ftp.isi.edu/in-notes/iana/assignments/media-types`
- Indicate whether to buffer output. By default, any content written to the output stream is immediately sent to the client. Buffering allows content to be written before anything is actually sent back to the client, thus providing the servlet with more time to set appropriate status codes and headers or forward to another Web resource.
- Set localization information.

HTTP response objects, `HttpServletResponse`, have fields representing HTTP headers such as

- Status codes, which are used to indicate the reason a request is not satisfied.
- Cookies, which are used to store application-specific information at the client. Sometimes cookies are used to maintain an identifier for tracking a user's session (see Session Tracking (page 601)).

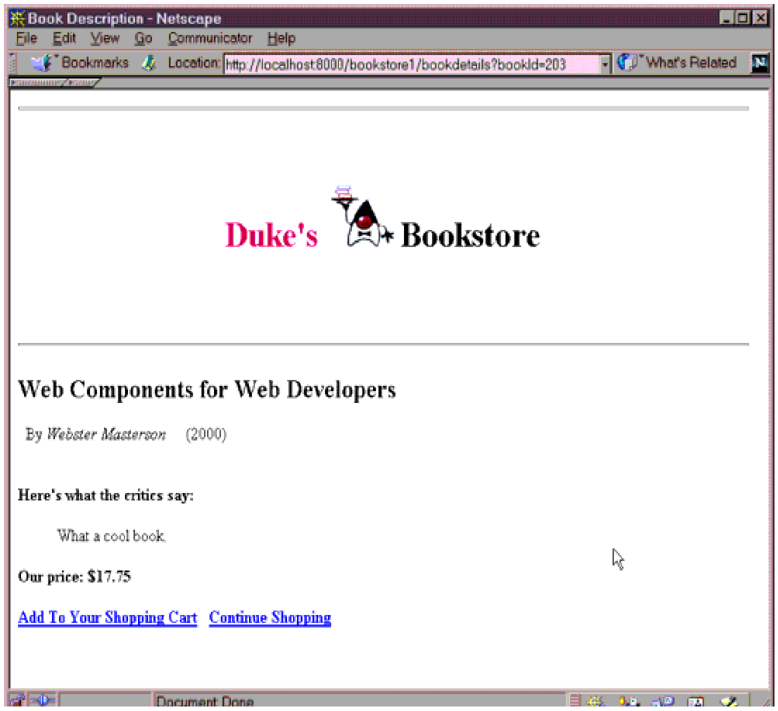
In Duke's Bookstore, `BookDetailsServlet` generates an HTML page that displays information about a book that the servlet retrieves from a database. The servlet first sets response headers: the content type of the response and the buffer size. The servlet buffers the page content because the database access can generate an exception that would cause forwarding to an error page. By buffering the response, the client will not see a concatenation of part of a Duke's Bookstore page with the error page should an error occur. The `doGet` method then retrieves a `PrintWriter` from the response.

For filling in the response, the servlet first dispatches the request to `BannerServlet`, which generates a common banner for all the servlets in the application. This process is discussed in Including Other Resources in the Response (page 595). Then the servlet retrieves the book identifier from a request parameter and uses the identifier to retrieve information about the book from the bookstore database. Finally, the servlet generates HTML markup that describes the book information and commits the response to the client by calling the `close` method on the `PrintWriter`.

```
public class BookDetailsServlet extends HttpServlet {
   public void doGet (HttpServletRequest request,
       HttpServletResponse response)
       throws ServletException, IOException {
     // set headers before accessing the Writer
     response.setContentType("text/html");
     response.setBufferSize(8192);
     PrintWriter out = response.getWriter();

     // then write the response
     out.println("<html>" +
       "<head><title>+
       messages.getString("TitleBookDescription")
       +</title></head>");

     // Get the dispatcher; it gets the banner to the user
     RequestDispatcher dispatcher =
       getServletContext().
       getRequestDispatcher("/banner");
     if (dispatcher != null)
       dispatcher.include(request, response);

     //Get the identifier of the book to display
     String bookId = request.getParameter("bookId");
     if (bookId != null) {
       // and the information about the book
       try {
          BookDetails bd =
            bookDB.getBookDetails(bookId);
          ...
          //Print out the information obtained
          out.println("<h2>" + bd.getTitle() + "</h2>" +
          ...
       } catch (BookNotFoundException ex) {
          response.resetBuffer();
          throw new ServletException(ex);
       }
     }
```

```
         out.println("</body></html>");
         out.close();
      }
   }
```

`BookDetailsServlet` generates a page that looks like:


Book Description - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://localhost:8000/bookstore1/bookdetails?bookId=203 What's Related
Duke's Bookstore
Document Done

Web Components for Web Developers

By *Webster Masterson* (2000)

Here's what the critics say:

What a cool book.

Our price: $17.75

Add To Your Shopping Cart Continue Shopping

Figure 14–2 Book Details

Filtering Requests and Responses

A *filter* is an object that can transform the header and content (or both) of a request or response. Filters differ from Web components in that they usually do not themselves create a response. Instead, a filter provides functionality that can be “attached” to any kind of Web resource. As a consequence, a filter should not have any dependencies on a Web resource for which it is acting as a filter, so that it can be composable with more than one type of Web resource. The main tasks that a filter can perform are as follows:

- Query the request and act accordingly.
- Block the request and response pair from passing any further.
- Modify the request headers and data. You do this by providing a customized version of the request.
- Modify the response headers and data. You do this by providing a customized version of the response.
- Interact with external resources.

Applications of filters include authentication, logging, image conversion, data compression, encryption, tokenizing streams, and XML transformations, and so on.

You can configure a Web resource to be filtered by a chain of zero, one, or more filters in a specific order. This chain is specified when the Web application containing the component is deployed and is instantiated when a Web container loads the component.

In summary, the tasks involved in using filters include

- Programming the filter
- Programming customized requests and responses
- Specifying the filter chain for each Web resource

Programming Filters

The filtering API is defined by the `Filter`, `FilterChain`, and `FilterConfig` interfaces in the `javax.servlet` package. You define a filter by implementing the `Filter` interface. The most important method in this interface is the `doFilter` method, which is passed request, response, and filter chain objects. This method can perform the following actions:

- Examine the request headers.
- Customize the request object if it wishes to modify request headers or data.
- Customize the response object if it wishes to modify response headers or data.
- Invoke the next entity in the filter chain. If the current filter is the last filter in the chain that ends with the target Web component or static resource, the next entity is the resource at the end of the chain; otherwise, it is the next

filter that was configured in the WAR. It invokes the next entity by calling the doFilter method on the chain object (passing in the request and response it was called with, or the wrapped versions it may have created). Alternatively, it can choose to block the request by not making the call to invoke the next entity. In the latter case, the filter is responsible for filling out the response.

- Examine response headers after it has invoked the next filter in the chain
- Throw an exception to indicate an error in processing

In addition to doFilter, you must implement the init and destroy methods. The init method is called by the container when the filter is instantiated. If you wish to pass initialization parameters to the filter, you retrieve them from the FilterConfig object passed to init.

The Duke's Bookstore application uses the filters HitCounterFilter and OrderFilter to increment and log the value of a counter when the entry and receipt servlets are accessed.

In the doFilter method, both filters retrieve the servlet context from the filter configuration object so that they can access the counters stored as context attributes. After the filters have completed application-specific processing, they invoke doFilter on the filter chain object passed into the original doFilter method. The elided code is discussed in the next section.

```
public final class HitCounterFilter implements Filter {
  private FilterConfig filterConfig = null;

  public void init(FilterConfig filterConfig)
    throws ServletException {
    this.filterConfig = filterConfig;
  }
  public void destroy() {
    this.filterConfig = null;
  }
  public void doFilter(ServletRequest request,
    ServletResponse response, FilterChain chain)
    throws IOException, ServletException {
    if (filterConfig == null)
      return;
    StringWriter sw = new StringWriter();
    PrintWriter writer = new PrintWriter(sw);
    Counter counter = (Counter)filterConfig.
      getServletContext().
      getAttribute("hitCounter");
    writer.println();
```

```
            writer.println("===============");
            writer.println("The number of hits is: " +
               counter.incCounter());
            writer.println("===============");
            // Log the resulting string
            writer.flush();
            filterConfig.getServletContext().
               log(sw.getBuffer().toString());
            ...
            chain.doFilter(request, wrapper);
            ...
         }
      }
```

Programming Customized Requests and Responses

There are many ways for a filter to modify a request or response. For example, a filter could add an attribute to the request or insert data in the response. In the Duke's Bookstore example, `HitCounterFilter` inserts the value of the counter into the response.

A filter that modifies a response must usually capture the response before it is returned to the client. The way to do this is to pass a stand-in stream to the servlet that generates the response. The stand-in stream prevents the servlet from closing the original response stream when it completes and allows the filter to modify the servlet's response.

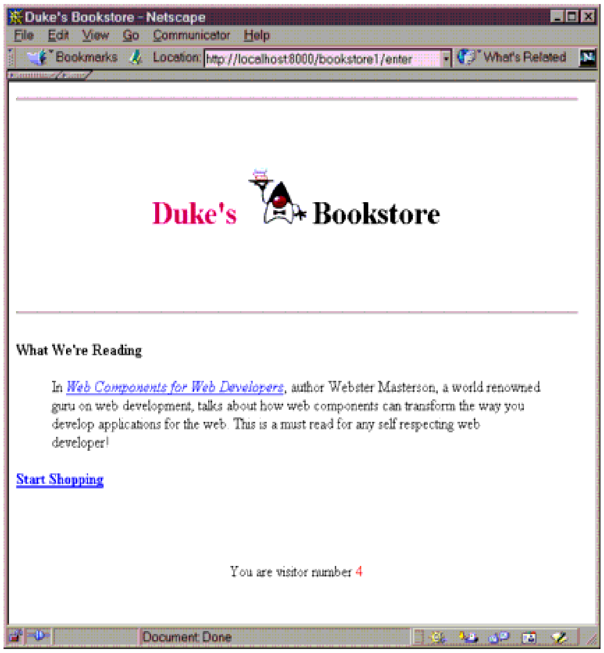
To pass this stand-in stream to the servlet, the filter creates a response wrapper that overrides the `getWriter` or `getOutputStream` method to return this stand-in stream. The wrapper is passed to the `doFilter` method of the filter chain. Wrapper methods default to calling through to the wrapped request or response object. This approach follows the well-known Wrapper or Decorator pattern described in *Design Patterns, Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995). The following sections describe how the hit counter filter described earlier and other types of filters use wrappers.

To override request methods, you wrap the request in an object that extends `ServletRequestWrapper` or `HttpServletRequestWrapper`. To override response methods, you wrap the response in an object that extends `ServletResponseWrapper` or `HttpServletResponseWrapper`.

HitCounterFilter wraps the response in a CharResponseWrapper. The wrapped response is passed to the next object in the filter chain, which is BookStoreServlet. BookStoreServlet writes its response into the stream created by CharResponseWrapper. When chain.doFilter returns, HitCounterFilter retrieves the servlet's response from PrintWriter and writes it to a buffer. The filter inserts the value of the counter into the buffer, resets the content length header of the response, and finally writes the contents of the buffer to the response stream.

```
PrintWriter out = response.getWriter();
CharResponseWrapper wrapper = new CharResponseWrapper(
  (HttpServletResponse)response);
chain.doFilter(request, wrapper);
CharArrayWriter caw = new CharArrayWriter();
caw.write(wrapper.toString().substring(0,
  wrapper.toString().indexOf("</body>")-1));
caw.write("<p>\n<center>" +
  messages.getString("Visitor") + "<font color='red'>" +
  counter.getCounter() + "</font></center>");
caw.write("\n</body></html>");
response.setContentLength(caw.toString().length());
out.write(caw.toString());
out.close();

public class CharResponseWrapper extends
  HttpServletResponseWrapper {
  private CharArrayWriter output;
  public String toString() {
    return output.toString();
  }
  public CharResponseWrapper(HttpServletResponse response){
    super(response);
    output = new CharArrayWriter();
  }
  public PrintWriter getWriter(){
    return new PrintWriter(output);
  }
}
```

Figure 14–3 shows the entry page for Duke's Bookstore with the hit counter.


Duke's Bookstore - Netscape
File Edit View Go Communicator Help
Bookmarks
Location: http://localhost:8000/bookstore1/enter
What's Related
Document: Done

Duke's Bookstore

What We're Reading

In *Web Components for Web Developers*, author Webster Masterson, a world renowned guru on web development, talks about how web components can transform the way you develop applications for the web. This is a must read for any self respecting web developer!

Start Shopping

You are visitor number 4

Figure 14–3 Duke's Bookstore

Specifying Filter Mappings

A Web container uses filter mappings to decide how to apply filters to Web resources. A filter mapping matches a filter to a Web component by name or to Web resources by URL pattern. The filters are invoked in the order in which filter mappings appear in the filter mapping list of a WAR. You specify a filter mapping list for a WAR by coding them directly in the Web application deployment descriptor:

- Declare the filter using the `<filter>` element. This element creates a name for the filter and declares the filter's implementation class and initialization parameters.
- Map the filter to a Web resource by defining a `<filter-mapping>` element. This element maps a filter name to a Web resource by name or by URL pattern.

The following elements show how to specify the hit counter and order filters. To define a filter you provide a name for the filter, the class that implements the filter, and optionally some initialization parameters.

```
<filter>
  <filter-name>OrderFilter</filter-name>
  <filter-class>filters.OrderFilter</filter-class>
</filter>
<filter>
  <filter-name>HitCounterFilter</filter-name>
  <filter-class>filters.HitCounterFilter</filter-class>
</filter>
```

The `filter-mapping` element maps the order filter to the `/receipt` URL. The mapping could also have specified the servlet `ReceiptServlet`. Note that the `filter`, `filter-mapping`, `servlet`, and `servlet-mapping` elements must appear in the Web application deployment descriptor in that order.

```
<filter-mapping>
    <filter-name>OrderFilter</filter-name>
    <url-pattern>/receipt</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>HitCounterFilter</filter-name>
    <url-pattern>/enter</url-pattern>
</filter-mapping>
```

If you want to log every request to a Web application, you would map the hit counter filter to the URL pattern `/*`. Table 14–6 summarizes the filter mapping list for the Duke's Bookstore application. The filters are matched by URL pattern and each filter chain contains only one filter.

Table 14–6 Duke's Bookstore Filter Mapping List

URL	**Filter**
`/enter`	`HitCounterFilter`
`/receipt`	`OrderFilter`

You can map a filter to one or more Web resources and you can map more than one filter to a Web resource. This is illustrated in Figure 14–4, where filter F1 is mapped to servlets S1, S2, and S3, filter F2 is mapped to servlet S2, and filter F3 is mapped to servlets S1 and S2.

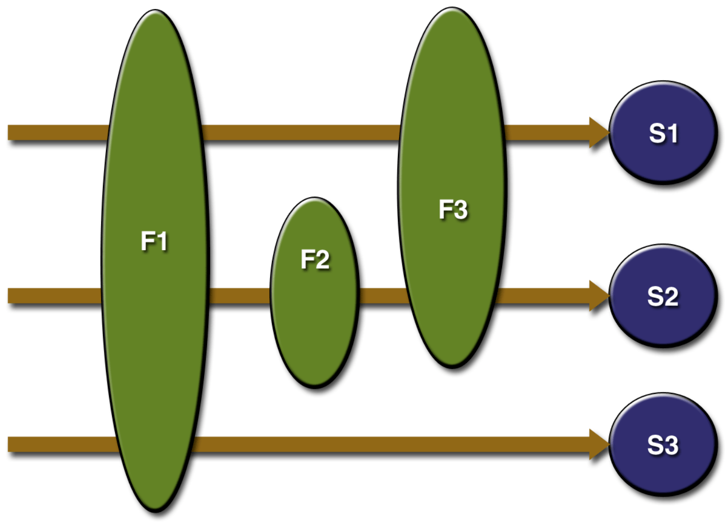


Figure 14–4 Filter to Servlet Mapping

Recall that a filter chain is one of the objects passed to the `doFilter` method of a filter. This chain is formed indirectly via filter mappings. The order of the filters in the chain is the same as the order in which filter mappings appear in the Web application deployment descriptor.

When a filter is mapped to servlet S1, the Web container invokes the `doFilter` method of F1. The `doFilter` method of each filter in S1's filter chain is invoked by the preceding filter in the chain via the `chain.doFilter` method. Since S1's filter chain contains filters F1 and F3, F1's call to `chain.doFilter` invokes the `doFilter` method of filter F3. When F3's `doFilter` method completes, control returns to F1's `doFilter` method.

Invoking Other Web Resources

Web components can invoke other Web resources in two ways: indirect and direct. A Web component indirectly invokes another Web resource when it embeds in content returned to a client a URL that points to another Web component. In the Duke's Bookstore application, most Web components contain embedded URLs that point to other Web components. For example, `ShowCart-`

`Servlet` indirectly invokes the `CatalogServlet` through the embedded URL `/bookstore1/catalog`.

A Web component can also directly invoke another resource while it is executing. There are two possibilities: it can include the content of another resource, or it can forward a request to another resource.

To invoke a resource available on the server that is running a Web component, you must first obtain a `RequestDispatcher` object using the `getRequestDispatcher("URL")` method.

You can get a `RequestDispatcher` object from either a request or the Web context, however, the two methods have slightly different behavior. The method takes the path to the requested resource as an argument. A request can take a relative path (that is, one that does not begin with a `/`), but the Web context requires an absolute path. If the resource is not available, or if the server has not implemented a `RequestDispatcher` object for that type of resource, `getRequestDispatcher` will return null. Your servlet should be prepared to deal with this condition.

Including Other Resources in the Response

It is often useful to include another Web resource, for example, banner content or copyright information, in the response returned from a Web component. To include another resource, invoke the `include` method of a `RequestDispatcher object`:

```
include(request, response);
```

If the resource is static, the `include` method enables programmatic server-side includes. If the resource is a Web component, the effect of the method is to send the request to the included Web component, execute the Web component, and then include the result of the execution in the response from the containing servlet. An included Web component has access to the request object, but it is limited in what it can do with the response object:

- It can write to the body of the response and commit a response.
- It cannot set headers or call any method (for example, `setCookie`) that affects the headers of the response.

The banner for the Duke's Bookstore application is generated by BannerServlet. Note that both the doGet and doPost methods are implemented because BannerServlet can be dispatched from either method in a calling servlet.

```
public class BannerServlet extends HttpServlet {
  public void doGet (HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {

    PrintWriter out = response.getWriter();
    out.println("<body bgcolor=\"#ffffff\">" +
    "<center>" + "<hr> <br> &nbsp;" + "<h1>" +
    "<font size=\"+3\" color=\"#CC0066\">Duke's </font>" +
    <img src=\"" + request.getContextPath() +
    "/duke.books.gif\">" +
    "<font size=\"+3\" color=\"black\">Bookstore</font>" +
    "</h1>" + "</center>" + "<br> &nbsp; <hr> <br> ");
  }
  public void doPost (HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {

    PrintWriter out = response.getWriter();
    out.println("<body bgcolor=\"#ffffff\">" +
    "<center>" + "<hr> <br> &nbsp;" + "<h1>" +
    "<font size=\"+3\" color=\"#CC0066\">Duke's </font>" +
    <img src=\"" + request.getContextPath() +
    "/duke.books.gif\">" +
    "<font size=\"+3\" color=\"black\">Bookstore</font>" +
    "</h1>" + "</center>" + "<br> &nbsp; <hr> <br> ");
  }
}
```

Each servlet in the Duke's Bookstore application includes the result from BannerServlet with the following code:

```
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher("/banner");
if (dispatcher != null)
  dispatcher.include(request, response);
}
```

Transferring Control to Another Web Component

In some applications, you might want to have one Web component do preliminary processing of a request and have another component generate the response. For example, you might want to partially process a request and then transfer to another component depending on the nature of the request.

To transfer control to another Web component, you invoke the `forward` method of a `RequestDispatcher`. When a request is forwarded, the request URL is set to the path of the forwarded page. If the original URL is required for any processing, you can save it as a request attribute. The `Dispatcher` servlet, used by a version of the Duke's Bookstore application described in The Example JSP Pages (page 638), saves the path information from the original URL, retrieves a `RequestDispatcher` from the request, and then forwards to the JSP page `template.jsp`.

```
public class Dispatcher extends HttpServlet {
  public void doGet(HttpServletRequest request,
    HttpServletResponse response) {
    request.setAttribute("selectedScreen",
      request.getServletPath());
    RequestDispatcher dispatcher = request.
      getRequestDispatcher("/template.jsp");
    if (dispatcher != null)
      dispatcher.forward(request, response);
  }
  public void doPost(HttpServletRequest request,
  ...
}
```

The `forward` method should be used to give another resource responsibility for replying to the user. If you have already accessed a `ServletOutputStream` or `PrintWriter` object within the servlet, you cannot use this method; it throws an `IllegalStateException`.

Accessing the Web Context

The context in which Web components execute is an object that implements the `ServletContext` interface. You retrieve the Web context with the `getServletContext` method. The Web context provides methods for accessing:

- Initialization parameters
- Resources associated with the Web context
- Object-valued attributes
- Logging capabilities

The Web context is used by the Duke's Bookstore filters `filters.HitCounterFilter` and `OrderFilter`, which were discussed in Filtering Requests and Responses (page 587). The filters store a counter as a context attribute. Recall from Controlling Concurrent Access to Shared Resources (page 579) that the counter's access methods are synchronized to prevent incompatible operations by servlets that are running concurrently. A filter retrieves the counter object with the context's `getAttribute` method. The incremented value of the counter is recorded with the context's `log` method.

```
public final class HitCounterFilter implements Filter {
  private FilterConfig filterConfig = null;
  public void doFilter(ServletRequest request,
    ServletResponse response, FilterChain chain)
    throws IOException, ServletException {
    ...
    StringWriter sw = new StringWriter();
    PrintWriter writer = new PrintWriter(sw);
    ServletContext context = filterConfig.
      getServletContext();
    Counter counter = (Counter)context.
      getAttribute("hitCounter");
    ...
    writer.println("The number of hits is: " +
      counter.incCounter());
    ...
    context.log(sw.getBuffer().toString());
    ...
  }
}
```

Maintaining Client State

Many applications require a series of requests from a client to be associated with one another. For example, the Duke's Bookstore application saves the state of a user's shopping cart across requests. Web-based applications are responsible for maintaining such state, called a *session*, because the HTTP protocol is stateless. To support applications that need to maintain state, Java Servlet technology provides an API for managing sessions and allows several mechanisms for implementing sessions.

Accessing a Session

Sessions are represented by an `HttpSession` object. You access a session by calling the `getSession` method of a request object. This method returns the current session associated with this request, or, if the request does not have a session, it creates one. Since `getSession` may modify the response header (if cookies are the session tracking mechanism), it needs to be called before you retrieve a `PrintWriter` or `ServletOutputStream`.

Associating Attributes with a Session

You can associate object-valued attributes with a session by name. Such attributes are accessible by any Web component that belongs to the same Web context *and* is handling a request that is part of the same session.

The Duke's Bookstore application stores a customer's shopping cart as a session attribute. This allows the shopping cart to be saved between requests and also allows cooperating servlets to access the cart. `CatalogServlet` adds items to the cart; `ShowCartServlet` displays, deletes items from, and clears the cart; and `CashierServlet` retrieves the total cost of the books in the cart.

```
public class CashierServlet extends HttpServlet {
  public void doGet (HttpServletRequest request,
     HttpServletResponse response)
     throws ServletException, IOException {

     // Get the user's session and shopping cart
     HttpSession session = request.getSession();
     ShoppingCart cart =
        (ShoppingCart)session.
```

```
          getAttribute("cart");
    ...
    // Determine the total price of the user's books
    double total = cart.getTotal();
```

Notifying Objects That Are Associated with a Session

Recall that your application can notify Web context and session listener objects of servlet life cycle events (Handling Servlet Life Cycle Events (page 575)). You can also notify objects of certain events related to their association with a session such as the following:

- When the object is added to or removed from a session. To receive this notification, your object must implement the `javax.http.HttpSessionBindingListener` interface.
- When the session to which the object is attached will be passivated or activated. A session will be passivated or activated when it is moved between virtual machines or saved to and restored from persistent storage. To receive this notification, your object must implement the `javax.http.HttpSessionActivationListener` interface.

Session Management

Since there is no way for an HTTP client to signal that it no longer needs a session, each session has an associated timeout so that its resources can be reclaimed. The timeout period can be accessed with a session's `[get|set]MaxInactiveInterval` methods. You can also set the time-out period in the deployment descriptor:

```
<web-app>
   <display-name>Hello World Application</display-name>
   <description>A web application</description>
   <session-config>
      <session-timeout>60</session-timeout>
   </session-config>
</web-app>
```

To ensure that an active session is not timed out, you should periodically access the session via service methods because this resets the session's time-to-live counter.

When a particular client interaction is finished, you use the session's invalidate method to invalidate a session on the server side and remove any session data.

The bookstore application's ReceiptServlet is the last servlet to access a client's session, so it has responsibility for invalidating the session:

```
public class ReceiptServlet extends HttpServlet {
  public void doPost(HttpServletRequest request,
           HttpServletResponse response)
           throws ServletException, IOException {
     // Get the user's session and shopping cart
     HttpSession session = request.getSession();
     // Payment received -- invalidate the session
     session.invalidate();
     ...
```

Session Tracking

A Web container can use several methods to associate a session with a user, all of which involve passing an identifier between the client and server. The identifier can be maintained on the client as a cookie or the Web component can include the identifier in every URL that is returned to the client.

If your application makes use of session objects, you must ensure that session tracking is enabled by having the application rewrite URLs whenever the client turns off cookies. You do this by calling the response's encodeURL(URL) method on all URLs returned by a servlet. This method includes the session ID in the URL only if cookies are disabled; otherwise, it returns the URL unchanged.

The doGet method of ShowCartServlet encodes the three URLs at the bottom of the shopping cart display page as follows:

```
out.println("<p> &nbsp; <p><strong><a href=\"" +
  response.encodeURL(request.getContextPath() + "/catalog") +
     "\">" + messages.getString("ContinueShopping") +
     "</a> &nbsp; &nbsp; &nbsp;" +
     "<a href=\"" +
  response.encodeURL(request.getContextPath() + "/cashier") +
     "\">" + messages.getString("Checkout") +
     "</a> &nbsp; &nbsp; &nbsp;" +
     "<a href=\"" +
    response.encodeURL(request.getContextPath() +
      "/showcart?Clear=clear") +
      "\">" + messages.getString("ClearCart") +
      "</a></strong>");
```

If cookies are turned off, the session is encoded in the Check Out URL as follows:

```
http://localhost:8080/bookstore1/cashier;
  jsessionid=c0o7fszeb1
```

If cookies are turned on, the URL is simply

```
http://localhost:8080/bookstore1/cashier
```

Finalizing a Servlet

When a servlet container determines that a servlet should be removed from service (for example, when a container wants to reclaim memory resources, or when it is being shut down), it calls the `destroy` method of the `Servlet` interface. In this method, you release any resources the servlet is using and save any persistent state. The following `destroy` method releases the database object created in the `init` method described in Initializing a Servlet (page 581):

```
public void destroy() {
  bookDB = null;
}
```

All of a servlet's service methods should be complete when a servlet is removed. The server tries to ensure this by calling the `destroy` method only after all service requests have returned, or after a server-specific grace period, whichever comes first. If your servlet has operations that take a long time to run (that is, operations that may run longer than the server's grace period), the operations could still be running when `destroy` is called. You must make sure that any threads still handling client requests complete; the remainder of this section describes how to:

- Keep track of how many threads are currently running the `service` method
- Provide a clean shutdown by having the `destroy` method notify long-running threads of the shutdown and wait for them to complete
- Have the long-running methods poll periodically to check for shutdown and, if necessary, stop working, clean up, and return

Tracking Service Requests

To track service requests, include in your servlet class a field that counts the number of service methods that are running. The field should have synchronized access methods to increment, decrement, and return its value.

```
public class ShutdownExample extends HttpServlet {
  private int serviceCounter = 0;
  ...
  //Access methods for serviceCounter
  protected synchronized void enteringServiceMethod() {
    serviceCounter++;
  }
  protected synchronized void leavingServiceMethod() {
    serviceCounter--;
  }
  protected synchronized int numServices() {
    return serviceCounter;
  }
}
```

The `service` method should increment the service counter each time the method is entered and should decrement the counter each time the method returns. This is one of the few times that your `HttpServlet` subclass should override the `service` method. The new method should call `super.service` to preserve all of the original `service` method's functionality:

```
protected void service(HttpServletRequest req,
            HttpServletResponse resp)
            throws ServletException,IOException {
  enteringServiceMethod();
  try {
    super.service(req, resp);
  } finally {
    leavingServiceMethod();
  }
}
```

Notifying Methods to Shut Down

To ensure a clean shutdown, your `destroy` method should not release any shared resources until all of the service requests have completed. One part of doing this is to check the service counter. Another part is to notify the long-running methods that it is time to shut down. For this notification another field is required. The field should have the usual access methods:

```
public class ShutdownExample extends HttpServlet {
  private boolean shuttingDown;
  ...
  //Access methods for shuttingDown
  protected synchronized void setShuttingDown(boolean flag) {
    shuttingDown = flag;
  }
  protected synchronized boolean isShuttingDown() {
    return shuttingDown;
  }
}
```

An example of the destroy method using these fields to provide a clean shutdown follows:

```
public void destroy() {
  /* Check to see whether there are still service methods /*
  /* running, and if there are, tell them to stop. */
  if (numServices() > 0) {
    setShuttingDown(true);
  }

  /* Wait for the service methods to stop. */
  while(numServices() > 0) {
    try {
      Thread.sleep(interval);
    } catch (InterruptedException e) {
    }
  }
}
```

Creating Polite Long-Running Methods

The final step in providing a clean shutdown is to make any long-running methods behave politely. Methods that might run for a long time should check the value of the field that notifies them of shutdowns and should interrupt their work, if necessary.

```
public void doPost(...) {
  ...
  for(i = 0; ((i < lotsOfStuffToDo) &&
    !isShuttingDown()); i++) {
            try {
               partOfLongRunningOperation(i);
            } catch (InterruptedException e) {
               ...
            }
         }
      }
```

Further Information

For further information on Java Servlet technology see:

- Resources listed on the Web site `http://java.sun.com/products/servlet`.
- The Java Servlet 2.3 Specification.

15

JavaServer Pages Technology

Stephanie Bodoff

JAVASERVER Pages (JSP) technology allows you to easily create Web content that has both static and dynamic components. JSP technology projects all the dynamic capabilities of Java Servlet technology but provides a more natural approach to creating static content. The main features of JSP technology are

- A language for developing JSP pages, which are text-based documents that describe how to process a request and construct a response
- Constructs for accessing server-side objects
- Mechanisms for defining extensions to the JSP language

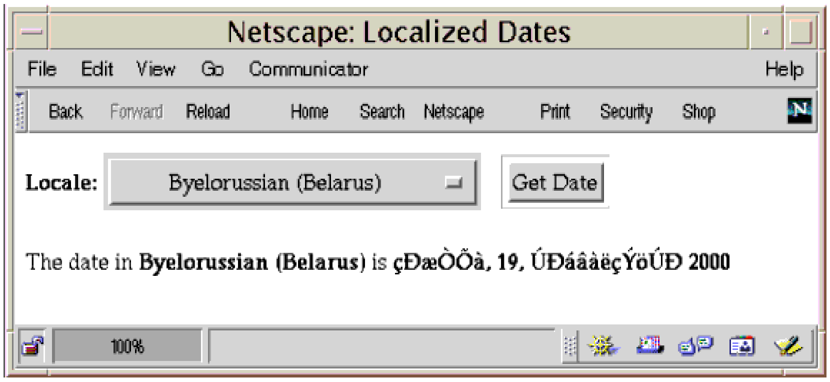
JSP technology also contains an API that is used by developers of Web containers, but this API is not covered in this chapter.

What Is a JSP Page?

A *JSP page* is a text-based document that contains two types of text: static template data, which can be expressed in any text-based format, such as HTML, SVG, WML, and XML; and JSP elements, which construct dynamic content. A syntax card and reference for the JSP elements are available at

```
http://java.sun.com/products/jsp/technical.html#syntax
```

The Web page in Figure 15–1 is a form that allows you to select a locale and displays the date in a manner appropriate to the locale.


Netscape: Localized Dates
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop
Locale:
Byelorussian (Belarus)
Get Date
100%

The date in **Byelorussian (Belarus)** is **çĐæÒÕà, 19, ÚĐáâàëçÝöÚĐ 2000**

Figure 15–1 Localized Date Form

The source code for this example is in the `docs/tutorial/examples/web/date` directory created when you unzip the tutorial bundle. The JSP page `index.jsp` used to create the form appears below; it is a typical mixture of static HTML markup and JSP elements. If you have developed Web pages, you are probably familiar with the HTML document structure statements (`<head>`, `<body>`, and so on) and the HTML statements that create a form `<form>` and a menu `<select>`. The lines in bold in the example code contains the following types of JSP constructs:

- Directives (**`<%@page ... %>`**) import classes in the `java.util` package and the `MyLocales` class, and set the content type returned by the page.
- The **`jsp:useBean`** element creates an object containing a collection of locales and initializes a variable that points to that object.
- Scriptlets (**`<% ... %>`**) retrieve the value of the `locale` request parameter, iterate over a collection of locale names, and conditionally insert HTML text into the output.
- Expressions (**`<%= ... %>`**) insert the value of the locale name into the response.

- The **jsp:include** element sends a request to another page (`date.jsp`) and includes the response in the response from the calling page.

```
<%@ page import="java.util.*,MyLocales" %>
<%@ page contentType="text/html; charset=ISO-8859-5" %>
<html>
<head><title>Localized Dates</title></head>
<body bgcolor="white">
<jsp:useBean id="locales" scope="application"
  class="MyLocales"/>
<form name="localeForm" action="index.jsp" method="post">
<b>Locale:</b>
<select name=locale>
<%
  String selectedLocale = request.getParameter("locale");
  Iterator i = locales.getLocaleNames().iterator();
  while (i.hasNext()) {
     String locale = (String)i.next();
     if (selectedLocale != null &&
        selectedLocale.equals(locale)) {
%>
        <option selected><%=locale%></option>
<%
     } else {
%>
        <option><%=locale%></option>
<%
     }
  }
%>
</select>
<input type="submit" name="Submit" value="Get Date">
</form>
<jsp:include page="date.jsp"/>
</body>
</html>
```

To build, deploy, and execute this JSP page:

1. In a terminal window, go to `docs/tutorial/examples/web/date`.
2. Run `ant build`. The `build` target will spawn any necessary compilations and copy files to the `docs/tutorial/examples/web/date/build` directory.
3. Run `ant install`. The `install` target notifies Tomcat that the new context is available.
4. Open the date URL `http://localhost:8080/date`.

You will see a combo box whose entries are locales. Select a locale and click Get Date. You will see the date expressed in a manner appropriate for that locale.

The Example JSP Pages

To illustrate JSP technology, this chapter rewrites each servlet in the Duke's Bookstore application introduced in Chapter 14 as a JSP page:

Table 15–1 Duke's Bookstore Example JSP Pages

Function	JSP Pages
Enter the bookstore	`bookstore.jsp`
Create the bookstore banner	`banner.jsp`
Browse the books offered for sale	`catalog.jsp`
Put a book in a shopping cart	`catalog.jsp` and `bookdetails.jsp`
Get detailed information on a specific book	`bookdetails.jsp`
Display the shopping cart	`showcart.jsp`
Remove one or more books from the shopping cart	`showcart.jsp`
Buy the books in the shopping cart	`cashier.jsp`
Receive an acknowledgement for the purchase	`receipt.jsp`

The data for the bookstore application is still maintained in a database. However, two changes are made to the database helper object `database.BookDB`:

- The database helper object is rewritten to conform to JavaBeans component design patterns as described in JavaBeans Component Design Conventions (page 627). This change is made so that JSP pages can access the helper object using JSP language elements specific to JavaBeans components.
- Instead of accessing the bookstore database directly, the helper object goes through a data access object `database.BookDAO`.

The implementation of the database helper object follows. The bean has two instance variables: the current book and a reference to the database enterprise bean.

```
public class BookDB {
  private String bookId = "0";
  private BookDBEJB database = null;

  public BookDB () throws Exception {
  }
  public void setBookId(String bookId) {
    this.bookId = bookId;
  }
  public void setDatabase(BookDBEJB database) {
    this.database = database;
  }
  public BookDetails getBookDetails()
    throws Exception {
    try {
      return (BookDetails)database.
          getBookDetails(bookId);
    } catch (BookNotFoundException ex) {
      throw ex;
    }
  }
  ...
}
```

Finally, this version of the example contains an applet to generate a dynamic digital clock in the banner. See Including an Applet (page 624) for a description of the JSP element that generates HTML for downloading the applet.

The source code for the application is located in the `docs/tutorial/examples/web/bookstore2` directory created when you unzip the tutorial bundle (see Running the Examples, page xiii). To build, deploy, and run the example:

1. In a terminal window, go to `docs/tutorial/examples/web/bookstore2`.
2. Run `ant build`. The `build` target will spawn any necessary compilations and copy files to the `docs/tutorial/examples/web/bookstore2/build` directory.
3. Make sure Tomcat is started.
4. Run `ant install`. The `install` target notifies Tomcat that the new context is available.

5. Start the PointBase database server and populate the database if you have not done so already (see Accessing Databases from Web Applications, page 109).
6. Open the bookstore URL `http://localhost:8080/bookstore2/enter`.

See Common Problems and Their Solutions (page 87) and Troubleshooting (page 574) for help with diagnosing common problems.

The Life Cycle of a JSP Page

A JSP page services requests as a servlet. Thus, the life cycle and many of the capabilities of JSP pages (in particular the dynamic aspects) are determined by Java Servlet technology, and much of the discussion in this chapter refers to functions described in Chapter 14.

When a request is mapped to a JSP page, it is handled by a special servlet that first checks whether the JSP page's servlet is older than the JSP page. If it is, it translates the JSP page into a servlet class and compiles the class. During development, one of the advantages of JSP pages over servlets is that the build process is performed automatically.

Translation and Compilation

During the translation phase each type of data in a JSP page is treated differently. Template data is transformed into code that will emit the data into the stream that returns data to the client. JSP elements are treated as follows:

- Directives are used to control how the Web container translates and executes the JSP page.
- Scripting elements are inserted into the JSP page's servlet class. See JSP Scripting Elements (page 619) for details.
- Elements of the form `<jsp:`*`XXX`*` ... />` are converted into method calls to JavaBeans components or invocations of the Java Servlet API.

For a JSP page named *`pageName`*, the source for a JSP page's servlet is kept in the file:

```
<JWSDP_HOME>/work/Standard Engine/
  localhost/context_root/pageName$jsp.java
```

For example, the source for the index page (named `index.jsp`) for the `date` localization example discussed at the beginning of the chapter would be named:

```
<JWSDP_HOME>/work/Standard Engine/
  localhost/date/index$jsp.java
```

Both the translation and compilation phases can yield errors that are only observed when the page is requested for the first time. If an error occurs while the page is being translated (for example, if the translator encounters a malformed JSP element), the server will return a `ParseException`, and the servlet class source file will be empty or incomplete. The last incomplete line will give a pointer to the incorrect JSP element.

If an error occurs while the JSP page is being compiled (for example, there is a syntax error in a scriptlet), the server will return a `JasperException` and a message that includes the name of the JSP page's servlet and the line where the error occurred.

Once the page has been translated and compiled, the JSP page's servlet for the most part follows the servlet life cycle described in Servlet Life Cycle (page 574):

1. If an instance of the JSP page's servlet does not exist, the container
   a. Loads the JSP page's servlet class
   b. Instantiates an instance of the servlet class
   c. Initializes the servlet instance by calling the `jspInit` method
2. The container invokes the `_jspService` method, passing a request and response object.

If the container needs to remove the JSP page's servlet, it calls the `jspDestroy` method.

Execution

You can control various JSP page execution parameters by using `page` directives. The directives that pertain to buffering output and handling errors are discussed here. Other directives are covered in the context of specific page authoring tasks throughout the chapter.

Buffering

When a JSP page is executed, output written to the response object is automatically buffered. You can set the size of the buffer with the following page directive:

```
<%@ page buffer="none|xxxkb" %>
```

A larger buffer allows more content to be written before anything is actually sent back to the client, thus providing the JSP page with more time to set appropriate status codes and headers or to forward to another Web resource. A smaller buffer decreases server memory load and allows the client to start receiving data more quickly.

Handling Errors

Any number of exceptions can arise when a JSP page is executed. To specify that the Web container should forward control to an error page if an exception occurs, include the following `page` directive at the beginning of your JSP page:

```
<%@ page errorPage="file_name" %>
```

The Duke's Bookstore application page `initdestroy.jsp` contains the directive

```
<%@ page errorPage="errorpage.jsp"%>
```

The beginning of `errorpage.jsp` indicates that it is serving as an error page with the following `page` directive:

```
<%@ page isErrorPage="true|false" %>
```

This directive makes the exception object (of type `javax.servlet.jsp.JspException`) available to the error page, so that you can retrieve, interpret, and possibly display information about the cause of the exception in the error page.

Note: You can also define error pages for the WAR that contains a JSP page. If error pages are defined for both the WAR and a JSP page, the JSP page's error page takes precedence.

Initializing and Finalizing a JSP Page

You can customize the initialization process to allow the JSP page to read persistent configuration data, initialize resources, and perform any other one-time activities by overriding the `jspInit` method of the `JspPage` interface. You release resources using the `jspDestroy` method. The methods are defined using JSP declarations, discussed in Declarations (page 619).

The bookstore example page `initdestroy.jsp` defines the `jspInit` method to retrieve the object `database.BookDBAO` that accesses the bookstore database and stores a reference to the bean in `bookDBAO`.

```
private BookDBAO bookDBAO;
public void jspInit() {
bookDBAO =
  (BookDBAO)getServletContext().getAttribute("bookDB");
  if (bookDBAO == null)
     System.out.println("Couldn’t get database.");
}
```

When the JSP page is removed from service, the `jspDestroy` method releases the `BookDBAO` variable.

```
public void jspDestroy() {
  bookDBAO = null;
}
```

Since the enterprise bean is shared between all the JSP pages, it should be initialized when the application is started, instead of in each JSP page. Java Servlet technology provides application life-cycle events and listener classes for this purpose. As an exercise, you can move the code that manages the creation of the enterprise bean to a context listener class. See Handling Servlet Life Cycle Events (page 575) for the context listener that initializes the Java Servlet version of the bookstore application.

Creating Static Content

You create static content in a JSP page by simply writing it as if you were creating a page that consisted only of that content. Static content can be expressed in any text-based format, such as HTML, WML, and XML. The default format is HTML. If you want to use a format other than HTML, you include a `page` directive with the `contentType` attribute set to the format type at the beginning of your JSP page. For example, if you want a page to contain data expressed in the wireless markup language (WML), you need to include the following directive:

```
<%@ page contentType="text/vnd.wap.wml"%>
```

A registry of content type names is kept by the IANA at:

```
ftp://ftp.isi.edu/in-notes/iana/assignments/media-types
```

Creating Dynamic Content

You create dynamic content by accessing Java programming language objects from within scripting elements.

Using Objects within JSP Pages

You can access a variety of objects, including enterprise beans and JavaBeans components, within a JSP page. JSP technology automatically makes some objects available, and you can also create and access application-specific objects.

Implicit Objects

Implicit objects are created by the Web container and contain information related to a particular request, page, or application. Many of the objects are defined by the Java Servlet technology underlying JSP technology and are discussed at length in Chapter 14. Table 15–2 summarizes the implicit objects.

Table 15–2 Implicit Objects

Variable	Class	Description
`application`	`javax.servlet.ServletContext`	The context for the JSP page's servlet and any Web components contained in the same application. See Accessing the Web Context (page 598).
`config`	`javax.servlet.ServletConfig`	Initialization information for the JSP page's servlet.

Table 15–2 Implicit Objects (Continued)

Variable	Class	Description
`exception`	`java.lang.Throwable`	Accessible only from an error page. See Handling Errors (page 614).
`out`	`javax.servlet.jsp.JspWriter`	The output stream.
`page`	`java.lang.Object`	The instance of the JSP page's servlet processing the current request. Not typically used by JSP page authors.
`pageContext`	`javax.servlet.jsp.PageContext`	The context for the JSP page. Provides a single API to manage the various scoped attributes described in Using Scope Objects (page 578). This API is used extensively when implementing tag handlers (see Tag Handlers, page 645).
`request`	subtype of `javax.servlet.ServletRequest`	The request triggering the execution of the JSP page. See Getting Information from Requests (page 583).
`response`	subtype of `javax.servlet.ServletResponse`	The response to be returned to the client. Not typically used by JSP page authors.
`session`	`javax.servlet.http.HttpSession`	The session object for the client. See Maintaining Client State (page 599).

Application-Specific Objects

When possible, application behavior should be encapsulated in objects so that page designers can focus on presentation issues. Objects can be created by developers who are proficient in the Java programming language and in accessing databases and other services. There are four ways to create and use objects within a JSP page:

- Instance and class variables of the JSP page's servlet class are created in *declarations* and accessed in *scriptlets* and *expressions*.
- Local variables of the JSP page's servlet class are created and used in *scriptlets* and *expressions*.
- Attributes of scope objects (see Using Scope Objects, page 578) are created and used in *scriptlets* and *expressions*.
- JavaBeans components can be created and accessed using streamlined JSP elements. These elements are discussed in JavaBeans Components in JSP Pages (page 627). You can also create a JavaBeans component in a declaration or scriptlet and invoke the methods of a JavaBeans component in a scriptlet or expression.

Declarations, scriptlets, and expressions are described in JSP Scripting Elements (page 619).

Shared Objects

The conditions affecting concurrent access to shared objects described in Controlling Concurrent Access to Shared Resources (page 579) apply to objects accessed from JSP pages that run as multithreaded servlets. You can indicate how a Web container should dispatch multiple client requests with the following `page` directive:

```
<%@ page isThreadSafe="true|false" %>
```

When `isThreadSafe` is set to `true`, the Web container may choose to dispatch multiple concurrent client requests to the JSP page. This is the *default* setting. If using `true`, you must ensure that you properly synchronize access to any shared objects defined at the page level. This includes objects created within declarations, JavaBeans components with page scope, and attributes of the `page` scope object.

If `isThreadSafe` is set to `false`, requests are dispatched one at a time, in the order they were received, and access to page level objects does not have to be controlled. However, you still must ensure that access to attributes of the `application` or `session` scope objects and to JavaBeans components with application or session scope is properly synchronized.

JSP Scripting Elements

JSP scripting elements are used to create and access objects, define methods, and manage the flow of control. Since one of the goals of JSP technology is to separate static template data from the code needed to dynamically generate content, very sparing use of JSP scripting is recommended. Much of the work that requires the use of scripts can be eliminated by using custom tags, described in Custom Tags in JSP Pages (page 637).

JSP technology allows a container to support any scripting language that can call Java objects. If you wish to use a scripting language other than the default, `java`, you must specify it in a `page` directive at the beginning of a JSP page:

```
<%@ page language="scripting language" %>
```

Since scripting elements are converted to programming language statements in the JSP page's servlet class, you must import any classes and packages used by a JSP page. If the page language is `java`, you import a class or package with the `page` directive:

```
<%@ page import="packagename.*, fully_qualified_classname" %>
```

For example, the bookstore example page `showcart.jsp` imports the classes needed to implement the shopping cart with the following directive:

```
<%@ page import="java.util.*, cart.*" %>
```

Declarations

A *JSP declaration* is used to declare variables and methods in a page's scripting language. The syntax for a declaration is as follows:

```
<%! scripting language declaration %>
```

When the scripting language is the Java programming language, variables and methods in JSP declarations become declarations in the JSP page's servlet class.

The bookstore example page `initdestroy.jsp` defines an instance variable named `bookDBAO` and the initialization and finalization methods `jspInit` and `jspDestroy` discussed earlier in a declaration:

```
<%!
  private BookDBAO bookDBAO;

  public void jspInit() {
     ...
  }
  public void jspDestroy() {
     ...
  }
%>
```

Scriptlets

A *JSP scriptlet* is used to contain any code fragment that is valid for the scripting language used in a page. The syntax for a scriptlet is as follows:

```
<%
  scripting language statements
%>
```

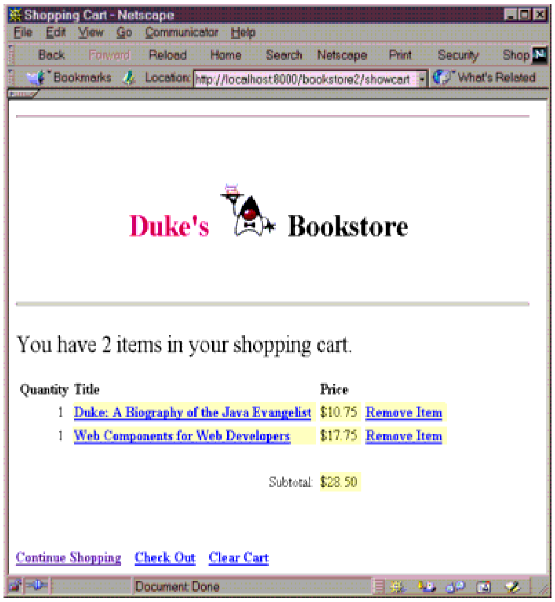
When the scripting language is set to `java`, a scriptlet is transformed into a Java programming language statement fragment and is inserted into the service method of the JSP page's servlet. A programming language variable created within a scriptlet is accessible from anywhere within the JSP page.

The JSP page `showcart.jsp` contains a scriptlet that retrieves an iterator from the collection of items maintained by a shopping cart and sets up a construct to loop through all the items in the cart. Inside the loop, the JSP page extracts properties of the book objects and formats them using HTML markup. Since the `while` loop opens a block, the HTML markup is followed by a scriptlet that closes the block.

```
<%
  Iterator i = cart.getItems().iterator();
  while (i.hasNext()) {
     ShoppingCartItem item =
        (ShoppingCartItem)i.next();
     BookDetails bd = (BookDetails)item.getItem();
%>

     <tr>
```

```
        <td align="right" bgcolor="#ffffff">
        <%=item.getQuantity()%>
        </td>
        <td bgcolor="#ffffaa">
        <strong><a href="
        <%=request.getContextPath()%>/bookdetails?bookId=
        <%=bd.getBookId()%>"><%=bd.getTitle()%></a></strong>
        </td>
        ...
<%
   // End of while
   }
%>
```

The output appears in Figure 15–2.


Shopping Cart - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop
Bookmarks Location: http://localhost:8000/bookstore2/showcart What's Related
Duke's Bookstore
Continue Shopping Check Out Clear Cart
Document: Done

You have 2 items in your shopping cart.

Quantity	Title	Price	
1	Duke: A Biography of the Java Evangelist	$10.75	Remove Item
1	Web Components for Web Developers	$17.75	Remove Item
	Subtotal:	$28.50	

Figure 15–2 Duke’s Bookstore Shopping Cart

Expressions

A *JSP expression* is used to insert the value of a scripting language expression, converted into a string, into the data stream returned to the client. When the scripting language is the Java programming language, an expression is transformed into a statement that converts the value of the expression into a `String` object and inserts it into the implicit `out` object.

The syntax for an expression is as follows:

```
<%= scripting language expression %>
```

Note that a semicolon is not allowed within a JSP expression, even if the same expression has a semicolon when you use it within a scriptlet.

The following scriptlet retrieves the number of items in a shopping cart:

```
<%
  // Print a summary of the shopping cart
  int num = cart.getNumberOfItems();
  if (num > 0) {
%>
```

Expressions are then used to insert the value of `num` into the output stream and determine the appropriate string to include after the number:

```
<font size="+2">
<%=messages.getString("CartContents")%> <%=num%>
  <%=(num==1 ? <%=messages.getString("CartItem")%> :
  <%=messages.getString("CartItems"))%></font>
```

Including Content in a JSP Page

There are two mechanisms for including another Web resource in a JSP page: the `include` directive and the `jsp:include` element.

The `include` directive is processed when the JSP page is *translated* into a servlet class. The effect of the directive is to insert the text contained in another file—either static content or another JSP page—in the including JSP page. You would probably use the `include` directive to include banner content, copyright information, or any chunk of content that you might want to reuse in another page. The syntax for the `include` directive is as follows:

```
<%@ include file="filename" %>
```

For example, all the bookstore application pages include the file `banner.jsp` which contains the banner content, with the following directive:

```
<%@ include file="banner.jsp" %>
```

In addition, the pages `bookstore.jsp`, `bookdetails.jsp`, `catalog.jsp`, and `showcart.jsp` include JSP elements that create and destroy a database bean with the following directive:

```
<%@ include file="initdestroy.jsp" %>
```

Because you must statically put an `include` directive in each file that reuses the resource referenced by the directive, this approach has its limitations. For a more flexible approach to building pages out of content chunks, see A Template Tag Library (page 665).

The `jsp:include` element is processed when a JSP page is *executed*. The `include` action allows you to include either a static or dynamic resource in a JSP file. The results of including static and dynamic resources are quite different. If the resource is static, its content is inserted into the calling JSP file. If the resource is dynamic, the request is sent to the included resource, the included page is executed, and then the result is included in the response from the calling JSP page. The syntax for the `jsp:include` element is:

```
<jsp:include page="includedPage" />
```

Note: Tomcat will not reload a statically included page that has been modified unless the including page is also modified.

The `date` application introduced at the beginning of this chapter includes the page that generates the display of the localized date with the following statement:

```
<jsp:include page="date.jsp"/>
```

Transferring Control to Another Web Component

The mechanism for transferring control to another Web component from a JSP page uses the functionality provided by the Java Servlet API as described in Transferring Control to Another Web Component (page 597). You access this functionality from a JSP page with the `jsp:forward` element:

```
<jsp:forward page="/main.jsp" />
```

Note that if any data has already been returned to a client, the `jsp:forward` element will fail with an `IllegalStateException`.

jsp:param Element

When an `include` or `forward` element is invoked, the original request object is provided to the target page. If you wish to provide additional data to that page, you can append parameters to the request object with the `jsp:param` element:

```
<jsp:include page="..." >
  <jsp:param name="param1" value="value1"/>
</jsp:include>
```

Including an Applet

You can include an applet or JavaBeans component in a JSP page by using the `jsp:plugin` element. This element generates HTML that contains the appropriate client-browser-dependent constructs (`<object>` or `<embed>`) that will result in the download of the Java Plug-in software (if required) and client-side component and subsequent execution of any client-side component. The syntax for the `jsp:plugin` element is as follows:

```
<jsp:plugin
  type="bean|applet"
  code="objectCode"
  codebase="objectCodebase"
  { align="alignment" }
  { archive="archiveList" }
  { height="height" }
  { hspace="hspace" }
```

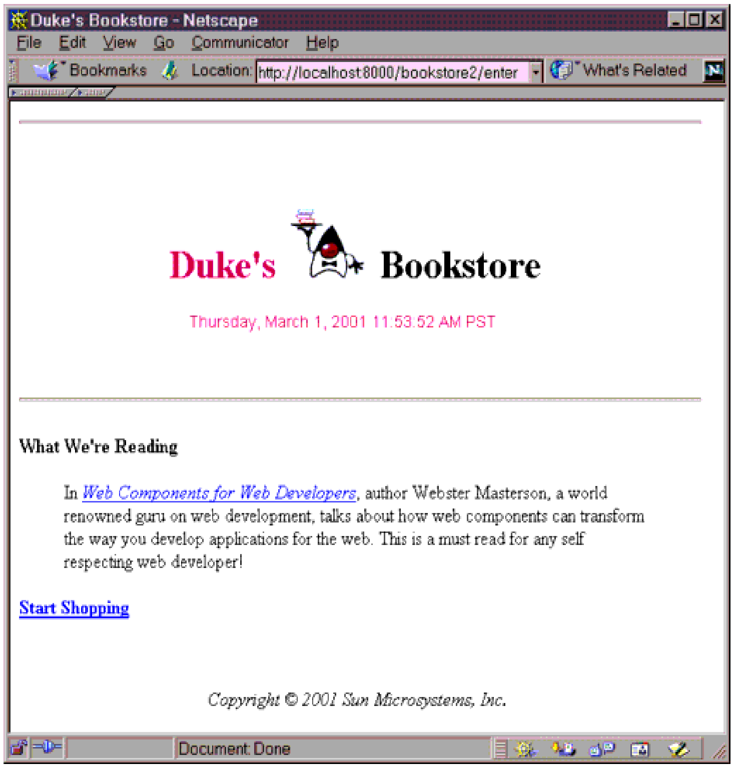
```
    { jreversion="jreversion" }
    { name="componentName" }
    { vspace="vspace" }
    { width="width" }
    { nspluginurl="url" }
    { iepluginurl="url" } >
    { <jsp:params>
      { <jsp:param name="paramName" value= paramValue" /> }+
    </jsp:params> }
    { <jsp:fallback> arbitrary_text </jsp:fallback> }
  </jsp:plugin>
```

The `jsp:plugin` tag is replaced by either an `<object>` or `<embed>` tag as appropriate for the requesting client. The attributes of the `jsp:plugin` tag provide configuration data for the presentation of the element as well as the version of the plug-in required. The `nspluginurl` and `iepluginurl` attributes specify the URL where the plug-in can be downloaded.

The `jsp:param` elements specify parameters to the applet or JavaBeans component. The `jsp:fallback` element indicates the content to be used by the client browser if the plug-in cannot be started (either because `<object>` or `<embed>` is not supported by the client or because of some other problem).

If the plug-in can start but the applet or JavaBeans component cannot be found or started, a plug-in-specific message will be presented to the user, most likely a pop-up window reporting a `ClassNotFoundException`.

The Duke's Bookstore page `banner.jsp` that creates the banner displays a dynamic digital clock generated by `DigitalClock`:


Duke's Bookstore - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://localhost:8000/bookstore2/enter What's Related
Duke's Bookstore
Thursday, March 1, 2001 11:53:52 AM PST
Document: Done

What We're Reading

In *Web Components for Web Developers*, author Webster Masterson, a world renowned guru on web development, talks about how web components can transform the way you develop applications for the web. This is a must read for any self respecting web developer!

Start Shopping

Copyright © 2001 Sun Microsystems, Inc.

Figure 15–3 Duke's Bookstore with Applet

The `jsp:plugin` element used to download the applet follows:

```
<jsp:plugin
  type="applet"
  code="DigitalClock.class"
  codebase="/bookstore2"
  jreversion="1.3"
  align="center" height="25" width="300"
  nspluginurl="http://java.sun.com/products/plugin/1.3.0_01
     /plugin-install.html"
  iepluginurl="http://java.sun.com/products/plugin/1.3.0_01
     /jinstall-130_01-win32.cab#Version=1,3,0,1" >
  <jsp:params>
     <jsp:param name="language"
```

```
            value="<%=request.getLocale().getLanguage()%>" />
          <jsp:param name="country"
            value="<%=request.getLocale().getCountry()%>" />
          <jsp:param name="bgcolor" value="FFFFFF" />
          <jsp:param name="fgcolor" value="CC0066" />
        </jsp:params>
          <jsp:fallback>
          <p>Unable to start plugin.</p>
        </jsp:fallback>
      </jsp:plugin>
```

JavaBeans Components in JSP Pages

JavaBeans components are Java classes that can be easily reused and composed together into applications. Any Java class that follows certain design conventions can be a JavaBeans component.

JavaServer Pages technology directly supports using JavaBeans components with JSP language elements. You can easily create and initialize beans and get and set the values of their properties. This chapter provides basic information about JavaBeans components and the JSP language elements for accessing JavaBeans components in your JSP pages. For further information about the JavaBeans component model see `http://java.sun.com/products/javabeans`.

JavaBeans Component Design Conventions

JavaBeans component design conventions govern the properties of the class and govern the public methods that give access to the properties.

A JavaBeans component property can be

- Read/write, read-only, or write-only
- Simple, which means it contains a single value, or indexed, which means it represents an array of values

There is no requirement that a property be implemented by an instance variable; the property must simply be accessible using public methods that conform to certain conventions:

- For each readable property, the bean must have a method of the form

  ```
  PropertyClass getProperty() { ... }
  ```

- For each writable property, the bean must have a method of the form

  ```
  setProperty(PropertyClass pc) { ... }
  ```

In addition to the property methods, a JavaBeans component must define a constructor that takes no parameters.

The Duke's Bookstore application JSP pages `enter.jsp`, `bookdetails.jsp`, `catalog.jsp`, and `showcart.jsp` use the `database.BookDB` and `database.BookDetails` JavaBeans components. `BookDB` provides a JavaBeans component front end to the access object `BookDBAO`. Both beans are used extensively by bean-oriented custom tags (see Custom Tags in JSP Pages, page 637). The JSP pages `showcart.jsp` and `cashier.jsp` use `cart.ShoppingCart` to represent a user's shopping cart.

The JSP pages `catalog.jsp`, `showcart.jsp`, and `cashier.jsp` use the `util.Currency` JavaBeans component to format currency in a locale-sensitive manner. The bean has two writable properties, `locale` and `amount`, and one readable property, `format`. The `format` property does not correspond to any instance variable, but returns a function of the `locale` and `amount` properties.

```
public class Currency {
  private Locale locale;
  private double amount;
  public Currency() {
    locale = null;
    amount = 0.0;
  }
  public void setLocale(Locale l) {
    locale = l;
  }
  public void setAmount(double a) {
    amount = a;
  }
  public String getFormat() {
    NumberFormat nf =
```

```
            NumberFormat.getCurrencyInstance(locale);
        return nf.format(amount);
    }
}
```

Why Use a JavaBeans Component?

A JSP page can create and use any type of Java programming language object within a declaration or scriptlet. The following scriptlet creates the bookstore shopping cart and stores it as a session attribute:

```
<%
    ShoppingCart cart = (ShoppingCart)session.
        getAttribute("cart");
    // If the user has no cart, create a new one
    if (cart == null) {
        cart = new ShoppingCart();
        session.setAttribute("cart", cart);
    }
%>
```

If the shopping cart object conforms to JavaBeans conventions, JSP pages can use JSP elements to create and access the object. For example, the Duke's Bookstore pages `bookdetails.jsp`, `catalog.jsp`, and `showcart.jsp` replace the scriptlet with the much more concise JSP `useBean` element:

```
<jsp:useBean id="cart" class="cart.ShoppingCart"
    scope="session"/>
```

Creating and Using a JavaBeans Component

You declare that your JSP page will use a JavaBeans component using either one of the following formats:

```
<jsp:useBean id="beanName"
    class="fully_qualified_classname" scope="scope"/>
```

or

```
<jsp:useBean id="beanName"
  class="fully_qualified_classname" scope="scope">
  <jsp:setProperty .../>
</jsp:useBean>
```

The second format is used when you want to include `jsp:setProperty` statements, described in the next section, for initializing bean properties.

The `jsp:useBean` element declares that the page will use a bean that is stored within and accessible from the specified scope, which can be `application`, `session`, `request`, or `page`. If no such bean exists, the statement creates the bean and stores it as an attribute of the scope object (see Using Scope Objects, page 578). The value of the `id` attribute determines the *name* of the bean in the scope and the *identifier* used to reference the bean in other JSP elements and scriptlets.

Note: In JSP Scripting Elements (page 619), we mentioned that you must import any classes and packages used by a JSP page. This rule is slightly altered if the class is only referenced by `useBean` elements. In these cases, you must only import the class if the class is in the unnamed package. For example, in What Is a JSP Page? (page 607), the page `index.jsp` imports the `MyLocales` class. However, in the Duke's Bookstore example, all classes are contained in packages and thus are not explicitly imported.

The following element creates an instance of `Currency` if none exists, stores it as an attribute of the `session` object, and makes the bean available throughout the session by the identifier `currency`:

```
<jsp:useBean id="currency" class="util.Currency"
  scope="session"/>
```

Setting JavaBeans Component Properties

There are two ways to set JavaBeans component properties in a JSP page: with the `jsp:setProperty` element or with a scriptlet

```
<% beanName.setPropName(value); %>
```

The syntax of the `jsp:setProperty` element depends on the source of the property value. Table 15–3 summarizes the various ways to set a property of a JavaBeans component using the `jsp:setProperty` element.

Table 15–3 Setting JavaBeans Component Properties

Value Source	Element Syntax

String constant	`<jsp:setProperty name="`*`beanName`*`"`
	`property="`*`propName`*`" value="`*`string constant`*`"/>`

Request parameter	`<jsp:setProperty name="`*`beanName`*`"`
	`property="`*`propName`*`" param="`*`paramName`*`"/>`

Request parameter name matches bean property	`<jsp:setProperty name="`*`beanName`*`"`
	`property="`*`propName`*`"/>`
	
	`<jsp:setProperty name="`*`beanName`*`"`
	`property="*"/>`

Expression	`<jsp:setProperty name="`*`beanName`*`"`
	`property="`*`propName`*`"`
	`value="<%= `*`expression`*` %>"/>`

	1. *`beanName`* must be the same as that specified for the `id` attribute in a `useBean` element. 2. There must be a *`setPropName`* method in the JavaBeans component. 3. *`paramName`* must be a request parameter name.

A property set from a constant string or request parameter must have a type listed in Table 15–4. Since both a constant and request parameter are strings, the Web container automatically converts the value to the property's type; the conversion applied is shown in the table. `String` values can be used to assign values to a property that has a `PropertyEditor` class. When that is the case, the `setAsText(String)` method is used. A conversion failure arises if the method throws an `IllegalArgumentException`. The value assigned to an indexed property must be an array, and the rules just described apply to the elements.

Table 15–4 Valid Value Assignments

Property Type	**Conversion on String Value**
Bean Property	Uses `setAsText(`*`string-literal`*`)`
`boolean` or `Boolean`	As indicated in `java.lang.Boolean.valueOf(String)`
`byte` or `Byte`	As indicated in `java.lang.Byte.valueOf(String)`
`char` or `Character`	As indicated in `java.lang.String.charAt(0)`
`double` or `Double`	As indicated in `java.lang.Double.valueOf(String)`
`int` or `Integer`	As indicated in `java.lang.Integer.valueOf(String)`
`float` or `Float`	As indicated in `java.lang.Float.valueOf(String)`
`long` or `Long`	As indicated in `java.lang.Long.valueOf(String)`
`short` or `Short`	As indicated in `java.lang.Short.valueOf(String)`
`Object`	new `String(`*`string-literal`*`)`

You would use a runtime expression to set the value of a property whose type is a compound Java programming language type. Recall from Expressions (page 621) that a JSP expression is used to insert the value of a scripting language expression, converted into a String, into the stream returned to the client. When used within a `setProperty` element, an expression simply returns its value; no automatic conversion is performed. As a consequence, the type returned from an expression must match or be castable to the type of the property.

The Duke's Bookstore application demonstrates how to use the `setProperty` element and a scriptlet to set the current book for the database helper bean. For example, `bookstore3/web/bookdetails.jsp` uses the form:

```
<jsp:setProperty name="bookDB" property="bookId"/>
```

whereas `bookstore2/web/bookdetails.jsp` uses the form:

```
<% bookDB.setBookId(bookId); %>
```

The following fragments from the page `bookstore3/web/showcart.jsp` illustrate how to initialize a currency bean with a `Locale` object and amount determined by evaluating request-time expressions. Because the first initialization is nested in a `useBean` element, it is only executed when the bean is created.

```
<jsp:useBean id="currency" class="util.Currency"
  scope="session">
  <jsp:setProperty name="currency" property="locale"
    value="<%= request.getLocale() %>"/>
</jsp:useBean>

<jsp:setProperty name="currency" property="amount"
  value="<%=cart.getTotal()%>"/>
```

Retrieving JavaBeans Component Properties

There are several ways to retrieve JavaBeans component properties. Two of the methods (the `jsp:getProperty` element and an expression) convert the value of the property into a `String` and insert the value into the current implicit `out` object:

- `<jsp:getProperty name="`*`beanName`*`" property="`*`propName`*`"/>`
- `<%= `*`beanName`*`.get`*`PropName`*`() %>`

For both methods, *`beanName`* must be the same as that specified for the `id` attribute in a `useBean` element, and there must be a `get`*`PropName`* method in the JavaBeans component.

If you need to retrieve the value of a property without converting it and inserting it into the out object, you must use a scriptlet:

```
<% Object o = beanName.getPropName(); %>
```

Note the differences between the expression and the scriptlet; the expression has an = after the opening % and does not terminate with a semicolon, as does the scriptlet.

The Duke’s Bookstore application demonstrates how to use both forms to retrieve the formatted currency from the currency bean and insert it into the page. For example, `bookstore3/web/showcart.jsp` uses the form

```
<jsp:getProperty name="currency" property="format"/>
```

whereas `bookstore2/web/showcart.jsp` uses the form:

```
<%= currency.getFormat() %>
```

The Duke’s Bookstore application page `bookstore2/web/showcart.jsp` uses the following scriptlet to retrieve the number of books from the shopping cart bean and open a conditional insertion of text into the output stream:

```
<%
  // Print a summary of the shopping cart
  int num = cart.getNumberOfItems();
  if (num > 0) {
%>
```

Although scriptlets are very useful for dynamic processing, using custom tags (see Custom Tags in JSP Pages, page 637) to access object properties and perform flow control is considered to be a better approach. For example, `bookstore3/web/showcart.jsp` replaces the scriptlet with the following custom tags:

```
<bean:define id="num" name="cart" property="numberOfItems" />
<logic:greaterThan name="num" value="0" >
```

Figure 15–4 summarizes where various types of objects are stored and how those objects can be accessed from a JSP page. Objects created by the `jsp:useBean` tag are stored as attributes of the scope objects and can be accessed by `jsp:[get|set]Property` tags and in scriptlets and expressions. Objects created in declarations and scriptlets are stored as variables of the JSP page's servlet class and can be accessed in scriptlets and expressions.

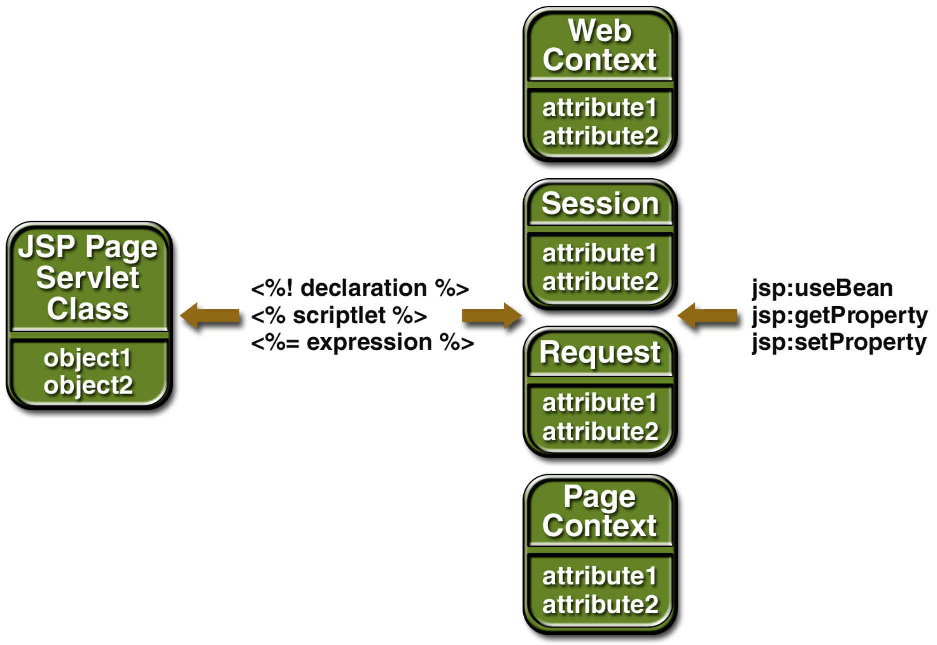


Figure 15–4 Accessing Objects From a JSP Page

Extending the JSP Language

You can perform a wide variety of dynamic processing tasks, including accessing databases, using enterprise services such as e-mail and directories, and flow control, with JavaBeans components in conjunction with scriptlets. One of the drawbacks of scriptlets, however, is that they tend to make JSP pages more difficult to maintain. Alternatively, JSP technology provides a mechanism, called *custom tags*, that allows you to encapsulate dynamic functionality in objects that are accessed through extensions to the JSP language. Custom tags bring the benefits of another level of componentization to JSP pages.

For example, recall the scriptlet used to loop through and display the contents of the Duke’s Bookstore shopping cart:

```
<%
  Iterator i = cart.getItems().iterator();
  while (i.hasNext()) {
     ShoppingCartItem item =
        (ShoppingCartItem)i.next();
     ...
%>
     <tr>
     <td align="right" bgcolor="#ffffff">
     <%=item.getQuantity()%>
     </td>
     ...
<%
  }
%>
```

An `iterate` custom tag eliminates the code logic and manages the scripting variable `item` that references elements in the shopping cart:

```
<logic:iterate id="item"
  collection="<%=cart.getItems()%>">
  <tr>
  <td align="right" bgcolor="#ffffff">
  <%=item.getQuantity()%>
  </td>
  ...
</logic:iterate>
```

Custom tags are packaged and distributed in a unit called a *tag library*. The syntax of custom tags is the same as that used for the JSP elements, namely `<prefix:tag>`, for custom tags, however, `prefix` is defined by the *user* of the tag library, and `tag` is defined by the *tag developer*. Custom Tags in JSP Pages (page 637) explains how to use and develop custom tags.

Further Information

For further information on JavaServer Pages technology see:

- Resources listed on the Web site `http://java.sun.com/products/jsp`.
- The JavaServer Pages 1.2 Specification for a complete description of the syntax and semantics of JSP technology.

16

Custom Tags in JSP Pages

Stephanie Bodoff

THE standard JSP tags for invoking operations on JavaBeans components and performing request dispatching simplify JSP page development and maintenance. JSP technology also provides a mechanism for encapsulating other types of dynamic functionality in *custom tags*, which are extensions to the JSP language. Custom tags are usually distributed in the form of a *tag library*, which defines a set of related custom tags and contains the objects that implement the tags.

Some examples of tasks that can be performed by custom tags include operations on implicit objects, processing forms, accessing databases and other enterprise services such as e-mail and directories, and performing flow control. JSP tag libraries are created by developers who are proficient at the Java programming language and expert in accessing data and other services, and are used by Web application designers who can focus on presentation issues rather than being concerned with how to access enterprise services. As well as encouraging division of labor between library developers and library users, custom tags increase productivity by encapsulating recurring tasks so that they can be reused across more than one application.

Tag libraries are receiving a great deal of attention in the JSP technology community. For more information about tag libraries and for pointers to some freely-available libraries, see

```
http://java.sun.com/products/jsp/taglibraries.html
```

What Is a Custom Tag?

A custom tag is a user-defined JSP language element. When a JSP page containing a custom tag is translated into a servlet, the tag is converted to operations on an object called a *tag handler*. The Web container then invokes those operations when the JSP page's servlet is executed.

Custom tags have a rich set of features. They can

- Be customized via attributes passed from the calling page.
- Access all the objects available to JSP pages.
- Modify the response generated by the calling page.
- Communicate with each other. You can create and initialize a JavaBeans component, create a variable that refers to that bean in one tag, and then use the bean in another tag.
- Be nested within one another, allowing for complex interactions within a JSP page.

The Example JSP Pages

This chapter describes the tasks involved in using and defining tags. The chapter illustrates the tasks with excerpts from the JSP version of the Duke's Bookstore application discussed in The Example JSP Pages (page 610) rewritten to take advantage of two tag libraries: Struts and tutorial-template. The third section in the chapter, Examples (page 661), describes two tags in detail: the `iterate` tag from Struts and the set of tags in the tutorial-template tag library.

The Struts tag library provides a framework for building internationalized Web applications that implement the Model-View-Controller design pattern. Struts includes a comprehensive set of utility custom tags for handling:

- HTML forms
- Templates
- JavaBeans components
- Logic processing

The Duke's Bookstore application uses tags from the Struts `bean` and `logic` sublibraries.

The tutorial-template tag library defines a set of tags for creating an application template. The template is a JSP page with placeholders for the parts that need to change with each screen. Each of these placeholders is referred to as a parameter of the template. For example, a simple template could include a title parameter for the top of the generated screen and a body parameter to refer to a JSP page for the custom content of the screen. The template is created with a set of nested tags—`definition`, `screen`, and `parameter`—that are used to build a table of screen definitions for Duke's Bookstore and with an `insert` tag to insert parameters from the table into the screen.

Figure 16–1 shows the flow of a request through the following Duke's Bookstore Web components:

- `template.jsp`, which determines the structure of each screen. It uses the `insert` tag to compose a screen from subcomponents.
- `screendefinitions.jsp`, which defines the subcomponents used by each screen. All screens have the same banner, but different title and body content (specified by the JSP Pages column in Table 15–1).
- `Dispatcher`, a servlet, which processes requests and forwards to `template.jsp`.

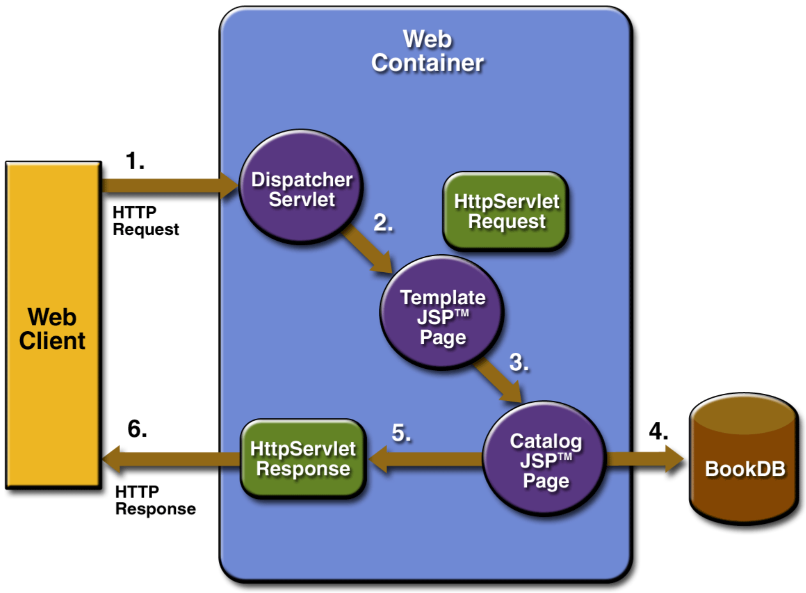


Figure 16–1 Request Flow Through Duke's Bookstore Components

The source code for the Duke's Bookstore application is located in the `docs/tutorial/examples/web/bookstore3` directory created when you unzip the tutorial bundle (see Running the Examples, page xiii). To build, deploy, and run the example:

1. Download Struts version 1.0.2 from

   ```
   http://jakarta.apache.org/builds/jakarta-struts/
   release/v1.0.2/
   ```

2. Unpack Struts and copy `struts-bean.tld`, `struts-logic.tld`, and `struts.jar` from `jakarta-struts-1.0/lib` to `<`*`JWSDP_HOME`*`>/docs/tutorial/examples/web/bookstore3`.
3. In a terminal window, go to `<`*`JWSDP_HOME`*`>/docs/tutorial/examples/bookstore3`.
4. Run `ant build`. The `build` target will spawn any necessary compilations and copy files to the `<`*`JWSDP_HOME`*`>/docs/tutorial/examples/web/bookstore3/build` directory.

5. Make sure Tomcat is started.
6. Run `ant install`. The `install` target notifies Tomcat that the new context is available.
7. Start the PointBase database server and populate the database if you have not done so already (see Accessing Databases from Web Applications, page 109).
8. Open the bookstore URL `http://localhost:8080/bookstore3/enter`.

See Common Problems and Their Solutions (page 87) and Troubleshooting (page 574) for help with diagnosing common problems.

Using Tags

This section describes how a JSP page uses tags and introduces the different types of tags.

To use a tag, a page author must do two things:

- Declare the tag library containing the tag
- Make the tag library implementation available to the Web application

Declaring Tag Libraries

You declare that a JSP page will use tags defined in a tag library by including a `taglib` directive in the page before any custom tag is used:

```
<%@ taglib uri="/WEB-INF/tutorial-template.tld" prefix="tt" %>
```

The `uri` attribute refers to a URI that uniquely identifies the tag library descriptor (TLD), described in Tag Library Descriptors (page 647). This URI can be direct or indirect. The `prefix` attribute defines the prefix that distinguishes tags defined by a given tag library from those provided by other tag libraries.

Tag library descriptor file names must have the extension `.tld`. TLD files are stored in the `WEB-INF` directory of the WAR or in a subdirectory of `WEB-INF`. You can reference a TLD directly and indirectly.

The following `taglib` directive directly references a TLD filename:

```
<%@ taglib uri="/WEB-INF/tutorial-template.tld" prefix="tt" %>
```

This `taglib` directive uses a short logical name to indirectly reference the TLD:

```
<%@ taglib uri="/tutorial-template" prefix="tt" %>
```

You map a logical name to an absolute location in the Web application deployment descriptor. To map the logical name `/tutorial-template` to the absolute location `/WEB-INF/tutorial-template.tld`, you add a `taglib` element to `web.xml`:

```
<taglib>
  <taglib-uri>/tutorial-template</taglib-uri>
  <taglib-location>
    /WEB-INF/tutorial-template.tld
  </taglib-location>
</taglib>
```

Making the Tag Library Implementation Available

A tag library implementation can be made available to a Web application in two basic ways. The classes implementing the tag handlers can be stored in an unpacked form in the `WEB-INF/classes` subdirectory of the Web application. Alternatively, if the library is distributed as a JAR, it is stored the `WEB-INF/lib` directory of the Web application. A tag library shared between more than one application is stored in the `<`*`JWSDP_HOME`*`>/common/lib` directory of the Java WSDP.

Types of Tags

JSP custom tags are written using XML syntax. They have a start tag and end tag, and possibly a body:

```
<tt:tag>
  body
</tt:tag>
```

A custom tag with no body is expressed as follows:

```
<tt:tag />
```

Simple Tags

A simple tag contains no body and no attributes:

```
<tt:simple />
```

Tags with Attributes

A custom tag can have attributes. Attributes are listed in the start tag and have the syntax `attr="value"`. Attribute values serve to customize the behavior of a custom tag just as parameters are used to customize the behavior of a method. You specify the types of a tag's attributes in a tag library descriptor, (see Tags with Attributes, page 650).

You can set an attribute value from a `String` constant or a runtime expression. The conversion process between the constants and runtime expressions and attribute types follows the rules described for JavaBeans component properties in Setting JavaBeans Component Properties (page 630).

The attributes of the Struts `logic:present` tag determine whether the body of the tag is evaluated. In the following example, an attribute specifies a request parameter named `Clear`:

```
<logic:present parameter="Clear">
```

The Duke's Bookstore application page `catalog.jsp` uses a runtime expression to set the value of the attribute that determines the collection of books over which the Struts `logic:iterate` tag iterates:

```
<logic:iterate collection="<%=bookDB.getBooks()%>"
  id="book" type="database.BookDetails">
```

Tags with Bodies

A custom tag can contain custom and core tags, scripting elements, HTML text, and tag-dependent body content between the start and end tag.

In the following example, the Duke's Bookstore application page `showcart.jsp` uses the Struts `logic:present` tag to clear the shopping cart and print a message if the request contains a parameter named `Clear`:

```
<logic:present parameter="Clear">
  <% cart.clear(); %>
  <font color="#ff0000" size="+2"><strong>
  You just cleared your shopping cart!
  </strong><br>&nbsp;<br></font>
</logic:present>
```

Choosing between Passing Information as Attributes or Body

As shown in the last two sections, it is possible to pass a given piece of data as an attribute of the tag or as the tag's body. Generally speaking, any data that is a simple string or can be generated by evaluating a simple expression is best passed as an attribute.

Tags That Define Scripting Variables

A custom tag can define a variable that can be used in scripts within a page. The following example illustrates how to define and use a scripting variable that contains an object returned from a JNDI lookup. Examples of such objects include enterprise beans, transactions, databases, environment entries, and so on:

```
<tt:lookup id="tx" type="UserTransaction"
  name="java:comp/UserTransaction" />
<% tx.begin(); %>
```

In the Duke's Bookstore application, several pages use bean-oriented tags from Struts to define scripting variables. For example, `bookdetails.jsp` uses the `bean:parameter` tag to create the `bookId` scripting variable and set it to the value of the `bookId` request parameter. The `jsp:setProperty` statement also sets the `bookId` property of the `bookDB` object to the value of the `bookId` request parameter. The `bean:define` tag retrieves the value of the bookstore database property `bookDetails` and defines the result as the scripting variable `book`:

```
<bean:parameter id="bookId" name="bookId" />
<jsp:setProperty name="bookDB" property="bookId"/>
<bean:define id="book" name="bookDB" property="bookDetails"
  type="database.BookDetails"/>
<h2><jsp:getProperty name="book" property="title"></h2>
```

Cooperating Tags

Custom tags can cooperate with each other through shared objects.

In the following example, `tag1` creates an object called `obj1`, which is then reused by `tag2`.

```
<tt:tag1 attr1="obj1" value1="value" />
<tt:tag2 attr1="obj1" />
```

In the next example, an object created by the enclosing tag of a group of nested tags is available to all inner tags. Since the object is not named, the potential for naming conflicts is reduced. This example illustrates how a set of cooperating nested tags would appear in a JSP page.

```
<tt:outerTag>
  <tt:innerTag />
</tt:outerTag>
```

The Duke's Bookstore page `template.jsp` uses a set of cooperating tags to define the screens of the application. These tags are described in A Template Tag Library (page 665).

Defining Tags

To define a tag, you need to:

- Develop a tag handler and helper classes for the tag
- Declare the tag in a tag library descriptor

This section describes the properties of tag handlers and TLDs and explains how to develop tag handlers and library descriptor elements for each type of tag introduced in the previous section.

Tag Handlers

A *tag handler* is an object invoked by a Web container to evaluate a custom tag during the execution of the JSP page that references the tag. Tag handlers must implement either the `Tag` or `BodyTag` interface. Interfaces can be used to take an existing Java object and make it a tag handler. For newly created handlers, you can use the `TagSupport` and `BodyTagSupport` classes as base classes. These classes and interfaces are contained in the `javax.servlet.jsp.tagext` package.

Tag handler methods defined by the `Tag` and `BodyTag` interfaces are called by the JSP page's servlet at various points during the evaluation of the tag. When the start tag of a custom tag is encountered, the JSP page's servlet calls methods to initialize the appropriate handler and then invokes the handler's `doStartTag` method. When the end tag of a custom tag is encountered, the handler's `doEndTag` method is invoked. Additional methods are invoked in between when a tag handler needs to interact with the body of the tag. For further information, see Tags with Bodies (page 653). In order to provide a tag handler implementation, you must implement the methods, summarized in Table 16–1, that are invoked at various stages of processing the tag.

Table 16–1 Tag Handler Methods

Tag Handler Type	**Methods**
Simple	`doStartTag, doEndTag, release`
Attributes	`doStartTag, doEndTag, set/getAttribute1...N, release`
Body, Evaluation and No Interaction	`doStartTag, doEndTag, release`
Body, Iterative Evaluation	`doStartTag, doAfterBody, doEndTag, release`
Body, Interaction	`doStartTag, doEndTag, release, doInitBody, doAfterBody, release`

A tag handler has access to an API that allows it to communicate with the JSP page. The entry point to the API is the page context object (`javax.servlet.jsp.PageContext`), through which a tag handler can retrieve all the other implicit objects (request, session, and application) accessible from a JSP page.

Implicit objects can have named attributes associated with them. Such attributes are accessed using `[set|get]Attribute` methods.

If the tag is nested, a tag handler also has access to the handler (called the *parent*) associated with the enclosing tag.

A set of related tag handler classes (a tag library) is usually packaged and deployed as a JAR archive.

Tag Library Descriptors

A *tag library descriptor* (TLD) is an XML document that describes a tag library. A TLD contains information about a library as a whole and about each tag contained in the library. TLDs are used by a Web container to validate the tags and by JSP page development tools.

TLD file names must have the extension `.tld`. TLD files are stored in the `WEB-INF` directory of the WAR file or in a subdirectory of `WEB-INF`.

A TLD must begin with an XML document prolog that specifies the version of XML and the document type definition (DTD):

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag
Library 1.2//EN"
"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
```

Tomcat supports version 1.1 and 1.2 DTDs. However, this chapter documents the 1.2 version because you should use the newer version in any tag libraries that you develop. The template library TLD, `tutorial-template.tld`, conforms to the 1.2 version. The Struts library TLDs conform to the 1.1 version of the DTD, which has fewer elements and uses slightly different names for some of the elements.

The root of a TLD is the `taglib` element. The subelements of `taglib` are listed in Table 16–2:

Table 16–2 `taglib` Subelements

Element	Description
`tlib-version`	The tag library’s version
`jsp-version`	The JSP specification version that the tag library requires
`short-name`	Optional name that could be used by a JSP page authoring tool to create names with a mnemonic value
`uri`	A URI that uniquely identifies the tag library

Table 16–2 `taglib` Subelements (Continued)

Element	Description
`display-name`	Optional name intended to be displayed by tools
`small-icon`	Optional small-icon that can be used by tools
`large-icon`	Optional large-icon that can be used by tools
`description`	Optional tag-specific information
`listener`	See listener Element (page 648)
`tag`	See tag Element (page 648)

listener Element

A tag library can specify some classes that are event listeners (see Handling Servlet Life Cycle Events, page 575). The listeners are listed in the TLD as `listener` elements, and the Web container will instantiate the listener classes and register them in a way analogous to listeners defined at the WAR level. Unlike WAR-level listeners, the order in which the tag library listeners are registered is undefined. The only subelement of the `listener` element is the `listener-class` element, which must contain the fully qualified name of the listener class.

tag Element

Each tag in the library is described by giving its name and the class of its tag handler, information on the scripting variables created by the tag, and information on the tag's attributes. Scripting variable information can be given directly in the TLD or through a tag extra info class (see Tags That Define Scripting Variables, page 644). Each attribute declaration contains an indication of whether the attribute is required, whether its value can be determined by request-time expressions, and the type of the attribute (see Attribute Element, page 651).

A tag is specified in a TLD in a `tag` element. The subelements of `tag` are listed in Table 16–3:

Table 16–3 `tag` Subelements

Element	Description
`name`	The unique tag name
`tag-class`	The fully-qualified name of the tag handler class
`tei-class`	Optional subclass of `javax.servlet.jsp.tagext.TagExtraInfo`. See Providing Information about the Scripting Variable (page 656).
`body-content`	The body content type. See body-content Element (page 650) and body-content Element (page 655).
`display-name`	Optional name intended to be displayed by tools
`small-icon`	Optional small-icon that can be used by tools
`large-icon`	Optional large-icon that can be used by tools
`description`	Optional tag-specific information
`variable`	Optional scripting variable information. See Providing Information about the Scripting Variable (page 656).
`attribute`	Tag attribute information. See Attribute Element (page 651).

The following sections describe the methods and TLD elements that you need to develop for each type of tag introduced in Types of Tags (page 642).

Simple Tags

Tag Handlers

The handler for a simple tag must implement the `doStartTag` and `doEndTag` methods of the `Tag` interface. The `doStartTag` method is invoked when the start tag is encountered. This method returns `SKIP_BODY` because a simple tag has no body. The `doEndTag` method is invoked when the end tag is encountered. The doEndTag method needs to return EVAL_PAGE if the rest of the page needs to be evaluated; otherwise, it should return SKIP_PAGE.

The simple tag discussed in the first section,

```
<tt:simple />
```

would be implemented by the following tag handler:

```
public SimpleTag extends TagSupport {
  public int doStartTag() throws JspException {
     try {
        pageContext.getOut().print("Hello.");
     } catch (Exception ex) {
        throw new JspTagException("SimpleTag: " +
           ex.getMessage());
     }
     return SKIP_BODY;
  }
  public int doEndTag() {
     return EVAL_PAGE;
  }
}
```

body-content Element

Tags without bodies must declare that their body content is empty using the body-content element:

```
<body-content>empty</body-content>
```

Tags with Attributes

Defining Attributes in a Tag Handler

For each tag attribute, you must define a property and get and set methods that conform to the JavaBeans architecture conventions in the tag handler. For example, the tag handler for the Struts logic:present tag,

```
<logic:present parameter="Clear">
```

contains the following declaration and methods:

```
protected String parameter = null;
public String getParameter() {
  return (this.parameter);
}
public void setParameter(String parameter) {
  this.parameter = parameter;
}
```

Note that if your attribute is named `id` and your tag handler inherits from the `TagSupport` class, you do not need to define the property and `set` and `get` methods because these are already defined by `TagSupport`.

A tag attribute whose value is a `String` can name an attribute of one of the implicit objects available to tag handlers. An implicit object attribute would be accessed by passing the tag attribute value to the `[set|get]Attribute` method of the implicit object. This is a good way to pass scripting variable names to a tag handler where they are associated with objects stored in the page context (See Implicit Objects, page 616).

Attribute Element

For each tag attribute, you must specify whether the attribute is required, whether the value can be determined by an expression, and, optionally, the type of the attribute in an `attribute` element. For static values the type is always `java.lang.String`. If the `rtexprvalue` element is `true` or `yes`, then the `type` element defines the return type expected from any expression specified as the value of the attribute.

```
<attribute>
  <name>attr1</name>
  <required>true|false|yes|no</required>
  <rtexprvalue>true|false|yes|no</rtexprvalue>
  <type>fully_qualified_type</type>
</attribute>
```

If a tag attribute is not required, a tag handler should provide a default value.

The `tag` element for the `logic:present` tag declares that the `parameter` attribute is not required (because the tag can also test for the presence of other entities such as bean properties) and that its value can be set by a runtime expression.

```
<tag>
  <name>present</name>
  <tag-class>org.apache.struts.taglib.
    logic.PresentTag</tag-class>
  <body-content>JSP</body-content>
  ...
  <attribute>
    <name>parameter</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  ...
</tag>
```

Attribute Validation

The documentation for a tag library should describe valid values for tag attributes. When a JSP page is translated, a Web container will enforce any constraints contained in the TLD element for each attribute.

The attributes passed to a tag can also be validated at translation time with the `isValid` method of a class derived from `TagExtraInfo`. This class is also used to provide information about scripting variables defined by the tag (see Providing Information about the Scripting Variable, page 656).

The `isValid` method is passed the attribute information in a `TagData` object, which contains attribute-value tuples for each of the tag's attributes. Since the validation occurs at translation time, the value of an attribute that is computed at request time will be set to `TagData.REQUEST_TIME_VALUE`.

The tag `<tt:twa attr1="value1"/>` has the following TLD `attribute` element:

```
<attribute>
  <name>attr1</name>
  <required>true</required>
  <rtexprvalue>true</rtexprvalue>
</attribute>
```

This declaration indicates that the value of `attr1` can be determined at runtime.

The following `isValid` method checks that the value of `attr1` is a valid Boolean value. Note that since the value of `attr1` can be computed at runtime, `isValid` must check whether the tag user has chosen to provide a runtime value.

```
public class TwaTEI extends TagExtraInfo {
  public boolean isValid(Tagdata data) {
    Object o = data.getAttribute("attr1");
    if (o != null && o != TagData.REQUEST_TIME_VALUE) {
      if (((String)o).toLowerCase().equals("true") ||
        ((String)o).toLowerCase().equals("false") )
        return true;
      else
        return false;
    }
    else
      return true;
  }
}
```

Tags with Bodies

Tag Handlers

A tag handler for a tag with a body is implemented differently depending on whether the tag handler needs to interact with the body or not. By interact, we mean that the tag handler reads or modifies the contents of the body.

Tag Handler Does Not Interact with the Body

If the tag handler does not need to interact with the body, the tag handler should implement the `Tag` interface (or be derived from `TagSupport`). If the body of the tag needs to be evaluated, the `doStartTag` method needs to return `EVAL_BODY_INCLUDE`; otherwise it should return `SKIP_BODY`.

If a tag handler needs to iteratively evaluate the body, it should implement the `IterationTag` interface or be derived from `TagSupport`. It should return `EVAL_BODY_AGAIN` from the `doStartTag` and `doAfterBody` methods if it determines that the body needs to be evaluated again.

Tag Handler Interacts with the Body

If the tag handler needs to interact with the body, the tag handler must implement `BodyTag` (or be derived from `BodyTagSupport`). Such handlers typically implement the `doInitBody` and the `doAfterBody` methods. These methods interact with body content passed to the tag handler by the JSP page's servlet.

Body content supports several methods to read and write its contents. A tag handler can use the body content's `getString` or `getReader` methods to extract information from the body, and the `writeOut(out)` method to write the body contents to an out stream. The writer supplied to the `writeOut` method is obtained using the tag handler's `getPreviousOut` method. This method is used to ensure that a tag handler's results are available to an enclosing tag handler.

If the body of the tag needs to be evaluated, the `doStartTag` method needs to return `EVAL_BODY_BUFFERED`; otherwise, it should return `SKIP_BODY`.

`doInitBody` Method

The `doInitBody` method is called after the body content is set but before it is evaluated. You generally use this method to perform any initialization that depends on the body content.

`doAfterBody` Method

The `doAfterBody` method is called *after* the body content is evaluated.

Like the `doStartTag` method, `doAfterBody` must return an indication of whether to continue evaluating the body. Thus, if the body should be evaluated again, as would be the case if you were implementing an iteration tag, `doAfterBody` should return `EVAL_BODY_BUFFERED`; otherwise, `doAfterBody` should return `SKIP_BODY`.

`release` Method

A tag handler should reset its state and release any private resources in the `release` method.

The following example reads the content of the body (which contains a SQL query) and passes it to an object that executes the query. Since the body does not need to be reevaluated, `doAfterBody` returns `SKIP_BODY`.

```
public class QueryTag extends BodyTagSupport {
  public int doAfterBody() throws JspTagException {
    BodyContent bc = getBodyContent();
    // get the bc as string
    String query = bc.getString();
    // clean up
    bc.clearBody();
    try {
      Statement stmt = connection.createStatement();
      result = stmt.executeQuery(query);
    } catch (SQLException e) {
      throw new JspTagException("QueryTag: " +
            e.getMessage());
      }
      return SKIP_BODY;
   }
}
```

body-content Element

For tags that have a body, you must specify the type of the body content using the `body-content` element:

```
<body-content>JSP|tagdependent</body-content>
```

Body content containing custom and core tags, scripting elements, and HTML text is categorized as `JSP`. This is the value declared for the Struts `logic:present` tag. All other types of body content—for example—SQL statements passed to the query tag, would be labeled `tagdependent`.

Note that the value of the `body-content` element does not affect the interpretation of the body by the tag handler; the element is only intended to be used by an authoring tool for rendering the body content.

Tags That Define Scripting Variables

Tag Handlers

A tag handler is responsible for creating and setting the object referred to by the scripting variable into a context accessible from the page. It does this by using the `pageContext.setAttribute(name, value, scope)` or `pageContext.setAttribute(name, value)` methods. Typically an attribute passed to the custom tag specifies the name of the scripting variable object; this name can be retrieved by invoking the attribute's `get` method described in Using Scope Objects (page 578).

If the value of the scripting variable is dependent on an object present in the tag handler's context, it can retrieve the object using the `pageContext.getAttribute(name, scope)` method.

The usual procedure is that the tag handler retrieves a scripting variable, performs some processing on the object, and then sets the scripting variable's value using the `pageContext.setAttribute(name, object)` method.

The scope that an object can have is summarized in Table 16–4. The scope constrains the accessibility and lifetime of the object.

Table 16–4 Scope of Objects

Name	Accessible From	Lifetime
`page`	Current page	Until the response has been sent back to the user or the request is passed to a new page
`request`	Current page and any included or forwarded pages	Until the response has been sent back to the user
`session`	Current request and any subsequent request from the same browser (subject to session lifetime)	The life of the user's session
`application`	Current and any future request from the same Web application	The life of the application

Providing Information about the Scripting Variable

The example described in Tags That Define Scripting Variables (page 644) defines a scripting variable `book` that is used for accessing book information:

```
<bean:define id="book" name="bookDB" property="bookDetails"
  type="database.BookDetails"/>
<font color="red" size="+2">
  <%=messages.getString("CartRemoved")%>
  <strong><jsp:getProperty name="book"
       property="title"/></strong>
<br>&nbsp;<br>
</font>
```

When the JSP page containing this tag is translated, the Web container generates code to synchronize the scripting variable with the object referenced by the variable. To generate the code, the Web container requires certain information about the scripting variable:

- Variable name
- Variable class
- Whether the variable refers to a new or existing object
- The availability of the variable.

There are two ways to provide this information: by specifying the `variable` TLD subelement or by defining a tag extra info class and including the `tei-class` element in the TLD. Using the `variable` element is simpler, but slightly less flexible.

variable Element

The `variable` element has the following subelements:

- `name-given`—The variable name as a constant
- `name-from-attribute`—The name of an attribute whose translation-time value will give the name of the variable

One of `name-given` or `name-from-attribute` is required. The following subelements are optional:

- `variable-class`—The fully qualified name of the class of the variable. `java.lang.String` is the default.
- `declare`—Whether the variable refers to a new object. `True` is the default.
- `scope`—The scope of the scripting variable defined. `NESTED` is the default. Table 16–5 describes the availability of the scripting variable and the methods where the value of the variable must be set or reset.

Table 16–5 Scripting Variable Availability

Value	Availability	Methods
`NESTED`	Between the start tag and the end tag	In `doInitBody` and `doAfterBody` for a tag handler implementing `BodyTag`; otherwise, in `doStartTag`
`AT_BEGIN`	From the start tag until the end of the page	In `doInitBody`, `doAfterBody`, and `doEndTag` for a tag handler implementing `BodyTag`; otherwise, in `doStartTag` and `doEndTag`

Table 16–5 Scripting Variable Availability (Continued)

Value	Availability	Methods
`AT_END`	After the end tag until the end of the page	In `doEndTag`

The implementation of the Struts `bean:define` tag conforms to the JSP specification version 1.1, which requires you to define a tag extra info class. The JSP specification version 1.2 adds the `variable` element. You could define the following `variable` element for the `bean:define` tag:

```
<tag>
  <variable>
    <name-from-attribute>id</name-from-attribute>
    <variable-class>database.BookDetails</variable-class>
    <declare>true</declare>
    <scope>AT_BEGIN</scope>
  </variable>
</tag>
```

TagExtraInfo Class

You define a tag extra info class by extending the class `javax.servlet.jsp.TagExtraInfo`. A `TagExtraInfo` must implement the `getVariableInfo` method to return an array of `VariableInfo` objects containing the following information:

- Variable name
- Variable class
- Whether the variable refers to a new object
- The availability of the variable

The Web container passes a parameter called `data` to the `getVariableInfo` method that contains attribute-value tuples for each of the tag's attributes. These attributes can be used to provide the `VariableInfo` object with a scripting variable's name and class.

The Struts tag library provides information about the scripting variable created by the `bean:define` tag in the `DefineTei` tag extra info class. Since the name (`book`) and class (`database.BookDetails`) of the scripting variable are passed in as tag attributes, they can be retrieved with the `data.getAttributeString` method and used to fill in the VariableInfo constructor. To allow the scripting variable book to be used in the rest of the page, the scope of book is set to be AT_BEGIN.

```
public class DefineTei extends TagExtraInfo {
  public VariableInfo[] getVariableInfo(TagData data) {
  String type = data.getAttributeString("type");
    if (type == null)
      type = "java.lang.Object";
    return new VariableInfo[] {
      new VariableInfo(data.getAttributeString("id"),
        type,
        true,
        VariableInfo.AT_BEGIN)
    };
  }
}
```

The fully qualified name of the tag extra info class defined for a scripting variable must be declared in the TLD in the tei-class subelement of the tag element. Thus, the tei-class element for DefineTei would be as follows:

```
<tei-class>
  org.apache.struts.taglib.bean.DefineTagTei
</tei-class>
```

Cooperating Tags

Tags cooperate by sharing objects. JSP technology supports two styles of object sharing.

The first style requires that a shared object be named and stored in the page context (one of the implicit objects accessible to both JSP pages and tag handlers). To access objects created and named by another tag, a tag handler uses the pageContext.getAttribute(name, scope) method.

In the second style of object sharing, an object created by the enclosing tag handler of a group of nested tags is available to all inner tag handlers. This form of object sharing has the advantage that it uses a private namespace for the objects, thus reducing the potential for naming conflicts.

To access an object created by an enclosing tag, a tag handler must first obtain its enclosing tag with the static method TagSupport.findAncestorWithClass(from, class) or the TagSupport.getParent method. The former method should be used when a specific nesting of tag handlers cannot be guaranteed. Once the ancestor has been retrieved, a tag handler can access any statically or dynamically created objects. Statically created objects are members of the parent. Private objects can also be created dynamically. Such objects can be stored in a tag handler with the `setValue` method and retrieved with the `getValue` method.

The following example illustrates a tag handler that supports both the named and private object approaches to sharing objects. In the example, the handler for a query tag checks whether an attribute named `connection` has been set in the `doStartTag` method. If the `connection` attribute has been set, the handler retrieves the connection object from the page context. Otherwise, the tag handler first retrieves the tag handler for the enclosing tag, and then retrieves the connection object from that handler.

```
public class QueryTag extends BodyTagSupport {
  private String connectionId;
  public int doStartTag() throws JspException {
    String cid = getConnection();
    if (cid != null) {
    // there is a connection id, use it
      connection =(Connection)pageContext.
        getAttribute(cid);
    } else {
      ConnectionTag ancestorTag =
        (ConnectionTag)findAncestorWithClass(this,
          ConnectionTag.class);
      if (ancestorTag == null) {
        throw new JspTagException("A query without
          a connection attribute must be nested
          within a connection tag.");
      }
      connection = ancestorTag.getConnection();
    }
  }
}
```

The query tag implemented by this tag handler could be used in either of the following ways:

```
<tt:connection id="con01" ....> ... </tt:connection>
<tt:query id="balances" connection="con01">
  SELECT account, balance FROM acct_table
    where customer_number = <%= request.getCustno()%>
</tt:query>
```

```
<tt:connection ...>
  <x:query id="balances">
    SELECT account, balance FROM acct_table
      where customer_number = <%= request.getCustno()%>
  </x:query>
</tt:connection>
```

The TLD for the tag handler must indicate that the `connection` attribute is optional with the following declaration:

```
<tag>
  ...
  <attribute>
    <name>connection</name>
    <required>false</required>
  </attribute>
</tag>
```

Examples

The custom tags described in this section demonstrate solutions to two recurring problems in developing JSP applications: minimizing the amount of Java programming in JSP pages and ensuring a common look and feel across applications. In doing so, they illustrate many of the styles of tags discussed in the first part of the chapter.

An Iteration Tag

Constructing page content that is dependent on dynamically generated data often requires the use of flow control scripting statements. By moving the flow control logic to tag handlers, flow control tags reduce the amount of scripting needed in JSP pages.

The Struts `logic:iterate` tag retrieves objects from a collection stored in a JavaBeans component and assigns them to a scripting variable. The body of the tag retrieves information from the scripting variable. While elements remain in the collection, the `iterate` tag causes the body to be reevaluated.

JSP Page

Two Duke's Bookstore application pages, `catalog.jsp` and `showcart.jsp`, use the `logic:iterate` tag to iterate over collections of objects. An excerpt from `catalog.jsp` is shown below. The JSP page initializes the `iterate` tag with a collection (named by the `property` attribute) of the `bookDB` bean. The `iterate` tag sets the `book` scripting variable on each iteration over the collection. The `bookId` property of the `book` variable is exposed as another scripting variable. Properties of both variables are used to dynamically generate a table containing links to other pages and book catalog information.

```
<logic:iterate name="bookDB" property="books"
  id="book" type="database.BookDetails">
  <bean:define id="bookId" name="book" property="bookId"
    type="java.lang.String"/>

  <tr>
  <td bgcolor="#ffffaa">
  <a href="<%=request.getContextPath()%>
    /bookdetails?bookId=<%=bookId%>">
    <strong><jsp:getProperty name="book"
    property="title"/>&nbsp;</strong></a></td>

  <td bgcolor="#ffffaa" rowspan=2>
  <jsp:setProperty name="currency" property="amount"
    value="<%=book.getPrice()%>"/>
  <jsp:getProperty name="currency" property="format"/>
  &nbsp;</td>

  <td bgcolor="#ffffaa" rowspan=2>
  <a href="<%=request.getContextPath()%>
    /catalog?Add=<%=bookId%>">
    &nbsp;<%=messages.getString("CartAdd")%>
    &nbsp;</a></td></tr>

  <tr>
  <td bgcolor="#ffffff">
  &nbsp;&nbsp;<%=messages.getString("By")%> <em>
    <jsp:getProperty name="book"
      property="firstName"/>&nbsp;
    <jsp:getProperty name="book"
      property="surname"/></em></td></tr>
</logic:iterate>
```

Tag Handler

The implementation of the Struts logic:iterate tag conforms to the capabilities of the JSP version 1.1 specification, which requires you to extend the BodyTagSupport class. The JSP version 1.2 specification adds features (described in Tag Handler Does Not Interact with the Body, page 653) that simplify programming tags that iteratively evaluate their body. The following discussion is based on an implementation that uses these features.

The logic:iterate tag supports initializing the collection in several ways: from a collection provided as a tag attribute or from a collection that is a bean or a property of a bean. Our example uses the latter method. Most of the code in doStartTag is concerned with constructing an iterator over the collection object. The method first checks if the handler's collection property is set and, if not, proceeds to checking the bean and property attributes. If the name and property attributes are both set, doStartTag calls a utility method that uses JavaBeans introspection methods to retrieve the collection. Once the collection object is determined, the method constructs the iterator.

If the iterator contains more elements, doStartTag sets the value of the scripting variable to the next element and then indicates that the body should be evaluated; otherwise it ends the iteration by returning SKIP_BODY.

After the body has been evaluated, the doAfterBody method retrieves the body content and writes it to the out stream. The body content object is then cleared in preparation for another body evaluation. If the iterator contains more elements, doAfterBody again sets the value of the scripting variable to the next element and returns EVAL_BODY_AGAIN to indicate that the body should be evaluated again. This causes the reexecution of doAfterBody. When there are no remaining elements, doAfterBody terminates the process by returning SKIP_BODY.

```
public class IterateTag extends TagSupport {
  protected Iterator iterator = null;
  protected Object collection = null;
  protected String id = null;
  protected String name = null;
  protected String property = null;
  protected String type = null;
  public int doStartTag() throws JspException {
    Object collection = this.collection;
    if (collection == null) {
      try {
        Object bean = pageContext.findAttribute(name);
        if (bean == null) {
          ... throw an exception
```

```
         }
         if (property == null)
            collection = bean;
         else
            collection =
               PropertyUtils.
                  getProperty(bean, property);
         if (collection == null) {
            ... throw an exception
         }
      } catch
         ... catch exceptions thrown
            by PropertyUtils.getProperty
      }
   }
   // Construct an iterator for this collection
   if (collection instanceof Collection)
      iterator = ((Collection) collection).iterator();
   else if (collection instanceof Iterator)
      iterator = (Iterator) collection;
      ...
   }
   // Store the first value and evaluate,
   // or skip the body if none
   if (iterator.hasNext()) {
      Object element = iterator.next();
      pageContext.setAttribute(id, element);
      return (EVAL_BODY_AGAIN);
   } else
      return (SKIP_BODY);
}
public int doAfterBody() throws JspException {
   if (bodyContent != null) {
      try {
         JspWriter out = getPreviousOut();
         out.print(bodyContent.getString());
         bodyContent.clearBody();
      } catch (IOException e) {
         ...
      }
   }
   if (iterator.hasNext()) {
      Object element = iterator.next();
      pageContext.setAttribute(id, element);
      return (EVAL_BODY_AGAIN);
   } else
```

```
            return (SKIP_BODY);
        }
    }
}
```

Tag Extra Info Class

Information about the scripting variable is provided in the IterateTei tag extra info class. The name and class of the scripting variable are passed in as tag attributes and used to fill in the VariableInfo constructor.

```
public class IterateTei extends TagExtraInfo {
  public VariableInfo[] getVariableInfo(TagData data) {
  String type = data.getAttributeString("type");
  if (type == null)
     type = "java.lang.Object";

  return new VariableInfo[] {
     new VariableInfo(data.getAttributeString("id"),
        type,
        true,
        VariableInfo.AT_BEGIN)
     };
  }
}
```

A Template Tag Library

A template provides a way to separate the common elements that are part of each screen from the elements that change with each screen of an application. Putting all the common elements together into one file makes it easier to maintain and enforce a consistent look and feel in all the screens. It also makes development of individual screens easier because the designer can focus on portions of a screen that are specific to that screen while the template takes care of the common portions.

The template is a JSP page with placeholders for the parts that need to change with each screen. Each of these placeholders is referred to as a *parameter* of the template. For example, a simple template could include a title parameter for the top of the generated screen and a body parameter to refer to a JSP page for the custom content of the screen.

The template uses a set of nested tags—`definition`, `screen`, and `parameter`—to define a table of screen definitions and uses an `insert` tag to insert parameters from a screen definition into a specific application screen.

JSP Page

The template for the Duke’s Bookstore example, `template.jsp`, is shown below. This page includes a JSP page that creates the screen definition and then uses the `insert` tag to insert parameters from the definition into the application screen.

```
<%@ taglib uri="/tutorial-template.tld" prefix="tt" %>
<%@ page errorPage="errorpage.jsp" %>
<%@ include file="screendefinitions.jsp" %><html>
  <head>
     <title>
        <tt:insert definition="bookstore"
           parameter="title"/>
     </title>
  </head>
     <tt:insert definition="bookstore"
        parameter="banner"/>
     <tt:insert definition="bookstore"
        parameter="body"/>
  </body>
</html>
```

`screendefinitions.jsp` creates a screen definition based on a request attribute `selectedScreen`:

```
<tt:definition name="bookstore"
  screen="<%= (String)request.
     getAttribute(\"selectedScreen\") %>">
  <tt:screen id="/enter">
     <tt:parameter name="title"
        value="Duke’s Bookstore" direct="true"/>
     <tt:parameter name="banner"
        value="/banner.jsp" direct="false"/>
     <tt:parameter name="body"
        value="/bookstore.jsp" direct="false"/>
  </tt:screen>
  <tt:screen id="/catalog">
     <tt:parameter name="title"
```

```
      value="<%=messages.getString("TitleBookCatalog")%>"
      direct="true"/>
      ...
</tt:definition>
```

The template is instantiated by the `Dispatcher` servlet. `Dispatcher` first gets the requested screen and stores it as an attribute of the request. This is necessary because when the request is forwarded to `template.jsp`, the request URL doesn't contain the original request (for example, `/bookstore3/catalog`) but instead reflects the path (`/bookstore3/template.jsp`) of the forwarded page. Finally, the servlet dispatches the request to `template.jsp`:

```
public class Dispatcher extends HttpServlet {
  public void doGet(HttpServletRequest request,
       HttpServletResponse response) {
     request.setAttribute("selectedScreen",
       request.getServletPath());
     RequestDispatcher dispatcher =
       request.getRequestDispatcher("/template.jsp");
     if (dispatcher != null)
       dispatcher.forward(request, response);
  }
  public void doPost(HttpServletRequest request,
         HttpServletResponse response) {
     request.setAttribute("selectedScreen",
       request.getServletPath());
     RequestDispatcher dispatcher =
       request.getRequestDispatcher("/template.jsp");
     if (dispatcher != null)
       dispatcher.forward(request, response);
  }
}
```

Tag Handlers

The template tag library contains four tag handlers—`DefinitionTag`, `ScreenTag`, `ParameterTag`, and `InsertTag`—that demonstrate the use of cooperating tags. `DefinitionTag`, `ScreenTag`, and `ParameterTag` comprise a set of nested tag handlers that share public and private objects. `DefinitionTag` creates a public object named `definition` that is used by `InsertTag`.

In `doStartTag`, `DefinitionTag` creates a public object named `screens` that contains a hash table of screen definitions. A screen definition consists of a screen identifier and a set of parameters associated with the screen.

```
public int doStartTag() {
  HashMap screens = null;
  screens = (HashMap) pageContext.getAttribute("screens",
    pageContext.APPLICATION_SCOPE);
  if (screens == null)
    pageContext.setAttribute("screens", new HashMap(),
      pageContext.APPLICATION_SCOPE);
  return EVAL_BODY_INCLUDE;
}
```

The table of screen definitions is filled in by `ScreenTag` and `ParameterTag` from text provided as attributes to these tags. Table 16–6 shows the contents of the screen definitions hash table for the Duke's Bookstore application.

Table 16–6 Screen Definitions

Screen Id	Title	Banner	Body
`/enter`	`Duke's Bookstore`	`/banner.jsp`	`/bookstore.jsp`
`/catalog`	`Book Catalog`	`/banner.jsp`	`/catalog.jsp`
`/bookdetails`	`Book Description`	`/banner.jsp`	`/bookdetails.jsp`
`/showcart`	`Shopping Cart`	`/banner.jsp`	`/showcart.jsp`
`/cashier`	`Cashier`	`/banner.jsp`	`/cashier.jsp`
`/receipt`	`Receipt`	`/banner.jsp`	`/receipt.jsp`

In `doEndTag`, `DefinitionTag` creates a public object of class `Definition`, selects a screen definition from the `screens` object based on the URL passed in the request, and uses it to initialize the `Definition` object.

```
public int doEndTag()throws JspTagException {
  try {
    Definition definition = new Definition();
    HashMap screens = null;
    ArrayList params = null;
    TagSupport screen = null;
    screens = (HashMap)
      pageContext.getAttribute("screens",
        pageContext.APPLICATION_SCOPE);
    if (screens != null)
      params = (ArrayList) screens.get(screenId);
```

```
         else
            ...
         if (params == null)
            ...
         Iterator ir = null;
         if (params != null)
            ir = params.iterator();
         while ((ir != null) && ir.hasNext())
            definition.setParam((Parameter) ir.next());
            // put the definition in the page context
         pageContext.setAttribute(
            definitionName, definition);
      } catch (Exception ex) {
         ex.printStackTrace();
      }
      return EVAL_PAGE;
   }
```

If the URL passed in the request is `/enter`, the `Definition` contains the items from the first row of Table 16–6:

Title	**Banner**	**Body**
`Duke's Bookstore`	`/banner.jsp`	`/bookstore.jsp`

The definition for the URL `/enter` is shown in Table 16–7. The definition specifies that the value of the `Title` parameter, Duke's Bookstore, should be inserted directly into the output stream, but the values of `Banner` and `Body` should be dynamically included.

Table 16–7 Screen Definition for the URL `/enter`

Parameter Name	**Parameter Value**	**isDirect**
`title`	`Duke's Bookstore`	`true`
`banner`	`/banner.jsp`	`false`
`body`	`/bookstore.jsp`	`false`

InsertTag uses Definition to insert parameters of the screen definition into the response. In the doStartTag method, it retrieves the definition object from the page context.

```
public int doStartTag() {
  // get the definition from the page context
  definition = (Definition) pageContext.
    getAttribute(definitionName);
  // get the parameter
  if (parameterName != null && definition != null)
    parameter = (Parameter)definition.
      getParam(parameterName);
  if (parameter != null)
    directInclude = parameter.isDirect();
  return SKIP_BODY;
}
```

The doEndTag method inserts the parameter value. If the parameter is direct, it is directly inserted into the response; otherwise, the request is sent to the parameter, and the response is dynamically included into the overall response.

```
public int doEndTag()throws JspTagException {
  try {
    if (directInclude && parameter != null)
      pageContext.getOut().print(parameter.getValue());
    else {
      if ((parameter != null) &&
        (parameter.getValue() != null))
        pageContext.include(parameter.getValue());
    }
  } catch (Exception ex) {
    throw new JspTagException(ex.getMessage());
  }
  return EVAL_PAGE;
}
```

How Is a Tag Handler Invoked?

The Tag interface defines the basic protocol between a tag handler and a JSP page's servlet. It defines the life cycle and the methods to be invoked when the start and end tags are encountered.

The JSP page's servlet invokes the setPageContext, setParent, and attribute setting methods before calling doStartTag. The JSP page's servlet also guarantees that release will be invoked on the tag handler before the end of the page.

Here is a typical tag handler method invocation sequence:

```
ATag t = new ATag();
t.setPageContext(...);
t.setParent(...);
t.setAttribute1(value1);
t.setAttribute2(value2);
t.doStartTag();
t.doEndTag();
t.release();
```

The `BodyTag` interface extends `Tag` by defining additional methods that let a tag handler access its body. The interface provides three new methods:

- `setBodyContent`—Creates body content and adds to the tag handler
- `doInitBody`—Called before evaluation of the tag body
- `doAfterBody`—Called after evaluation of the tag body

A typical invocation sequence is:

```
t.doStartTag();
out = pageContext.pushBody();
t.setBodyContent(out);
// perform any initialization needed after body content is set
t.doInitBody();
t.doAfterBody();
// while doAfterBody returns EVAL_BODY_BUFFERED we
// iterate body evaluation
...
t.doAfterBody();
t.doEndTag();
t.pageContext.popBody();
t.release();
```

17 JavaServer Pages Standard Tag Library

Stephanie Bodoff

THE JavaServer Pages Standard Tag Library (JSTL) encapsulates core functionality common to many JSP applications. For example, instead of iterating over lists using a scriptlet or different iteration tags from numerous vendors, JSTL defines a standard tag that works the same everywhere. This standardization lets you learn a single tag and use it on multiple JSP containers. Also, when tags are standard, containers can optimize their implementation.

JSTL has support for common, structural tasks such as iteration and conditionals, tags for manipulating XML documents, internationalization tags, and tags for accessing databases using SQL. It also introduces the concept of an expression language to simplify page development. JSTL also provides a framework for integrating existing tag libraries with JSTL.

This chapter demonstrates the JSTL through excerpts from the JSP version of the Duke's Bookstore application discussed in previous chapters. It assumes that you are familiar with the material in the Using Tags (page 641) section of Chapter 16.

The Example JSP Pages

This chapter illustrates JSTL with excerpts from the JSP version of the Duke's Bookstore application discussed in Chapter 16 rewritten as follows:

- The Struts logic tags were replaced with JSTL core tags.
- The scriptlets accessing a message store were replaced with message formatting tags.
- The JavaBeans component database helper object was replaced with direct calls to the database via the JSTL SQL tags. For most applications, it is better to encapsulate calls to a database in a bean. JSTL includes SQL tags for situations where a new application is being prototyped and the overhead of creating a bean may not be warranted.

The source for the Duke's Bookstore application is located in the `<JWSDP_HOME>/docs/tutorial/examples/web/bookstore4` directory created when you unzip the tutorial bundle (see Running the Examples, page xiii).

To build, install, and run the example:

1. In a terminal window, go to `<JWSDP_HOME>/docs/tutorial/examples/web/bookstore4`.
2. Run `ant build`. The `build` target will spawn any necessary compilations and copy files to the `<JWSDP_HOME>/docs/tutorial/examples/web/bookstore4/build` directory.
3. Make sure Tomcat is started.
4. Run `ant install`. The `install` target notifies Tomcat that the new context is available.
5. Start the PointBase database server and populate the database if you have not done so already (see Accessing Databases from Web Applications, page 109).
6. Open the bookstore URL `http://localhost:8080/bookstore4/enter`.

See Common Problems and Their Solutions (page 87) and Troubleshooting (page 574) for help with diagnosing common problems.

Using JSTL

JSTL includes a wide variety of tags that naturally fit into discrete functional areas. Therefore, JSTL is exposed via multiple tag library descriptors (TLDs) to clearly show the functional areas it covers and give each area its own namespace. Table 17–1 summarizes these functional areas along with the logical TLD names and prefixes used in this chapter and Duke's Bookstore application.

Table 17–1 JSTL Tags

Area	Function	Tags	TLD	Prefix
Core	Expression Language Support	catch out remove set	/jstl-c	c

	Flow Control	choose
		  when
		  otherwise
		forEach
		forTokens
		if		

	URL Management	import
		  param
		redirect
		  param
		url
		  param		

XML	Core	out parse set	/jstl-x	x

	Flow Control	choose
		  when
		  otherwise
		forEach
		if		

	Transformation	transform param		

Table 17–1 JSTL Tags (Continued)

Area	Function	Tags	TLD	Prefix
I18n	Locale	`setLocale`	`/jstl-fmt`	`fmt`

	Message formatting	`bundle`
		`message`
		  `param`
		`setBundle`		

	Number and date formatting	`formatNumber`
		`formatDate`
		`parseDate`
		`parseNumber`
		`setTimeZone`
		`timeZone`		

Data-base		`setDataSource`	`/jstl-sql`	`sql`

	SQL	`query`
		  `dateParam`
		  `param`
		`transaction`
		`update`
		  `dateParam`
		  `param`		

For example, to use the JSTL core tags in a JSP page, you declare the library using a `taglib` directive that references the TLD:

```
<%@ taglib uri="/jstl-core" prefix="c" %>
```

The JSTL tag libraries comes in two versions (see Twin Libraries, page 678). The TLDs for the JSTL-EL library are named *`prefix`*`.tld`. The TLDs for the JSTL-RT library are named *`prefix`*`-rt.tld`. Since the examples discussed in this chapter use logical TLD names, we map the logical names to actual TLD locations with `taglib` elements in the Web application deployment descriptor. Here is the entry that maps the core library logical TLD name `/jstl-c` to its location `/WEB-INF/c.tld`:

```
<taglib>
  <taglib-uri>/jstl-c</taglib-uri>
  <taglib-location>/WEB-INF/c.tld</taglib-location>
</taglib>
```

In the Java WSDP, the JSTL TLDs are stored in `<JWSDP_HOME>/jstl-1.0.3/tld`. When you build the Duke's Bookstore application these TLDs are automatically copied into `<JWSDP_HOME>/docs/tutorial/examples/web/bookstore4/build/WEB-INF`.

You can also reference a TLD in a `taglib` directive with an absolute URI:

- Core: `http://java.sun.com/jstl/core`
- XML: `http://java.sun.com/jstl/xml`
- Internationalization: `http://java.sun.com/jstl/fmt`
- SQL: `http://java.sun.com/jstl/sql`

When you use an absolute URI, you do not have to add the `taglib` element to `web.xml`; the JSP container automatically locates the TLD inside the JSTL library implementation.

In addition to declaring the tag library, you also need to make the JSTL API and implementation available to the Web application. These are distributed as the archives `jstl.jar` in `<JWSDP_HOME>/jstl-1.0.3` and `standard.jar` in `<JWSDP_HOME>/jstl-1.0.3/standard`. When you build the Duke's Bookstore application these libraries are automatically copied into `<JWSDP_HOME>/docs/tutorial/examples/web/bookstore4/build/WEB-INF/lib`.

Expression Language Support

A primary feature of JSTL is its support for an expression language (EL). An expression language, in concert with JSTL tags, makes it possible to easily access application data and manipulate it in simple ways without having to use scriptlets or request-time expressions. Currently, a page author has to use an expression `<%= aName %>` to access the value of a system or user-defined JavaBeans component. For example:

```
<x:aTag att="<%= pageContext.getAttribute("aName") %>">
```

Referring to nested bean properties is even more complex:

```
<%= aName.getFoo().getBar() %>
```

This makes page authoring more complicated than it need be.

An expression language allows a page author to access an object using a simplified syntax such as

```
<x:atag att="${aName}">
```

for a simple variable or

```
<x:aTag att="${aName.foo.bar}">
```

for a nested property.

The JSTL expression language promotes JSP *scoped attributes* as the standard way to communicate information from business logic to JSP pages. For example, the `test` attribute of the this conditional tag is supplied with an expression that compares the number of items in the session-scoped attribute named `cart` with 0:

```
<c:if test="${sessionScope.cart.numberOfItems > 0}">
  ...
</c:if>
```

The next version of the JSP specification will standardize on an expression language for all custom tag libraries. This release of JSTL includes a snapshot of that expression language.

Twin Libraries

The JSTL tag libraries come in two versions which differ only in the way they support the use of runtime expressions for attribute values.

In the JSTL-RT tag library, expressions are specified in the page's scripting language. This is exactly how things currently work in current tag libraries.

In the JSTL-EL tag library, expressions are specified in the JSTL expression language. An expression is a `String` literal in the syntax of the EL.

When using the EL tag library you cannot pass a scripting language expression for the value of an attribute. This rule makes it possible to validate the syntax of an expression at translation time.

JSTL Expression Language

The JSTL expression language is responsible for handling both expressions and literals. Expressions are enclosed by the `${ }` characters. For example:

```
<c:if test="${bean1.a < 3}" />
```

Any value that does not begin with `${` is treated as a literal that is parsed to the expected type using the `PropertyEditor` for the expected type:

```
<c:if test="true" />
```

Literal values that contain the `${` characters must be escaped as follows:

```
<mytags:example attr1="an expression is ${'${'}true}" />
```

Attributes

Attributes are accessed by name, with an optional scope. Properties of attributes are accessed using the `.` operator, and may be nested arbitrarily.

The EL unifies the treatment of the `.` and `[ ]` operators. Thus, `expr-a.expr-b` is equivalent to `expr-a[expr-b]`. To evaluate `expr-a[expr-b]`, evaluate `expr-a` into `value-a` and evaluate `expr-b` into `value-b`.

- If `value-a` is a `Map` return `value-a.get(value-b)`.
- If `value-a` is a `List` or array coerce `value-b` to `int` and return `value-a.get(value-b)` or `Array.get(value-a, value-b)`, as appropriate.
- If `value-a` is a JavaBeans object, coerce `value-b` to `String`. If `value-b` is a readable property of `value-a` the return result of getter call.

The EL evaluates an identifier by looking up its value as an attribute, according to the behavior of `PageContext.findAttribute(String)`. For example, `${product}` will look for the attribute named `product`, searching the page, request, session, and application scopes and will return its value. If the attribute is not found, `null` is returned. Note that an identifier that matches one of the implicit objects described in the next section will return that implicit object instead of an attribute value.

Implicit Objects

The JSTL expression language defines a set of implicit objects:

- `pageContext` - the `PageContext` object
- `pageScope` - a `Map` that maps page-scoped attribute names to their values
- `requestScope` - a `Map` that maps request-scoped attribute names to their values
- `sessionScope` - a `Map` that maps session-scoped attribute names to their values
- `applicationScope` - a `Map` that maps application-scoped attribute names to their values
- `param` - a `Map` that maps parameter names to a single `String` parameter value (obtained by calling `ServletRequest.getParameter(String)`)
- `paramValues` - a `Map` that maps parameter names to a `String[ ]` of all values for that parameter (obtained by calling `ServletRequest.getParameterValues(String)`)
- `header` - a `Map` that maps header names to a single `String` header value (obtained by calling `ServletRequest.getheader(String)`)
- `headerValues` - a `Map` that maps header names to a `String[ ]` of all values for that parameter (obtained by calling `ServletRequest.getHeaders(String)`)
- `cookie` - a `Map` that maps cookie names to a single `Cookie` (obtained by calling `HttpServletRequest.getCookie(String)`)
- `initParam` - a `Map` that maps a parameter names to a single `String` parameter value (obtained by calling `ServletRequest.getInitParameter(String)`)

When an expression references one of these objects by name, the appropriate object is returned instead of the corresponding attribute. For example: `${pageContext}` returns the `PageContext` object, even if there is an existing `pageCon-`

`text` attribute containing some other value. Table 17–2 shows some examples of using these implicit objects.

Table 17–2 Example JSTL Expressions

Expression	**Result**
`${pageContext.request.contextPath}`	The context path (obtained from `HttpServletRequest`)
`${sessionScope.cart.numberOfItems}`	The `numberOfItems` property of the session-scoped attribute named `cart`
`${param["mycom.productId"]}`	The `String` value of the `mycom.productId` parameter

Literals

- Boolean: `true` and `false`
- Long: as in Java
- Floating point: as in Java
- String: with single and double quotes. `"` is escaped as `\"`, `'` is escaped as `\'`, and `\` is escaped as `\\`.
- Null: `null`

Operators

The EL provides the following operators:

- Arithmetic: `+`, `-`, `*`, `/` and `div`, `%` and `mod`, `-`
- Logical: `and`, `&&`, `or`, `||`, `not`, `!`
- Relational: `==`, `eq`, `!=`, `ne`, `<`, `lt`, `>`, `gt`, `<=`, `ge`, `>=`, `le`. Comparisons may be made against other values, or against boolean, string, integer, or floating point literals.
- Empty: The `empty` operator is a prefix operation that can be used to determine if a value is `null` or empty.

Consult the JSTL 1.0 Specification for the precedence and effects of these operators.

Tag Collaboration

Tags usually collaborate with their environment in implicit and explicit ways. Implicit collaboration is done via a well defined interface that allows nested tags to work seamlessly with the ancestor tag exposing that interface. The JSTL iterator tags support this mode of collaboration.

Explicit collaboration happens when a tag exposes information to its environment. Traditionally, this has been done by exposing a scripting variable (with a JSP scoped attribute providing the actual object). Because JSTL has an expression language, there is less need for scripting variables. So the JSTL tags (both the EL and RT versions) expose information only as JSP scoped attributes; no scripting variables are used. The convention JSTL follows is to use the name `var` for any tag attribute that exports information about the tag. For example, the `forEach` tag exposes the current item of the shopping cart it is iterating over in the following way:

```
<c:forEach var="item" items="${sessionScope.cart.items}">
  ...
</c:forEach>
```

The name `var` was selected to highlight the fact that the scoped variable exposed is not a scripting variable (which is normally the case for attributes named `id`).

In situations where a tag exposes more than one piece of information, the name `var` is used for the primary piece of information being exported, and an appropriate name is selected for any other secondary piece of information exposed. For example, iteration status information is exported by the `forEach` tag via the attribute `status`.

Core Tags

The core tags include those related to expressions, flow control, and a generic way to access URL-based resources whose content can then be included or processed within the JSP page.

Table 17–3 Core Tags

Area	Function	Tags	TLD	Prefix

Core	Expression Language Support	`catch`
		`out`
		`remove`
		`set`
			`/jstl-c`	`c`

	Flow Control	`choose`
		  `when`
		  `otherwise`
		`forEach`
		`forTokens`
		`if`		

	URL Management	`import`
		  `param`
		`redirect`
		  `param`
		`url`
		  `param`		

Expression Tags

The `out` tag evaluates an expression and outputs the result of the evaluation to the current `JspWriter` object. It is the equivalent of the JSP syntax `<%=` *`expression`* `%>`. For example, `showcart.jsp` displays the number of items in a shopping cart as follows:

```
<c:out value="${sessionScope.cart.numberOfItems}"/>
```

The `set` tag sets the value of an attribute in any of the JSP scopes (page, request, session, application). If the attribute does not already exist, it is created.

The JSP scoped attribute can be set either from attribute value:

```
<c:set var="foo" scope="session" value="..."/>
```

or from the body of the tag:

```
<c:set var="foo">
  ...
</c:set>
```

For example, the following sets a page-scoped attribute named `bookID` with the value of the request parameter named `Remove`:

```
<c:set var="bookId" value="${param.Remove}"/>
```

If you were using the RT version of the library, the statement would be:

```
<c_rt:set var="bookId"
  value="<%= request.getParameter("Remove") %>" />
```

To remove a scoped attribute, you use the `remove` tag. When the bookstore JSP page `receipt.jsp` is invoked, the shopping session is finished, so the `cart` session attribute is removed as follows:

```
<c:remove var="cart" scope="session"/>
```

The JSTL expression language reduces the need for scripting. However, page authors will still have to deal with situations where some attributes of non-JSTL tags must be specified as expressions in the page's scripting language. The standard JSP element `jsp:useBean` is used to declare a scripting variable that can be used in a scripting language expression or scriptlet. For example, `showcart.jsp` removes a book from a shopping cart using a scriptlet. The ID of the book to be removed is passed as a request parameter. The value of the request parameter is first set as a page attribute (to be used later by the JSTL `sql:query` tag) and then declared as scripting variable and passed to the `cart.remove` method:

```
<c:set var="bookId" value="${param.Remove}"/>
<jsp:useBean id="bookId" type="java.lang.String" />
<% cart.remove(bookId); %>
<sql:query var="books"
  dataSource="${applicationScope.bookDS}">
  select * from PUBLIC.books where id = ?
  <sql:param value="${bookId}" />
</sql:query>
```

The `catch` tag provides a complement to the JSP error page mechanism. It allows page authors to recover gracefully from error conditions that they can control. Actions that are of central importance to a page should *not* be encapsulated in a `catch`, so their exceptions will propagate to an error page. Actions with secondary importance to the page should be wrapped in a `catch`, so they never cause the error page mechanism to be invoked.

The exception thrown is stored in the scoped variable identified by `var`, which always has page scope. If no exception occurred, the scoped variable identified by `var` is removed if it existed. If `var` is missing, the exception is simply caught and not saved.

Flow Control Tags

To execute flow control logic, a page author must generally resort to using scriptlets. For example, the following scriptlet is used to iterate through a shopping cart:

```
<%
  Iterator i = cart.getItems().iterator();
  while (i.hasNext()) {
     ShoppingCartItem item =
        (ShoppingCartItem)i.next();
     ...
%>
     <tr>
     <td align="right" bgcolor="#ffffff">
     <%=item.getQuantity()%>
     </td>
     ...
<%
  }
%>
```

Flow control tags eliminate the need for scriptlets. The next two sections have examples that demonstrate the conditional and iterator tags.

Conditional Tags

The `if` tag allows the conditional execution of its body according to value of a test attribute. The following example from `catalog.jsp` tests whether the request parameter `Add` is empty. If the test evaluates to `true`, the page queries the database for the book record identified by the request parameter and adds the book to the shopping cart:

```
<c:if test="${!empty param.Add}">
  <c:set var="bid" value="${param.Add}"/>
  <jsp:useBean id="bid"  type="java.lang.String" />
   <sql:query var="books"
     dataSource="${applicationScope.bookDS}">
     select * from PUBLIC.books where id = ?
     <sql:param value="${bid}" />
  </sql:query>
  <c:forEach var="bookRow" begin="0" items="${books.rows}">
     <jsp:useBean id="bookRow" type="java.util.Map" />
     <jsp:useBean id="addedBook"
        class="database.BookDetails" scope="page" />
  ...
  <% cart.add(bid, addedBook); %>
...
</c:if>
```

The `choose` tag performs conditional block execution by the embedded `when` sub tags. It renders the body of the first `when` tag whose test condition evaluates to true. If none of the test conditions of nested `when` tags evaluate to `true`, then the body of an `otherwise` tag is evaluated, if present.

For example, the following sample code shows how to render text based on a customer's membership category.

```
<c:choose>
  <c:when test="${customer.category == 'trial'}" >
     ...
  </c:when>
  <c:when test="${customer.category == 'member'}" >
     ...
  </c:when>
     <c:when test="${customer.category == 'preferred'}" >
     ...
  </c:when>
  <c:otherwise>
     ...
  </c:otherwise>
</c:choose>
```

The `choose`, `when`, and `otherwise` tags can be used to construct an `if-then-else` statement as follows:

```
<c:choose>
  <c:when test="${count == 0} >
    No records matched your selection.
  </c:when>
  <c:otherwise>
    <c:out value="${count}"/> records matched your selection.
  </c:otherwise>
</c:choose>
```

Iterator Tags

The `forEach` tag allows you to iterate over a collection of objects. You specify the collection via the `items` attribute, and the current item is available through a scope variable named by the `item` attribute.

A large number of collection types are supported by `forEach`, including all implementations of `java.util.Collection` and `java.util.Map`. If the `items` attribute is of type `java.util.Map`, then the current item will be of type `java.util.Map.Entry`, which has the following properties:

- `key` - the key under which the item is stored in the underlying `Map`
- `value` - the value that corresponds to the key

Arrays of objects as well as arrays of primitive types (for example, `int`) are also supported. For arrays of primitive types, the current item for the iteration is automatically wrapped with its standard wrapper class (for example, `Integer` for `int`, `Float` for `float`, and so on).

Implementations of `java.util.Iterator` and `java.util.Enumeration` are supported but these must be used with caution. `Iterator` and `Enumeration` objects are not resettable so they should not be used within more than one iteration tag. Finally, `java.lang.String` objects can be iterated over if the string contains a list of comma separated values (for example: Monday,Tuesday,Wednesday,Thursday,Friday).

Here's the shopping cart iteration from the previous section with the `forEach` tag:

```
<c:forEach var="item" items="${sessionScope.cart.items}">
  ...
  <tr>
    <td align="right" bgcolor="#ffffff">
```

```
      <c:out value="${item.quantity}"/>
    </td>
    ...
  </c:forEach>
```

The forTokens tag is used to iterate over a collection of tokens separated by a delimiter.

URL Tags

The jsp:include element provides for the inclusion of static and dynamic resources in the same context as the current page. However, jsp:include cannot access resources that reside outside of the Web application and causes unnecessary buffering when the resource included is used by another element.

In the example below, the transform element uses the content of the included resource as the input of its transformation. The jsp:include element reads the content of the response, writes it to the body content of the enclosing transform element, which then re-reads the exact same content. It would be more efficient if the transform element could access the input source directly and avoid the buffering involved in the body content of the transform tag.

```
<acme:transform>
  <jsp:include page="/exec/employeesList"/>
<acme:transform/>
```

The import tag is therefore the simple, generic way to access URL-based resources whose content can then be included and or processed within the JSP page. For example, in XML Tags (page 689), import is used to read in the XML document containing book information and assign the content to the scoped variable xml:

```
<c:import url="/books.xml" var="xml" />
<x:parse xml="${xml}" var="booklist"
  scope="application" />
```

The param tag, analogous to the jsp:param tag (see jsp:param Element, page 624), can be used with import to specify request parameters.

In Session Tracking (page 601) we discussed how an application must rewrite URLs to enable session tracking whenever the client turns off cookies. You can use the url tag to rewrite URLs returned from a JSP page. The tag includes the session ID in the URL only if cookies are disabled; otherwise, it returns the URL unchanged. Note that this feature requires the URL to be *relative*. The `url` tag takes `param` subtags for including parameters in the returned URL. For example, `catalog.jsp` rewrites the URL used to add a book to the shopping cart as follows:

```
<c:url var="url"
  value="/catalog" >
  <c:param name="Add" value="${bookId}" />
</c:url>
<p><strong><a href="<c:out value='${url}'/>">
```

The `redirect` tag sends an HTTP redirect to the client. The `redirect` tag takes `param` subtags for including parameters in the returned URL.

XML Tags

A key aspect of dealing with XML documents is to be able to easily access their content. XPath, a W3C recommendation since 1999, provides an easy notation for specifying and selecting parts of an XML document. The JSTL XML tag set, listed in Table 17–4, is based on XPath (see How XPath Works, page 294).

Table 17–4 XML Tags

Area	Function	Tags	TLD	Prefix

XML	Core	`out`
		`parse`
		`set`
			`/jstl-x`	`x`

	Flow Control	`choose`
		  `when`
		  `otherwise`
		`forEach`
		`if`		

	Transformation	`transform`
		  `param`		

The XML tags use XPath as a *local* expression language; XPath expressions are always specified using attribute `select`. This means that only values specified for `select` attributes are evaluated using the XPath expression language. All other attributes are evaluated using the rules associated with the global expression language.

In addition to the standard XPath syntax, the JSTL XPath engine supports the following scopes to access Web application data within an XPath expression:

- `$foo`
- `$param:`
- `$header:`
- `$cookie:`
- `$initParam:`
- `$pageScope:`
- `$requestScope:`
- `$sessionScope:`
- `$applicationScope:`

These scopes are defined in exactly the same way as their counterparts in the JSTL expression language discussed in Implicit Objects (page 680). Table 17–5 shows some examples of using the scopes.

Table 17–5 Example XPath Expressions

XPath Expression	**Result**
`$sessionScope:profile`	The session-scoped attribute named `profile`
`$initParam:mycom.productId`	The `String` value of the `mycom.productId` context parameter

The XML tags are illustrated in another version (`bookstore5`) of the Duke's Bookstore application. This version replaces the database with an XML representation (`books.xml`) of the bookstore database. To build and install this version of the application, follow the directions in The Example JSP Pages (page 674) replacing `bookstore4` with `bookstore5`.

Since the XML tags require an XPath evaluator, the Java WSDP includes the Jaxen XPath evaluator in two libraries, `jaxen-full.jar` and `saxpath.jar`, in `<`*`JWSDP_HOME`*`>/jstl-1.0.3/standard`. When you build the Duke's Bookstore application these libraries are automatically copied into `<`*`JWSDP_HOME`*`>/docs/tutorial/examples/web/bookstore5/build/WEB-INF/lib`.

Core Tags

The core XML tags provide basic functionality to easily parse and access XML data.

The `parse` tag parses an XML document and saves the resulting object in the scoped attribute specified by attribute `var`. In `bookstore5`, the XML document is parsed and saved to a context attribute in `parseBooks.jsp`, which is included by all JSP pages that need access to the document:

```
<c:if test="${applicationScope:booklist == null}" >
  <c:import url="/books.xml" var="xml" />
  <x:parse xml="${xml}" var="booklist" scope="application" />
</c:if>
```

The `out` and `set` tags parallel the behavior described in Expression Tags (page 683) for the XPath local expression language. The `out` tag evaluates an XPath expression on the current context node and outputs the result of the evaluation to the current `JspWriter` object.

The `set` tag evaluates an XPath expression and sets the result into a JSP scoped attribute specified by attribute `var`.

The JSP page `bookdetails.jsp` selects a book element whose `id` attribute matches the request parameter `bookId` and sets the `abook` attribute. The `out` tag then selects the book's title element and outputs the result.

```
<x:set var="abook"
select="$applicationScope.booklist/
         books/book[@id=$param:bookId]" />
  <h2><x:out select="$abook/title"/></h2>
```

As you have just seen, `x:set` stores an internal XML representation of a *node* retrieved using an XPath expression; it doesn't convert the selected node into a `String` and store it. Thus, `x:set` is primarily useful for storing parts of documents for later retrieval.

If you want to store a `String`, you need to use `x:out` within `c:set`. The `x:out` tag converts the node to a `String`, and `c:set>` then stores the `String` as a scoped attribute. For example, bookdetails.jsp stores a scoped attribute containing a book price, which is later fed into a fmt tag, as follows:

```
<c:set var="price">
  <x:out select="$abook/price"/>
</c:set>
<h4><fmt:message key="ItemPrice"/>:
  <fmt:formatNumber value="${price}" type="currency"/>
```

The other option, which is more direct but requires that the user have more knowledge of XPath, is to coerce the node to a String manually using XPath's string function.

```
<x:set var="price" select="string($abook/price)"/>
```

Flow Control Tags

The XML flow control tags parallel the behavior described in Flow Control Tags (page 685) for the XPath expression language.

The JSP page catalog.jsp uses the forEach tag to display all the books contained in booklist as follows:

```
<x:forEach var="book"
  select="$applicationScope:booklist/books/*">
  <tr>
    <c:set var="bookId">
      <x:out select="$book/@id"/>
    </c:set>=
    <td bgcolor="#ffffaa">
      <c:url var="url"
      value="/bookdetails" >
        <c:param name="bookId" value="${bookId}" />
        <c:param name="Clear" value="0" />
      </c:url>
      <a href="<c:out value='${url}'/>">
      <strong><x:out select="$book/title"/>&nbsp;
      </strong></a></td>
    <td bgcolor="#ffffaa" rowspan=2>
      <c:set var="price">
        <x:out select="$book/price"/>
      </c:set>
      <fmt:formatNumber value="${price}" type="currency"/>
      &nbsp;
    </td>
```

```
      <td bgcolor="#ffffaa" rowspan=2>
      <c:url var="url" value="/catalog" >
        <c:param name="Add" value="${bookId}" />
      </c:url>
      <p><strong><a href="<c:out value='${url}'/>">&nbsp;
        <fmt:message key="CartAdd"/>&nbsp;</a>
      </td>
    </tr>
    <tr>
      <td bgcolor="#ffffff">
      &nbsp;&nbsp;<fmt:message key="By"/> <em>
        <x:out select="$book/firstname"/>&nbsp;
        <x:out select="$book/surname"/></em></td></tr>
</x:forEach>
```

Transformation Tags

The `transform` tag applies a transformation, specified by a XSLT stylesheet set by the attribute `xslt`, to an XML document, specified by the attribute `xml`. If the `xml` attribute is not specified, the input XML document is read from the tag's body content.

The `param` subtag can be used along with `transform` to set transformation parameters. The attributes `name` and `value` are used to specify the parameter. The value attribute is optional. If it is not specified the value is retrieved from the tag's body.

Internationalization Tags

In Internationalizing and Localizing Web Applications (page 108) we discussed the how to adapt Web applications to the language and formatting conventions of client locales. This section describes tags that support the internationalization of JSP pages.

JSTL defines tags for:

- Setting the locale for a page
- Creating locale-sensitive messages

- Formatting and parsing data elements such as numbers, currencies, dates, and times in a locale-sensitive or customized manner

Table 17–6 Internationalization Tags

Area	Function	Tags	TLD	Prefix
I18n	Setting Locale	`setLocale` `requestEncoding`	`/jstl-fmt`	`fmt`

	Messaging	`bundle`
		`message`
		  `param`
		`setBundle`		

	Number and Date Formatting	`formatNumber`
		`formatDate`
		`parseDate`
		`parseNumber`
		`setTimeZone`
		`timeZone`		

Setting the Locale

The `setLocale` tag is used to override the client-specified locale for a page. The `requestEncoding` tag is used to set the request's character encoding, in order to be able to correctly decode request parameter values whose encoding is different from ISO-8859-1.

Messaging Tags

By default, browser-sensing capabilities for locales are enabled. This means that the client determines (via its browser settings) which locale to use, and allows page authors to cater to the language preferences of their clients.

bundle Tag

You use the `bundle` tag to specify a resource bundle for a page.

To define a resource bundle for a Web application you specify the context parameter `javax.servlet.jsp.jstl.fmt.localizationContext` in the Web application deployment descriptor. Here is the declaration from the Duke's Bookstore descriptor:

```
<context-param>
  <param-name>
    javax.servlet.jsp.jstl.fmt.localizationContext
  </param-name>
  <param-value>messages.BookstoreMessages</param-value>
</context-param>
```

message Tag

The `message` tag is used to output localized strings. The following tag from `catalog.jsp`

```
<h3><fmt:message key="Choose"/></h3>
```

is used to output a string inviting customers to choose a book from the catalog.

The `param` subtag provides a single argument (for parametric replacement) to the compound message or pattern in its parent `message` tag. One `param` tag must be specified for each variable in the compound message or pattern. Parametric replacement takes place in the order of the `param` tags.

Formatting Tags

JSTL provides a set of tags for parsing and formatting locale-sensitive numbers and dates.

The `formatNumber` tag is used to output localized numbers. The following tag from `showcart.jsp`

```
<fmt:formatNumber value="${book.price}" type="currency"/>
```

is used to display a localized price for a book. Note that since the price is maintained in the database in dollars, the localization is somewhat simplistic, because the `formatNumber` tag is unaware of exchange rates. The tag formats currencies but does not convert them.

Analogous tags for formatting dates (`formatDate`), and parsing numbers and dates (`parseNumber`, `parseDate`) are also available. The `timeZone` tag establishes the time zone (specified via the `value` attribute) to be used by any nested `formatDate` tags.

In `receipt.jsp`, a "pretend" ship date is created and then formatted with the `formatDate` tag:

```
<jsp:useBean id="now" class="java.util.Date" />
<jsp:setProperty name="now" property="time"
  value="<%= now.getTime() + 432000000 %>" />
<fmt:message key="ShipDate"/>
<fmt:formatDate value="${now}" type="date"
  dateStyle="full"/>.
```

SQL Tags

The JSTL SQL tags are designed for quick prototyping and simple applications. For production applications, database operations are normally encapsulated in JavaBeans components.

Table 17–7 SQL Tags

Area	Function	Tags	TLD	Prefix
Database		`setDataSource`	`/jstl-sql`	`sql`

	SQL	`query`
		  `dateParam`
		  `param`
		`transaction`
		`update`
		  `dateParam`
		  `param`		

The `setDataSource` tag is provided to allow you to set data source information for the database. You can provide a JNDI name or `DriverManager` parameters to set the data source information. All the Duke's Bookstore pages that have more than one SQL tag use the following statement to set the data source:

```
<sql:setDataSource dataSource="jdbc/BookDB" />
```

The query tag is used to perform an SQL query that returns a result set. For parameterized SQL queries, you use a nested param tag inside the query tag.

In catalog.jsp, the value of the Add request parameter determines which book information should be retrieved from the database. This parameter is saved as the attribute name bid and passed to the param tag. Notice that the query tag obtains its data source from the context attribute bookDS set in the context listener.

```
<c:set var="bid" value="${param.Add}"/>
<sql:query var="books" >
  select * from PUBLIC.books where id = ?
  <sql:param value="${bid}" />
</sql:query>
```

The update tag is used to update a database row. The transaction tag is used to perform a series of SQL statements atomically.

The JSP page receipt.jsp page uses both tags to update the database inventory for each purchase. Since a shopping cart can contain more than one book, the transaction tag is used to wrap multiple queries and updates. First the page establishes that there is sufficient inventory, then the updates are performed.

```
<c:set var="sufficientInventory" value="true" />
<sql:transaction>
  <c:forEach var="item" items="${sessionScope.cart.items}">
     <c:set var="book" value="${item.item}" />
     <c:set var="bookId" value="${book.bookId}" />

     <sql:query var="books"
        sql="select * from PUBLIC.books where id = ?" >
        <sql:param value="${bookId}" />
     </sql:query>
     <jsp:useBean id="inventory"
        class="database.BookInventory" />
     <c:forEach var="bookRow" begin="0"
        items="${books.rowsByIndex}">
        <jsp:useBean id="bookRow"  type="java.lang.Object[]" />
        <jsp:setProperty name="inventory" property="quantity"
           value="<%=(Integer)bookRow[7]%>" />

        <c:if test="${item.quantity > inventory.quantity}">
           <c:set var="sufficientInventory" value="false" />
           <h3><font color="red" size="+2">
           <fmt:message key="OrderError"/>
           There is insufficient inventory for
           <i><c:out value="${bookRow[3]}"/></i>.</font></h3>
        </c:if>
        </c:forEach>
      </c:forEach>

      <c:if test="${sufficientInventory == 'true'}" />
        <c:forEach var="item" items="${sessionScope.cart.items}">
          <c:set var="book" value="${item.item}" />
          <c:set var="bookId" value="${book.bookId}" />

          <sql:query var="books"
            sql="select * from PUBLIC.books where id = ?" >
            <sql:param value="${bookId}" />
          </sql:query>

          <c:forEach var="bookRow" begin="0"
            items="${books.rows}">
            <sql:update var="books" sql="update PUBLIC.books set
              inventory = inventory - ? where id = ?" >
              <sql:param value="${item.quantity}" />
              <sql:param value="${bookId}" />
            </sql:update>
          </c:forEach>
        </c:forEach>
        <h3><fmt:message key="ThankYou"/>
          <c:out value="${param.cardname}" />.</h3><br>
      </c:if>
    </sql:transaction>
```

query Tag Result Interface

The `Result` interface is used to retrieve information from objects returned from a `query` tag.

```
public interface Result
  public String[] getColumnNames();
  public int getRowCount()
  public Map[] getRows();
  public Object[][] getRowsByIndex();
  public boolean isLimitedByMaxRows();
```

For complete information about this interface, see the API documentation for the `javax.servlet.jsp.jstl.sql` package.

The `var` attribute set by a `query` tag is of type `Result`. The `getRows` method returns an array of maps that can be supplied to the `items` attribute of a `forEach` tag. The JSTL expression language converts the syntax `${`*`result`*`.rows}` to a call to `result.getRows`. The expression `${books.rows}` in the following example returns an array of maps.

When you provide a array of maps to the `forEach` tag, the `var` attribute set by the tag is of type `Map`. To retrieve information from a row, use the `get("colname")` method to get a column value. The JSTL expression language converts the syntax `${map.colname}` to a call to `map.get("colname")`. For example, the expression `${book.title}` returns the value of the title entry of a book map.

The Duke's Bookstore page `bookdetails.jsp` retrieves the column values from the `book` map as follows.

```
<c:forEach var="book" begin="0" items="${books.rows}">
  <h2><c:out value="${book.title}"/></h2>
  &nbsp;<fmt:message key="By"/> <em><c:out
  value="${book.firstname}"/> <c:out
  value="${book.surname}"/></em>&nbsp;&nbsp;
  (<c:out value="${book.year}"/>)<br> &nbsp; <br>
  <h4><fmt:message key="Critics"/></h4>
  <blockquote><c:out value="${book.description}"/>
  </blockquote>
  <h4><fmt:message key="ItemPrice"/>:
  <fmt:formatNumber value="${book.price}" type="currency"/>
  </h4>
</c:forEach>
```

The following excerpt from `catalog.jsp` uses the `Row` interface to retrieve values from the columns of a book row using scripting language expressions. First the book row that matches a request parameter (`bid`) is retrieved from the database. Since the `bid` and `bookRow` objects are later used by tags that use scripting language expressions to set attribute values and a scriptlet that adds a book to the shopping cart, both objects are declared as scripting variables using the `jsp:useBean` tag. The page creates a bean that describes the book and scripting language expressions are used to set the book properties from book row column values. Finally the book is added to the shopping cart.

You might want to compare this version of `catalog.jsp` to the versions in JavaServer Pages Technology (page 607) and Custom Tags in JSP Pages (page 637) that use a book database JavaBeans component.

```
<sql:query var="books"
  dataSource="${applicationScope.bookDS}">
  select * from PUBLIC.books where id = ?
  <sql:param value="${bid}" />
```

```
</sql:query>
<c:forEach var="bookRow" begin="0"
      items="${books.rowsByIndex}">
  <jsp:useBean id="bid"  type="java.lang.String" />
  <jsp:useBean id="bookRow" type="java.lang.Object[]" />
  <jsp:useBean id="addedBook" class="database.BookDetails"
    scope="page" />
    <jsp:setProperty name="addedBook" property="bookId"
      value="<%=bookRow[0]%>" />
    <jsp:setProperty name="addedBook" property="surname"
      value="<%=bookRow[1]%>" />
    <jsp:setProperty name="addedBook" property="firstName"
      value="<%=bookRow[2]%>" />
    <jsp:setProperty name="addedBook" property="title"
      value="<%=bookRow[3]%>" />
    <jsp:setProperty name="addedBook" property="price"
      value="<%=((Double)bookRow[4]).floatValue()%>" />
    <jsp:setProperty name="addedBook" property="year"
      value="<%=(Integer)bookRow[5]%>" />
    <jsp:setProperty name="addedBook"
      property="description" value="<%=bookRow[6]%>" />
    <jsp:setProperty name="addedBook" property="inventory"
      value="<%=(Integer)bookRow[7]%>" />
  </jsp:useBean>
  <% cart.add(bid, addedBook); %>
  ...
</c:forEach>
```

Further Information

For further information on JSTL see:

- Reference documentation in the Java WSDP at *<JWSDP_HOME>*`/jstl-1.0.3/docs/index.html`.
- The JSTL examples in the Java WSDP. When Tomcat is running, you can access the examples at `http://localhost:8080/jstl-examples/index.html`.
- Resources listed on the Web site `http://java.sun.com/products/jsp/jstl`.
- The JSTL 1.0 Specification for a complete description of the syntax and semantics of JSTL.

18 Security

Debbie Bode Carson and Eric Jendrock

THE Web-tier security model used in the Java WSDP is based on the Java Servlet specification. This model insulates developers from mechanism-specific implementation details of application security. The Java WSDP provides this insulation in a way that enhances the portability of applications, allowing them to be deployed in diverse security environments.

Some of the material in this chapter assumes that you have an understanding of basic security concepts. To learn more about these concepts, we highly recommend that you explore the Security trail in *The Java™ Tutorial* (see `http://java.sun.com/docs/books/tutorial/security1.2/index.html`) before you begin this chapter.

Overview

The Java WSDP platform defines declarative contracts between those who develop and assemble application components and those who configure applications in operational environments. In the context of application security, application providers are required to declare the security requirements of their applications in such a way that these requirements can be satisfied during application configuration. The *declarative security* mechanisms used in an application are expressed in a declarative syntax in a document called a *deployment descriptor*. An application deployer then employs container-specific tools to map the application requirements that are in a deployment descriptor to security mechanisms that are implemented by Web components.

Programmatic security refers to security decisions that are made by security-aware applications. Programmatic security is useful when declarative security alone is not sufficient to express the security model of an application. For example, an application might make authorization decisions based on the time of day, the parameters of a call, or the internal state of a Web component. Another application might restrict access based on user information stored in a database.

Java Web Services applications are made up of components that can be deployed into different containers. These components are used to build a multi-tier application. The goal of the Java WSDP security architecture is to achieve end-to-end security by securing each tier.

The tiers can contain both protected and unprotected resources. Often, you need to protect resources to ensure that only authorized users have access. *Authorization* provides controlled access to protected resources. Authorization is based on identification and authentication. *Identification* is a process that enables recognition of an entity by a system, and *authentication* is a process that verifies the identity of a user, device, or other entity in a computer system, usually as a prerequisite to allowing access to resources in a system.

Authorization and authentication are not required to access unprotected resources. Accessing a resource without authentication is referred to as *unauthenticated* or *anonymous* access.

Users, Groups, and Roles

A Web services user is similar to an operating system user. Typically, both types of users represent people. However, these two types of users are not the same. The Tomcat server authentication service has no knowledge of the user and password you provide when you log on to the operating system. The Tomcat server authentication service is not connected to the security mechanism of the operating system. The two security services manage users that belong to different *realms*.

The Tomcat server authentication service includes the following components:

- *Role* - an abstract name for the permission to access a particular set of resources. A *role* can be compared to a key that can open a lock. Many people might have a copy of the key, and the lock doesn't care who you are, just that you have the right key.
- *User* - an individual (or application program) identity that has been authenticated (authentication was discussed in the previous section). A user can

have a set of *roles* associated with that identity, which entitles them to access all resources protected by those roles.

- *Group* - a set of authenticated *users* classified by common traits such as job title or customer profile. Groups are also associated with a set of *roles*, and every user that is a member of a group inherits all of the roles assigned to that group. In most cases, you will map users directly to roles and have no need to define a group.
- *Realm* - a complete database of *roles*, *users*, and *groups* that identify valid users of a Web application (or a set of Web applications).

Security Roles

When you design a Web component, you should always think about the kinds of users who will access the component. For example, a Web application for a Human Resources department might have a different request URL for someone who has been assigned the role of `admin` than for someone who has been assigned the role of `director`. The `admin` role may let you view some employee data, but the `director` role enables you to view salary information. Each of these *security roles* is an abstract logical grouping of users that is defined by the person who assembles the application. When an application is deployed, the deployer will map the roles to security identities in the operational environment.

To create a role for a Web services application, you can set up the users and roles using `admintool`, or in a deployment descriptor, then declare it for the WAR file that is contained in the application. For information on setting up users and roles using admintool, see Managing Roles and Users (page 704).

The following example code shows how you would set up roles in the `web.xml` deployment descriptor for the Web component. As is shown, you can define multiple security roles for an application.

```
<security-role>
  <role-name>admin</role-name>
</security-role>
<security-role>
  <description>Book customer</description>
  <role-name>customer</role-name>
</security-role>
```

The `<role-name>` that you specify in the deployment descriptor must have a corresponding entry in your server-specific deployment descriptor. For the Tomcat server, the file is `<`*`JWSDP_HOME`*`>/conf/tomcat-users.xml`. The entry needs to declare a mapping between a security role and one or more principals in the realm. An example for the Tomcat server might be as follows:

```
<?xml version='1.0'?>
<tomcat-users>
<user username='your_name' password='your_password'
  roles='admin,manager,provider'/>
<user username='Joe' password='scooby'
  roles='admin,manager,provider'/>
<user username='Mikhail' password='goblin'
  roles='admin'/>
</tomcat-users>
```

Another example for another Web Server might be as follows:

```
<rolemapping>
   <role name="admin">
   <principals>
      <principal>
         <name>Khalil Singh</name>
      </principal>
   </principals>
   </role>
</rolemapping>
```

Managing Roles and Users

To manage the information in the users file, we recommend that you use `admintool`. To use `admintool`, start Tomcat, then point your browser to `http://localhost:8080/admin` and log on with a user name and password combination that has been assigned the `admin` role, such as the user name and password that you entered during installation.

For security purposes, `admintool`, the Tomcat Web Server Administration Tool, verifies that you (as defined by the information you provide when you log into the application) are a user who is authorized to install and reload applications (defined as a user with the role of `admin` in `tomcat-users.xml`) before granting you access to the server.

The `<JWSDP_HOME>/conf/tomcat-users.xml` file is created by the installer. It contains, in plain text, the user name and password created during installation of the Java WSDP. This user name is initially associated with the predefined roles of `admin`, `manager`, and `provider`. You can edit the users file directly in order to add or remove users or modify roles, or you can use `admintool` to accomplish these tasks, as described herein.

The `tomcat-users.xml` file looks like this:

```
<?xml version='1.0'?>
<tomcat-users>
<user username='your_name' password='your_password'
  roles='admin,manager,provider'/>
</tomcat-users>
```

The following sections describe how to add roles and users using `admintool`. The file *`JWSDP_HOME`*`/conf/tomcat-users.xml` is updated as the changes are made in `admintool`.

Using the Tomcat Web Server Administration Tool

To use `admintool`, the Tomcat Web Server Administration Tool, you must start Tomcat. Before starting Tomcat, make sure that your `PATH` environment variables are set properly and that the `build.properties` file has been created properly. These steps are described in Setting Up (page 69).

Starting Tomcat

To start Tomcat, type the following command in a terminal window.

```
<JWSDP_HOME>/bin/startup.sh        (Unix platform)

<JWSDP_HOME>\bin\startup.bat       (Microsoft Windows)
```

The startup script starts the task in the background and then returns the user to the command line prompt immediately. The startup script does not completely start Tomcat for several minutes.

Note: The startup script for Tomcat can take several minutes to complete. To verify that Tomcat is running, point your browser to `http://localhost:8080`. When the Tomcat splash screen displays, you may continue. If the splash screen does not load immediately, wait up to several minutes and then retry. If, after several minutes, the Tomcat splash screen does not display, refer to the troubleshooting tips in “Unable to Locate the Server localhost:8080” Error (page 87).

Documentation for Tomcat can be found at <*JWSDP_HOME*>`/docs/tomcat/index.html`.

Starting admintool

Once the Tomcat server is started, follow these steps to start `admintool`.

1. Start a Web browser.
2. In the Web browser, point to the following URL:

   ```
   http://localhost:8080/admin
   ```

3. Log in to `admintool` using a user name and password combination that has been assigned the role of `admin`.

The `admintool` utility displays in the Web browser window:

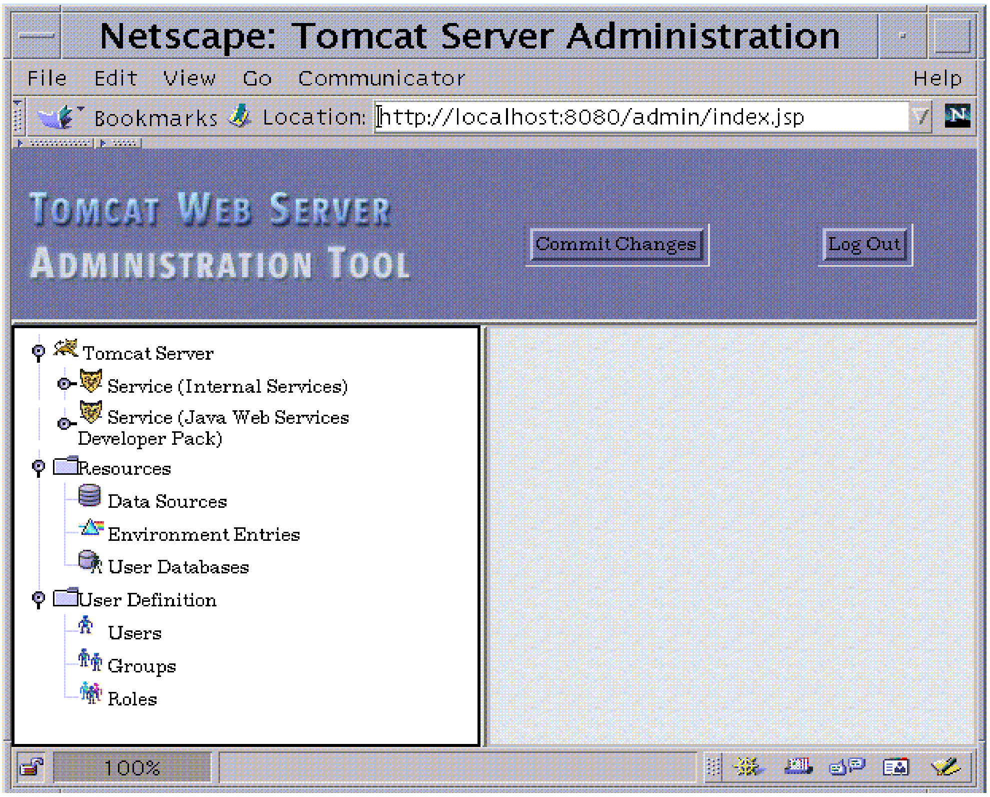


Figure 18–1 The Tomcat Web Server Administration Tool

4. When you have finished, log out of `admintool` by selecting Log Out.

The following sections show how to use `admintool` to do the following:

- Display all roles in the default realm
- Add a role to the default realm
- Remove a role from the default realm
- Display all users in the default realm
- Add a user to the default realm
- Remove a user

The modifications discussed in the following sections are made to the running Tomcat server—it is not necessary to stop and restart Tomcat.

Managing Roles

To view all existing roles in the realm, select Roles from the User Definition section in the left pane.

The Roles List and Role Actions list display in the right pane. By default, the roles defined during Java WSDP installation are displayed. These roles include `admin`, `manager`, and `provider`.

Use the following procedure to add a new role to the default realm.

1. From the Role Actions List, select Create New Role.
2. Enter the name of the role to add.
3. Enter the description of the role.
4. Select Save when done. The newly defined role displays in the list.

Use the following procedure to remove a role from the default realm.

1. From the Role Actions List, select Delete Existing Roles.
2. Select the role to remove by checking the box to its left.
3. Select Save.

If you entered a new role of `customer`, the `tomcat-users.xml` file would now look like this:

```
<?xml version='1.0'?>
<tomcat-users>
  <role rolename="customer" description="Customer of Java Web
    Service"/>
    <role rolename="provider"/>
    <role rolename="manager"/>
    <role rolename="admin"/>
    <user username="your_name" password="your_password"
      roles="admin,manager,provider"/>
  </tomcat-users>
```

Managing Users

To view all existing users in the realm, select Users from the User Definition section in the left pane.

The Users List and Available Actions list for User Actions display in the right pane. By default, the user name defined during Java WSDP installation is displayed.

Use the following procedure to edit a user's profile.

1. Select the user profile to edit in the right pane.
2. Edit the existing user properties. You can modify a password and/or modify role assignments in this window.

Use the following procedure to add a new user to the default realm.

1. From the User Actions List, select Create New User.
2. Enter the name of the user to add.
3. Enter the password for that user.
4. Enter the full name of the user.
5. Select the role assignments for this user.
6. Select Save when done. The newly defined user displays in the list.

Use the following procedure to remove a user from the default realm.

1. Select Delete Existing Users from the User Actions list.
2. Select the user to remove by checking the box to its left.
3. Select Save.

The addition of a new role and user as described in the previous section are reflected in the updated `tomcat-users.xml`. If I added a new user named Anil and assigned him the role of `customer`, the updated `tomcat-users.xml` would look like this:

```
<?xml version='1.0'?>
<tomcat-users>
  <role rolename="customer" description="Customer of Java Web
    Service"/>
  <role rolename="provider"/>
  <role rolename="manager"/>
  <role rolename="admin"/>
  <user username="your_name" password="your_password"
    roles="admin,manager,provider"/>
  <user username="Anil" password="12345" fullName=""
    roles="customer"/>
</tomcat-users>
```

Declaring and Linking Role References

A *security role reference* allows a Web component to reference an existing security role. A security role is an application-specific logical grouping of users, classified by common traits such as customer profile or job title. When an application is deployed, roles are mapped to security identities, such as *principals* (identities assigned to users as a result of authentication) or groups, in the operational environment. Based on this, a user with a certain security role has associated access rights to a Web application. The link is the actual name of the security role that is being referenced.

During application assembly, the assembler creates security roles for the application and associates these roles with available security mechanisms. The assembler then resolves the security role references in individual servlets and JSPs by linking them to roles defined for the application.

The security role reference defines a mapping between the name of a role that is called from a Web component using `isUserInRole(String name)` (see Using Programmatic Security in the Web Tier, page 719) and the name of a security role that has been defined for the application.

For example, the mapping of the security role reference `cust` to the security role with role name `bankCustomer`, is shown in the `<security-role-ref>` element of the deployment descriptor, as shown:

```
<security-role-ref>
  <role-name>cust</role-name>
  <role-link>bankCustomer</role-link>
</security-role-ref>
```

In this example, `isUserInRole("bankCustomer")` and `isUserInRole("cust")` will both return `true`.

Because a coded name is linked to a role name, you can change the role name at a later time without having to change the coded name. For example, if you were to change the role name from `bankCustomer` to something else, you wouldn't need to change the `cust` name in the code. However, you would need to relink the `cust` coded name to the new role name.

Mapping Application Roles to Realm Roles

When you are developing a Web services application, you don't need to know what roles have been defined for the realm in which the application will be run. In the Java WSDP, the Web services security architecture provides a mechanism for automatically mapping the roles defined in the application to the roles defined in the runtime realm. After your application has been deployed, the administrator of the Tomcat server will map the roles of the application to the roles of the `default` realm.

The following example shows the role mapping between the application-defined role `admin` and the `admin` role that was defined when the Java WSDP was installed. The following is an example of the security constraint that could be added to the application's deployment descriptor.

1. Select or open the Web application deployment descriptor, for example, `<`*`JWSDP_HOME`*`>/docs/tutorial/examples/gs/web/WEB-INF/web.xml`.
2. Add a security constraint such as the one shown below. In this example, the role of `admin` is authorized to access this application, and is assigned a security role. For example, in the Tomcat server, this information is reflected in the runtime XML, as shown:

```
<!-- SECURITY CONSTRAINT -->
<security-constraint>
  <web-resource-collection>
    <web-resource-name>WRCollection</web-resource-name>
    <url-pattern>/index.jsp</url-pattern>
    <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>NONE</transport-guarantee>
  </user-data-constraint>
</security-constraint>

<!-- SECURITY ROLES -->
<security-role>
  <description>the administrator role</description<
  <role-name>admin</role-name>
</security-role>
```

The `<role-name>` that you specify in the deployment descriptor must have a corresponding entry in your server-specific deployment descriptor. For the Tomcat server, the file is `<`*`JWSDP_HOME`*`>/conf/tomcat-users.xml`. The entry needs to declare a mapping between a security role and one or more principals in the realm. An example for the Tomcat server might be as follows:

```
<?xml version='1.0'?>
<tomcat-users>
<user username='your_name' password='your_password'
  roles='admin,manager,provider'/>
</tomcat-users>
```

Another example for another Web Server might be as follows:

```
<rolemapping>
   <role name="admin">
   <principals>
      <principal>
         <name>Khalil Singh</name>
      </principal>
   </principals>
   </role>
</rolemapping>
```

Web-Tier Security

The following sections address protecting resources and authenticating users in the Web tier.

Your Web application is defined using a standard `web.xml` deployment descriptor. The deployment descriptor must indicate which version of the web application schema (2.2, 2.3, or 2.4) it is using, and the elements specified within the deployment descriptor must comply with the rules for processing that version of the deployment descriptor. For version 2.4 of the Java Servlet Specification, this is “SRV.13.3, Rules for Processing the Deployment Descriptor”. These specifications may be downloaded from `http://java.sun.com/products/servlet/download.html`. For more information on deployment descriptors, see Chapter 4.

The deployment descriptor is used to convey the elements and configuration information of a Web application. Security in a Web application is configured using the following elements of the deployment descriptor:

- `<login-config>`
  The `<login-config>` element specifies how the user is prompted to login in. If this element is present, the user must be authenticated before it can access any resource that is constrained by a `<security-constraint>`. The `<login-config>` element is discussed in Configuring Login Authentication (page 716).
- `<security-constraint>`
  The `<security-constraint>` element is used to define the access privileges to a collection of resources using their URL mapping. Security constraints are discussed in Controlling Access to Web Resources (page 713).
- `<security-role>`
  The `<security-role>` element represents a defined group for the realm. Security roles are discussed in Security Roles (page 703).

These elements of the deployment descriptor may be entered directly into the `web.xml` file.

Some elements of Web application security need to be addressed in the deployment descriptor for the Web server, rather than the deployment descriptor for the Web application. This information is discussed in Installing and Configuring SSL Support (page 721), Using Programmatic Security in the Web Tier (page 719), and Security Roles (page 703).

Protecting Web Resources

You can protect Web resources by specifying a security constraint. A *security constraint* determines who is authorized to access a *Web resource collection*, a list of URL patterns and HTTP methods that describe a set of resources to be protected. Security constraints can be defined using a deployment descriptor, as discussed in Controlling Access to Web Resources (page 713).

If you try to access a protected Web resource as an unauthenticated user, the Web container will try to authenticate you. The container will only accept the request after you have proven your identity to the container and have been granted permission to access the resource.

Security constraints only work on the original request URI, not on calls made via a `RequestDispatcher` (which include `<jsp:include>` and `<jsp:forward>`). Inside the application, it is assumed that the application itself has complete access to all resources and would not forward a user request unless it had decided that the requesting user had access also.

Controlling Access to Web Resources

You can set up a security constraint by coding the information directly into the deployment descriptor between `<security-constraint></security-constraint>` tags. When you define security constraints, you need to make sure you have addressed the following issues:

- Set up login authentication (discussed in Configuring Login Authentication (page 716)).
- Add a security constraint.
- Add a web resource collection.
- Define and include an authorized security role (discussed in Security Roles (page 703)).
- Identify URL patterns to constrain.
- Identify HTTP methods to constrain (`POST`, `GET`).
- Specify whether there are any guarantees on how the data will be transported between client and server (`NONE`, `INTEGRAL`, `CONFIDENTIAL`).

If, for example, we were to look at the security portion of the deployment descriptor for a simple application, the `web.xml` file might look something like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC “-//Sun Microsystems, Inc.//DTD Web Application
     2.3//EN”
  “http://java.sun.com./dtd/web-app_2_3.dtd”>

...

  <display-name>SimpleApp</display-name>
  <servlet>
    <servlet-name>index</servlet-name>
    <display-name>index</display-name>
    <jsp-file>/index.jsp</jsp-file>
    <!-- SECURITY-ROLE-REF -->
    <security-role-ref>
      <role-name>SimpleAppCustomer</role-name>
      <role-link>customer</role-link>
    </security-role-ref>
  </servlet>
  <session-config>
    <session-timeout>30</session-timeout>
  </session-config>

  <!-- SECURITY CONSTRAINT -->
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>WRCollection</web-resource-name>
      <url-pattern>/index.jsp</url-pattern>
      <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
      <role-name>customer</role-name>
    </auth-constraint>
    <user-data-constraint>
      <transport-guarantee>NONE</transport-guarantee>
    </user-data-constraint>
  </security-constraint>

  <!-- LOGIN AUTHENTICATION -->
  <login-config>
    <realm-name></realm-name>
    <auth-method>BASIC</auth-method>
  </login-config>
```

```
    <!-- SECURITY ROLES -->
    <security-role>
      <role-name>admin</role-name>
    </security-role>
    <security-role>
      <description>Simple App Customer</description>
      <role-name>customer</role-name>
    </security-role>
    <security-role>
      <role-name>manager</role-name>
    </security-role>
    <security-role>
      <role-name>provider</role-name>
    </security-role>

  ...
```

Authenticating Users of Web Resources

When you try to access a protected Web resource, the Web container activates the authentication mechanism that has been configured for that resource. You can configure the following authentication mechanisms for a Web resource:

- None

  If you do not specify one of the following methods, the user will not be authenticated.

- HTTP Basic authentication

  If you specify *HTTP basic authentication,* (`<auth-method>BASIC</auth-method>`), the Web server will authenticate a user by using the user name and password obtained from the Web client.

- Form-based authentication

  If you specify *form-based authentication* (`<auth-method>FORM</auth-method>`), you can customize the login screen and error pages that are presented to the end user by an HTTP browser.

  Neither form-based authentication nor HTTP basic authentication is particularly secure. In form-based authentication, the content of the user dialog is sent as plain text, and the target server is not authenticated. Basic authentication sends user names and passwords over the Internet as text that is uu-encoded, but not encrypted. This form of authentication, which uses Base64 encoding, can expose your user names and passwords unless all connections are over SSL. If someone can intercept the transmission, the user name and password information can easily be decoded.

- Client-certificate authentication

  Client-certificate authentication (`<auth-method>CLIENT-CERT</auth-method>`) is a more secure method of authentication than either basic or form-based authentication. It uses HTTP over SSL, in which the server and, optionally, the client authenticate each other with Public Key Certificates. *Secure Sockets Layer* (SSL) provides data encryption, server authentication, message integrity, and optional client authentication for a TCP/IP connection. You can think of a *public key certificate* as the digital equivalent of a passport. It is issued by a trusted organization, which is called a *certificate authority* (CA), and provides identification for the bearer. If you specify client-certificate authentication, the Web server will authenticate the client using the client's *X.509 certificate*, a public key certificate that conforms to a standard that is defined by X.509 Public Key Infrastructure (PKI). Prior to running an application that uses SSL, you must configure SSL support on the server (see Installing and Configuring SSL Support, page 721) and set up the public key certificate (see Setting Up Digital Certificates (page 723)).

- Digest authentication

  Digested password authentication (`<auth-method>DIGEST</auth-method>`) supports the concept of digesting user passwords. This causes the stored version of the passwords to be encoded in a form that is not easily reversible, but that the Web server can still utilize for authentication.

  From a user perspective, digest authentication acts almost identically to basic authentication in that it triggers a login dialog. The difference between basic and digest authentication is that on the network connection between the browser and the server, the password is encrypted, even on a non-SSL connection. In the server, the password can be stored in clear text or encrypted text, which is true for all login methods and is independent of the choice that the application deployer makes.

Configuring Login Authentication

You can set up login authentication by coding the information directly into the deployment descriptor between `<login-config></login-config>` tags. When you configure the authentication mechanism that the Web resources in a WAR will use, you have the following options:

- Specify one of the user authentication methods described in Authenticating Users of Web Resources (page 715).
- Specify a security realm. If omitted, the `default` realm is assumed.
- If the authentication method is specified as FORM, specify a form login page and form error page.

  The form login page defines the location of the form that will be used to authenticate the user. The form error page is the resource that responds to a failed authentication.

The following sample code shows a section of a deployment descriptor that uses form-based login authentication. The `<form-login-page>` element provides the URI of a Web resource relative to the document root that will be used to authenticate the user. The login page can be an HTML page, a JSP page, or a servlet, and must return an HTML page containing a form that conforms to specific naming conventions (see the relevant Servlet specification for more information on these requirements). The `<form-error-page>` element requires a URI of a Web resource relative to the document root that send a response when authentication has failed.

A *Universal Resource Identifier* (URI), is a globally unique identifier for a resource. A *Universal Resource Locator* (URL) is a kind of URI that specifies the retrieval protocol (`http` or `https` for Web applications) and physical location of a resource (host name and host-relative path).

In the Java Servlet specification, the request URI is the part of a URL *after* the host name and port. For example, in the URL `http://localhost:8080/myApp/jsp/hello.jsp`, the request URI would be `/jsp/hello.jsp`. The request URI is further subdivided into the context path (which decides which Web application should process the request) and the rest of the path that is used to select the target servlet.

```
<!-- LOGIN AUTHENTICATION -->
  <login-config>
    <auth-method>FORM</auth-method>
    <realm-name>default</realm-name>
    <form-login-config>
      <form-login-page>login.jsp</form-login-page>
      <form-error-page>error.jsp</form-error-page>
    </form-login-config>
  </login-config>
```

Using SSL to Enhance the Confidentiality of HTTP Basic and Form-Based Authentication

Passwords are not protected for confidentiality with HTTP basic or form-based authentication, meaning that passwords sent between a client and a server on a non-protected session can be viewed and intercepted by third parties. To overcome this limitation, you can run these authentication protocols over an SSL-protected session and ensure that all message content is protected for confidentiality.

If the default configuration of your Web server does not support SSL, you must configure it with an SSL connector to make this work. The default configuration of the Tomcat server does not include an SSL Connector. To configure Tomcat for SSL, follow the instructions in Installing and Configuring SSL Support (page 721).

To configure HTTP basic or form-based authentication over SSL, specify `CONFIDENTIAL` or `INTEGRAL` as the user authentication method within the `<transport-guarantee>` elements. Specify `CONFIDENTIAL` when the application requires that data be transmitted so as to prevent other entities from observing the contents of the transmission. Specify `INTEGRAL` when the application requires that the data be sent between client and server in such a way that it cannot be changed in transit. The following example code from a `web.xml` file shows this setting in context:

```
<!-- SECURITY CONSTRAINT -->
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>WRCollection</web-resource-name>
      <url-pattern>/index.jsp</url-pattern>
      <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
      <role-name>customer</role-name>
    </auth-constraint>
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
```

If you specify `CONFIDENTIAL` or `INTEGRAL` as a security constraint, that type of security constraint applies to all requests that match the URL patterns in the Web resource collection, not just to the login dialog.

Note: Good Security Practice: If you are using sessions, once you switch to SSL you should never accept any further requests for that session that are non-SSL. For example, a shopping site might not use SSL until the checkout page, then it may switch to using SSL in order to accept your card number. After switching to SSL, you should stop listening to non-SSL requests for this session. The reason for this practice is that the session ID itself was non-encrypted on the earlier communications, which is not so bad when you're just doing your shopping, but once the credit card information is stored in the session, you don't want a bad guy trying to fake the purchase transaction against your credit card. This practice could be easily implemented using a filter.

Using Programmatic Security in the Web Tier

Programmatic security is used by security-aware applications when declarative security alone is not sufficient to express the security model of the application. Programmatic security consists of the following methods of the `HttpServletRequest` interface:

- `getRemoteUser` - used to determine the user name with which the client authenticated.
- `isUserInRole` - used to determine if a user is in a specific security role.
- `getUserPrincipal` - returns a `java.security.Principal` object.

These APIs allow servlets to make business logic decisions based on the logical role of the remote user. They also allow the servlet to determine the principal name of the current user.

When you use the `isUserInRole(String role)` method, the String `role` is mapped to the role name defined in the `<role-name>` element nested within the `<security-role-ref>` element of a `<servlet>` declaration of the `web.xml` deployment descriptor. The `<role-link>` element must match a `<role-name>` defined in the `<security-role>` element of the `web.xml` deployment descriptor. If the `isUserInRole("admin")` method is called within a servlet, the section of example code in **bold** below would need to be added to ensure security. In this example, the `<role-link>` parameters are used in the application, the `<role-`

`name>` element provides some form of abstraction. The applicable sections of the `web.xml` deployment descriptor would look like this:

```
<servlet>
   ...
   <role-name>administrator</role-name>
   <role-link>admin</role-link>
   ...
</servlet>

<security-role>
   <role-name>admin</role-name>
</security-role>
```

As discussed in Security Roles (page 703), there also must be a corresponding `<role-name>` entry in the Web server-specific deployment descriptor, which would look something like this:

```
<role name="admin">
   <principals>
      <principal>
         <name>Wanda</name>
      </principal>
      <principal>
         <name>Raja</name>
      </principal>
   </principals>
</role>
```

Creating the Login Form

The content of the login form in an HTML page, JSP page, or servlet for a login page should be as follows:

```
<form method="POST"  action="j_security_check" >
 <input type="text"  name= "j_username" >
 <input type="password"  name= "j_password" >
</form>
```

See the Servlet specification at `http://java.sun.com/products/servlet/` for additional information.

Unprotected Web Resources

Many applications feature unprotected Web content, which any caller can access without authentication. In the Web tier, unrestricted access is provided simply by not configuring a security constraint for that particular request URI. It is common to have some unprotected resources and some protected resources. In this case, you will have security constraints and a login method defined, but it will not be used to control access to the unprotected resources. The user won't be asked to log on until the first time they enter a protected request URI.

In the Java Servlet specification, the request URI is the part of a URL *after* the host name and port. For example, let's say you have an e-commerce site with a browsable catalog you would want anyone to be able to access and a shopping cart area for customers only. You could set up the paths for your Web application so that the pattern `/cart/*` is protected, but nothing else is protected. Assuming the application is installed at context path `/myapp`,

- `http://localhost:8080/myapp/index.jsp` is *not* protected
- `http://localhost:8080/myapp/cart/index.jsp` *is* protected

A user will not be prompted to log in until the first time that user accesses a resource in the `cart` subdirectory.

Installing and Configuring SSL Support

What is Secure Socket Layer Technology?

Secure Socket Layer (SSL) is a technology that allows Web browsers and Web servers to communicate over a secured connection. In this secure connection, the data that is being sent is encrypted before being sent, then decrypted upon receipt and prior to processing. Both the browser and the server encrypt all traffic before sending any data. SSL addresses the following important security considerations.

Authentication

During your initial attempt to communicate with a Web server over a secure connection, that server will present your Web browser with a set of credentials in the form of a server certificate. The purpose of the certificate is to verify that the site is who and what it claims to be. In some cases, the server may request a certificate that the client is who and what it claims to be (which is known as client authentication).

Confidentiality

When data is being passed between the client and server on a network, third parties can view and intercept this data. SSL responses are encrypted so that the data cannot be deciphered by the third-party and the data remains confidential.

Integrity

When data is being passed between the client and server on a network, third parties can view and intercept this data. SSL helps guarantee that the data will not be modified in transit by that third party.

To install and configure SSL support on your stand-alone Web server, you need the following components. The following sections discuss enabling SSL support for Tomcat specifically. If you are using a different Web server, consult the documentation for your product.

- Java Secure Socket Extension (JSSE) (see Using JSSE, page 722).
- A server certificate keystore (see Setting Up Digital Certificates (page 723)).
- An HTTPS connector (see Configuring the SSL Connector, page 729).

To verify that SSL support is enabled, see Verifying SSL Support (page 732).

Using JSSE

If you are using J2SE SDK v1.3.1, you need to have Java Secure Socket Extension (JSSE) installed in order to use SSL. JSSE is a part of the J2SE 1.4 SDK. JSSE is a set of Java packages that enable secure Internet communications. These packages implement a Java version of SSL (Secure Sockets Layer) and TLS (Transport Layer Security) protocols and include functionality for data encryption, server authentication, message integrity, and optional client authentication. Using JSSE, developers can provide for the secure passage of data between a client and a server running any application protocol (such as HTTP, Telnet, NNTP, and FTP) over TCP/IP.

By default, the location of the `jsse.jar` file is <*JAVA_HOME*>`/jre/lib/jsse.jar`. For more information on JSSE, see its Web site at `http://java.sun.com/products/jsse/`.

Setting Up Digital Certificates

In order to implement SSL, a Web server must have an associated certificate for each external interface, or IP address, that accepts secure connections. The theory behind this design is that a server should provide some kind of reasonable assurance that its owner is who you think it is, particularly before receiving any sensitive information. It may be useful to think of a certificate as a “digital driver’s license” for an Internet address. It states with which company the site is associated, along with some basic contact information about the site owner or administrator.

The digital certificate is cryptographically signed by its owner and is difficult for anyone else to forge. For sites involved in e-commerce, or any other business transaction in which authentication of identity is important, a certificate can be purchased from a well-known Certificate Authority (CA) such as Verisign or Thawte.

If authentication is not really a concern, such as if an administrator simply wants to ensure that data being transmitted and received by the server is private and cannot be snooped by anyone eavesdropping on the connection, you can simply save the time and expense involved in obtaining a CA certificate and simply use a self-signed certificate.

SSL uses *public key cryptography*, which is based on *key pairs*. Key pairs contain one public key and one private key. If data is encrypted with one key, it can only be decrypted with the other key of the pair. This property of is fundamental to establishing trust and privacy in transactions. For example, using SSL, the server computes a value and encrypts the value using its private key. The encrypted value is called a *digital signature*. The client decrypts the encrypted value using the server’s public key and compares the value to its own computed value. If the two values match, the client can trust that the signature is authentic since only the private key could have been used to produce such a signature.

Digital certificates are used with the HTTPS protocol to authenticate Web clients. The HTTPS service of most Web servers will not run unless a digital certificate has been installed. Use the procedure outlined below to set up a digital certificate that can be used by your Web server to enable SSL.

One tool that can be used to set up a digital certificate is `keytool`, a key and certificate management utility that ships with the J2SE 1.4 SDK. It enables users to administer their own public/private key pairs and associated certificates for use in self-authentication (where the user authenticates himself/herself to other users/services) or data integrity and authentication services, using digital signatures. It also allows users to cache the public keys (in the form of certificates) of their communicating peers. For a better understanding of public key cryptography, read the `keytool` documentation at

`http://java.sun.com/j2se/1.4.1/docs/tooldocs/solaris/keytool.html`

A certificate is a digitally-signed statement from one entity (person, company, etc.) saying that the public key (and some other information) of some other entity has a particular value. When data is digitally signed, the signature can be verified to check the data integrity and authenticity. *Integrity* means that the data has not been modified or tampered with, and *authenticity* means the data indeed comes from whoever claims to have created and signed it.

The `keytool` stores the keys and certificates in a file termed a *keystore*. The default keystore implementation implements the keystore as a file. It protects private keys with a password. For more information on `keytool`, read its documentation at

`http://java.sun.com/j2se/1.4.1/docs/tooldocs/solaris/keytool.html`

This section describes creating a server keystore called `server.keystore` and a client keystore called the `client.keystore`. The two files make a key pair. These files are usually created in either your *<HOME>* directory or in the application directory.

In addition to the server and client keystores, you must have a signed certificate, which must be present on the server. This file must contain the public key certificates of the Certificate Authority or the client's public key certificate at the time the server is authenticating the client. We will create the `server.cer` file in the *<HOME>* directory.

Typically, a keystore file is protected by a password. The default value for this password is `changeit` for `server.keystore`, `client.keystore`, and `server.cer` files.

To create a keystore file, we use the `keytool` utility. The `keytool` utility can be found in the *<JAVA_HOME>*`/bin` directory.

To set up a digital certificate,

1. Generate a key pair.

   The `keytool` utility enables you to generate the key pair. The `keytool` utility that ships with the J2SE SDK programmatically adds a Java Cryptographic Extension provider that has implementations of RSA algorithms. This provider enables you to import RSA-signed certificates.

To generate the keystore file, run the `keytool` utility as follows, replacing <*`keystore_filename`*> with the name of your keystore file, for example, `server.keystore`. If you are using the Tomcat server, the file must either be named .keystore and located in the home directory of the machine on which Tomcat is running, or you will need to tell Tomcat where the kestore file is by adding a `keystoreFile` attribute to the `<Factory>` element in the Tomcat configuration file or by specifying the location of the file on the `Connector (8443)` node of `admintool`.

```
keytool -genkey -keyalg RSA -alias tomcat-server
  -keystore <keystore_filename>
```

2. The `keytool` utility prompts you for the following information:
   a. Keystore password—Enter the default password, which is `changeit`. Refer to the `keytool` documentation for information on changing the password.
   b. First and last name—Enter the appropriate value, for example, `JWSDP`.
   c. Organizational unit—Enter the appropriate value, for example, `Java Web Services`.
   d. Organization—Enter the appropriate value, for example, `Sun Microsystems`.
   e. City or locality—Enter the appropriate value, for example, `Santa Clara`.
   f. State or province—Enter the unabbreviated name, for example, `CA`.
   g. Two-letter country code—For the USA, the two-letter country code is `US`.
   h. Review the information you've entered so far, enter `Yes` if it is correct.
   i. Key password for the Web server—Do not enter a password. Press Return.

The next step is generate a signed certificate for this keystore. A self-signed certificate is acceptable for most SSL communication. If you are using a self-signed certificate, continue with Creating a Self-Signed Certificate (page 726). If you'd like to have your certificate digitally signed by a CA, continue with Obtaining a Digitally-Signed Certificate (page 726).

Creating a Self-Signed Certificate

This example assumes that the keystore is named `server.keystore`, the certificate file is `server.cer`, and the CA file is `cacerts.jks`. Run these commands in your *<HOME>* directory so that they are created there.

1. Export the server certificate to a certificate file:

   ```
   keytool -keystore server.keystore -export -alias tomcat-
   server -file server.cer
   ```

2. Enter the password (`changeit`).

   `Keytool` returns the following message:

   ```
   Certificate stored in file <server.cer>
   ```

3. Import the new server certificate into the Certificate Authority file `cacerts.jks`:

   ```
   keytool -import -alias serverCA -keystore <HOME>/cacerts.jks
   -file server.cer
   ```

4. Enter the password (`changeit`).

   `Keytool` returns a message similar to the following:

   ```
   Owner: CN=JWSDP, OU=Java Web Services, O=Sun, L=Santa Clara,
   ST=CA, C=US
   Issuer: CN=JWSDP, OU=Java Web Services, O=Sun, L=Santa
   Clara,
   ST=CA, C=US
   Serial number: 3e39e3e0
   Valid from: Thu Jan 30 18:48:00 PST 2003 until: Wed Apr 30
   19:48:00 PDT 2003
   Certificate fingerprints:
   MD5:  44:89:AF:54:FE:79:66:DB:0D:BE:DC:15:A9:B6:09:84
   SHA1:21:09:8A:F6:78:E5:C2:19:D5:FF:CB:DB:AB:78:9B:98:8D:06:
   8C:71
   Trust this certificate? [no]:  yes
   Certificate was added to keystore
   ```

Obtaining a Digitally-Signed Certificate

This example assumes that the keystore is named `server.keystore`, the certificate file is `server.cer`, and the CA file is `cacerts.jks`.

1. Get your certificate digitally signed by a CA. To do this,

   a. Generate a Certificate Signing Request (CSR).

```
keytool -certreq -alias tomcat-server -keyalg RSA
    -file <csr_filename> -keystore cacerts.jks
```

b. Send the contents of the *`csr_filename`* for signing.

c. If you are using Verisign CA, go to `http://digitalid.verisign.com/`. Verisign will send the signed certificate in e-mail. Store this certificate in a file.

d. Import the signed certificate that you received in email into the server:

```
keytool -import -alias tomcat-server -trustcacerts -file
   <signed_cert_file> -keystore <keystore_filename>
```

2. Import the certificate (if using a CA-signed certificate).

   If your certificate will be signed by a Certification Authority (CA), you must import the CA certificate. You may skip this step if you are using only the self-signed certificate. If you are using a self-signed certificate or a certificate signed by a CA that your browser does not recognize, a dialog will be triggered the first time a user tries to access the server. The user can then choose to trust the certificate for this session only or permanently.

   To install the CA certificate in the Java 2 Platform, Standard Edition, run the `keytool` utility as follows.

```
keytool -import -trustcacerts -alias root
     -file <ca-cert-filename> -keystore <keystore-filename>
```

Creating a Client Certificate for Mutual Authentication

Creating a client certificate is similar to the procedure for server certificates.

1. Use `keytool` to create a client certificate in a keystore file of your choice:

```
keytool -genkey -keyalg RSA -alias jwsdp-client -keystore
client.keystore
```

   You will be prompted for a password. Enter `changeit`, the default password. When requested enter the name, organization, and other prompts for the client. Do not enter anything at “Key password for <client>”, just press Return.

2. Export the new client certificate from the keystore to a certificate file:

```
keytool -keystore client.keystore -export -alias jwsdp-cli-
ent -file client.cer
```

3. Enter the keystore password (`changeit`). `Keytool` returns this message:

```
Certificate stored in file <client.cer>
```

4. Import the new client certificate into the server's Certificate Authority file `cacerts.jks`. This allows the server to trust the client during SSL mutual authentication.

```
keytool -import -alias root -keystore <HOME>/cacerts.jks
-file client.cer
```

5. Enter the keystore password (`changeit`). `Keytool` returns this message:

```
Owner: CN=JWSDP Client, OU=Java Web Services, O=Sun, L=Santa
Clara, ST=CA, C=US
Issuer:  CN=JWSDP  Client,  OU=Java  Web  Services,  O=Sun,
L=Santa Clara, ST=CA, C=US
Serial number: 3e39e66a
Valid from: Thu Jan 30 18:58:50 PST 2003 until: Wed Apr 30
19:58:50 PDT 2003
Certificate fingerprints:
MD5:  5A:B0:4C:88:4E:F8:EF:E9:E5:8B:53:BD:D0:AA:8E:5A
SHA1:90:00:36:5B:E0:A7:A2:BD:67:DB:EA:37:B9:61:3E:26:B3:89:
46:
32
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

Checking That Mutual Authentication is Running

To prove that the SSL handshaking is occurring, shutdown Tomcat, set the `debug` flag in the file <*JWSDP_HOME*>`/bin/catalina.bat`, then restart Tomcat. The server will display the handshake messages, or write them to the file <*JWSDP_HOME*>`/logs/launcher.server.log`. The following example shows the new code in **bold**.

```
rem Execute the Tomcat launcher
"%JAVA_HOME%\bin\java.exe" -Djavax.net.debug=ssl,handshake
-classpath %PRG%\..;%PRG%\..\..\jwsdp-shared\bin;"%PRG%
```

Using a PKCS12 Certificate in the Tomcat Server

The Java WSDP supports PKCS12-format certificates. PKCS12 standard specifies a portable format for storing or transporting a user's private keys, certificates, miscellaneous secrets, etc. See the following Web site for additional information:

```
http://www.rsasecurity.com/rsalabs/pkcs/pkcs-12
```

If you have a PKCS12-format certificate, you must convert it into JKS format. The command for the conversion is:

```
keytool -pkcs12 -pkcsFile <fileName> -pkcsKeyStorePass
<password> -pkcsKeyPass <password> -jksFile <outputFileName>
-jksKeyStorePass <password>
```

The result is a JKS file that has the key – private key and the certificate chain – in the file.

To export the certificate into a file, such as `abc.cer`, use `keytool` with the `-export` option:

```
keytool -keystore <outputFileName> -export -alias <server>
-file abc.cer
```

Miscellaneous Commands for Certificates

- To check the contents of the server certificate:

```
keytool -list -keystore server.keystore -alias tomcat-server -v
```

- To check the contents of the `cacerts` file:

```
keytool -list -keystore cacerts.jks
```

Configuring the SSL Connector

Depending on your Web Server, an SSL HTTPS Connector may or may not be enabled. If you are using the Tomcat server, an SSL connector is not configured.

This section describes how to configure the SSL HTTPS Connector for Tomcat. If you are using another Web Server, consult the documentation for that server.

A `Connector` element for an SSL connector must be included in the server deployment descriptor. Before making changes to the server deployment descriptor, you must shut down the server. The following code is an example of code that will enable an SSL Connector for a Web server:

```
<Connector
   className="org.apache.coyote.tomcat5.CoyoteConnector"
   port="8443" minProcessors="5" maxProcessors="75"
   enableLookups="true" acceptCount="10" debug="0"
   scheme="https" secure="true" useURIValidationHack="false">
<Factory className="com.sun.web.security.SSLSocketFactory"
   clientAuth="false" protocol="TLS" debug="0" />
</Connector>
```

The attributes in this Connector element are described in more detail in the documentation for the Tomcat Server Administration Tool in Appendix A. You can add an SSL HTTPS Connector to Tomcat using either of these two methods:

- Add the `Connector` using `admintool`. See Adding an SSL Connector in admintool (page 730).
- Add a `Connector` element for an SSL connector to the server's deployment descriptor. See Configuring the SSL Connector in server.xml (page 731).

Adding an SSL Connector in admintool

To configure an SSL Connector using `admintool`, you must first have created a keystore as described in Setting Up Digital Certificates (page 723). Tomcat will be looking for a keystore file named `.keystore` in the home directory of the machine on which Tomcat is running. When you have verified that you have created the keystore file, follow these steps.

1. Start Tomcat, if you haven't already done so.
2. Start `admintool` by entering `http://localhost:8080/admin` in a Web browser.
3. Enter a user name and password combination that is assigned the role of `admin`.
4. Select Service (Java Web Services Developer Pack) in the left pane.
5. Select Create New Connector from the drop-down list in the right pane.
6. In the Type field, select HTTPS.

7. In the Port field, enter 8443 (or whatever port you require). This defines the TCP/IP port number on which Tomcat will listen for secure connections.
8. Enter the Keystore Name and Keystore Password if you have created a keystore named something other than `.keystore`, if `.keystore` is located in a directory other than the home directory of the machine on which Tomcat is running, or if the password is something other than the default value of `changeit`. If you have used the expected values, you can leave these fields blank.

   The home directory is generally `/home/`*`user_name`* on Unix and Linux systems, and `C:\Documents and Settings\user_name` on Microsoft Windows systems.
9. Select Save to save the new Connector for this session.
10. Select Commit Changes to write the new Connector information to the `server.xml` file so that it is available the next time Tomcat is started.

To view and/or edit the newly-created Connector, expand the Service (Java Web Services Developer Pack) node, and select Connector (8443).

Configuring the SSL Connector in server.xml

An example `Connector` element for an SSL connector is included in the default `server.xml`. This `Connector` element is commented out by default. To enable the SSL Connector for Tomcat, remove the comment tags around the SSL **Connector** element. To do this, follow these steps.

1. Shutdown Tomcat, if it is running. Changes to the file <*JWSDP_HOME*>`/conf/server.xml` are read by Tomcat when it is started.
2. Open the file <*JWSDP_HOME*>`/conf/server.xml` in a text editor.
3. Find the following section of code in the file (try searching for SSL Connector). Remove comment tags around the Connector entry. The comment tags that are to be removed are shown in bold below.

```
      <!-- SSL Connector on Port 8443 -->

      <!--
        <Connector

     className="org.apache.coyote.tomcat4.CoyoteConnector"
          port="8443" minProcessors="5"
          maxProcessors="75"
          enableLookups="false"
          acceptCount="10"
      connectionTimeout="60000" debug="0"
      scheme="https" secure="true">
    <Factory
  className="org.apache.coyote.tomcat4.
             CoyoteServerSocketFactory"
             clientAuth="false" protocol="TLS" />
    </Connector>
  -->
```

4. Save and close the file.
5. Start Tomcat.

The attributes in this Connector element are outlined in more detail in Tomcat Administration Tool (page 785) documentation.

Verifying SSL Support

For testing purposes, and to verify that SSL support has been correctly installed, load the default introduction page with a URL that connects to port defined in the server deployment descriptor:

```
https://localhost:8443/
```

The `https` in this URL indicates that the browser should be using the SSL protocol. The port of `8443` is where the SSL Connector was created in the previous step.

The first time a user loads this application, the New Site Certificate dialog displays. Select Next to move through the series of New Site Certificate dialogs, select Finish when you reach the last dialog.

General Tips on Running SSL

The SSL protocol is designed to be as efficient as securely possible. However, encryption/decryption is a computationally expensive process from a performance standpoint. It is not strictly necessary to run an entire Web application over SSL, and it is customary for a developer to decide which pages require a secure connection and which do not. Pages that might require a secure connection include login pages, personal information pages, shopping cart checkouts, or any pages where credit card information could possibly be transmitted. Any page within an application can be requested over a secure socket by simply prefixing the address with `https:` instead of `http:`. Any pages which absolutely require a secure connection should check the protocol type associated with the page request and take the appropriate action if `https:` is not specified.

Using name-based virtual hosts on a secured connection can be problematic. This is a design limitation of the SSL protocol itself. The SSL handshake, where the client browser accepts the server certificate, must occur before the HTTP request is accessed. As a result, the request information containing the virtual host name cannot be determined prior to authentication, and it is therefore not possible to assign multiple certificates to a single IP address. If all virtual hosts on a single IP address need to authenticate against the same certificate, the addition of multiple virtual hosts should not interfere with normal SSL operations on the server. Be aware, however, that most client browsers will compare the server's domain name against the domain name listed in the certificate, if any (applicable primarily to official, CA-signed certificates). If the domain names do not match, these browsers will display a warning to the client. In general, only address-based virtual hosts are commonly used with SSL in a production environment.

Troubleshooting SSL Connections

When Tomcat starts up, I get an exception like "java.io.FileNotFoundException: {some-directory}/{some-file} not found".

A likely explanation is that Tomcat cannot find the keystore file where it is looking. By default, Tomcat expects the keystore file to be named `.keystore`, and to be located in the home directory on the system under which Tomcat is running (which may or may not be the same as yours). If the keystore file is anywhere else, you will need to add a `keystoreFile` attribute to the `<Factory>` element in the Tomcat configuration file or specify the location of the file on the `Connector (8443)` node of `admintool`.

When Tomcat starts up, I get an exception like "java.io.FileNotFoundException: Keystore was tampered with, or password was incorrect".

Assuming that someone has not actually tampered with your keystore file, the most likely cause is that Tomcat is using a different password than the one you used when you created the keystore file. To fix this, you can either go back and recreate the keystore file, or you can add or update the `keystorePass` attribute on the `<Factory>` element in the Tomcat configuration file or on the Connector (8443) node of `admintool`. REMINDER - Passwords are case sensitive!

If you are still having problems,

If you are still having problems, a good source of information is the TOMCAT-USER mailing list. You can find pointers to archives of previous messages on this list, as well as subscription and unsubscription information, at `http://jakarta.apache.org/site/mail.html`.

Further information on SSL

For more information, please read the Tomcat document *SSL Configuration HOW-TO*, located at `<`*`JWSDP_HOME`*`>/docs/tomcat/ssl-howto.html`.

Security for JAX-RPC

In this section, you'll learn how to configure JAX-RPC-based Web service applications for basic and mutual authentication over HTTP/SSL. If the topic of authentication is new to you, please refer to the section titled Authenticating Users of Web Resources (page 715).

For this tutorial, we are going to modify the example application in `<`*`JWSDP_HOME`*`>/docs/tutorial/examples/jaxrpc/hello` to add HTTP/S basic and mutual authentication. The resulting application can be found in the directory `<`*`JWSDP_HOME`*`>/docs/tutorial/examples/jaxrpc/security`. The following steps are necessary to add basic authentication to the `hello` example:

- Create the appropriate certificates and keystores (see Step 1: Creating SSL Certificates for Basic Authentication, page 735).
- Make sure that your server is configured for an SSL Connector. The Tomcat server is not configured with an SSL Connector, so you need to add the SSL Connector (see Configuring the SSL Connector, page 729) and add

information on the generated keystore files (see Step 2: Configuring the SSL Connector, page 737).

- Add security elements to the `web.xml` deployment descriptor. See Step 3: Adding Security Elements to web.xml (page 737) for information on how to do this.
- Modify the endpoint address in the `build.properties` file for the application, and add other properties needed to run this example. See Step 4: Editing the Build Properties (page 739).
- Set security properties in the client code. See Step 5: Setting Security Properties in the Client Code (page 739) for information on how to do this.
- Build and run the Web service. See Step 6: Create a New Ant Target for Running this Example (page 741) for information on how to do this for the example application.

The steps for configuring a Web service for basic authentication over HTTP/S are outlined here. For mutual authentication, follow these steps, then add client authentication as discussed in Enabling Mutual Authentication Over SSL (page 742).

Step 1: Creating SSL Certificates for Basic Authentication

Note: This information is discussed in more detail in Setting Up Digital Certificates (page 723). This section provides a summary of the steps needed to create the SSL Certificates for this example.

We will use the tool `keytool` to generate SSL certificates and export them to the appropriate server and client keystores. Keep in mind that the server and client keystores are created in the directory from which you run `keytool`. Since we are adding security to the `hello` Web service, we will run `keytool` from the directory where the modified example application resides, which is the *`<JWSDP_HOME>`*`/docs/tutorial/examples/jaxrpc/security` directory. In so doing, the keystores are created in the same directory as the code for the `security` Web service.

1. Run `keytool` to generate the server and client keystores. For basic authentication, it is only necessary to import the server certificate into the client

keystore. Generate the server keystore with a default password of `changeit`.

To generate the server keystore, enter the following in a terminal window. Be sure that you are in the directory *<JWSDP_HOME>*`/docs/tutorial/examples/jaxrpc/security` before proceeding.

Note that when you press Enter, `keytool` prompts you to enter the server name, organizational unit, organization, locality, state, and country code. Note that **you must enter the server name in response to keytool's first prompt in which it asks for first and last names**. For testing purposes, this may be `localhost`. This host must match the host identified in the endpoint address specified in Step 4: Editing the Build Properties (page 739).

```
<J2SE_HOME>\bin\keytool -genkey -alias tomcat-server -keyalg
RSA -keypass changeit -storepass changeit -keystore
server.keystore
```

2. Export the generated server certificate.

```
<J2SE_HOME>\bin\keytool -export -alias tomcat-server
-storepass changeit -file server.cer -keystore server.key-
store
```

3. Generate the client keystore.

   To generate the client keystore, enter the following at a terminal window:

   Note that when you press Enter, `keytool` prompts you to enter the client's server name, organizational unit, organization, locality, state, and country code. Note that **you must enter the server name in response to keytool's first prompt in which it asks for first and last names**. In most cases, for testing purposes, this will be `localhost`. This host must match the host identified in the endpoint address specified in Step 4: Editing the Build Properties (page 739).

```
<J2SE_HOME>\bin\keytool -genkey -alias jwsdp-client -keyalg
RSA -keypass changeit -storepass changeit -keystore
client.keystore
```

4. Import the server certificate into the client's keystore.

```
<J2SE_HOME>\bin\keytool -import -v -trustcacerts
-alias tomcat-server -file server.cer
-keystore client.keystore -keypass changeit
-storepass changeit
```

Step 2: Configuring the SSL Connector

Note: The steps for configuring an SSL Connector are provided in more detail in the section Configuring the SSL Connector (page 729). The steps in this section are provided for your convenience.

You need to configure an SSL Connector for Tomcat. If you are using a different server, see Configuring the SSL Connector (page 729) for general information on configuring an SSL Connector. In addition to configuring the server for an SSL Connector, you must also add information on the keystore file and its password in the same place where you've added the SSL connector. For example, in the Java WSDP, you first need to remove the comment tags (<!-- ... -->) from around the SSL Connector and then add the information in **bold** to this section in the file `<JWSDP_HOME>/conf/server.xml`.

```
<!-- SSL Connector on Port 8443 -->
<Connector
className="org.apache.coyote.tomcat4.CoyoteConnector"
     port="8443" minProcessors="5" maxProcessors="75"
     enableLookups="false"
     acceptCount="10" debug="0" scheme="https" secure="true">
<Factory className=
"org.apache.coyote.tomcat4.CoyoteServerSocketFactory">
keystoreFile=
  "<JWSDP_HOME>/docs/tutorial/examples/jaxrpc/security/
       server.keystore"
     keystorePass="changeit"
     clientAuth="false" protocol="TLS" debug="0" />
</Connector>
```

Step 3: Adding Security Elements to web.xml

The files for this example are in the `<JWSDP_HOME>/docs/tutorial/examples/jaxrpc/security` directory. For authentication over SSL, the `web.xml` file includes the `<security-constraint>`, `<login-config>`, and

`<security-role>` elements. Code in **bold** has been added from the basic *<JWSDP_HOME>*`/docs/tutorial/examples/jaxrpc/hello` example.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
    2.3//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">

<web-app>
  <display-name>Hello World Application (secure)</display-name>
  <description>HTTPS example using JAX-RPC </description>
  <session-config>
    <session-timeout>60</session-timeout>
  </session-config>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>SecureHello</web-resource-name>
      <url-pattern>/security</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
      <role-name>manager</role-name>
    </auth-constraint>
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
  <login-config>
    <auth-method>BASIC</auth-method>
  </login-config>
  <security-role>
    <role-name>manager</role-name>
  </security-role>
</web-app>
```

Note that the `<role-name>` element specifies `manager`, a role that has already been specified in a deployment descriptor for Tomcat (*<JWSDP_HOME>*`/conf/tomcat-users.xml`). For more information on defining and linking roles, see Security Roles (page 703).

Step 4: Editing the Build Properties

To run the application with basic authentication over HTTP/SSL, we have added some properties related to the keystore file and its password to the `build.properties` file, which is located in the *<JWSDP_HOME>*`/docs/tutorial/examples/jaxrpc/security` directory. The following example assumes you are running on Java WSDP 1.1. The items marked in **bold** have been be added to the file. The items that have been added will be passed as arguments to the client application when it is run in a later section.

```
# This file is referenced by the build.xml file.

example=security
context-path=security-jaxrpc

client-class=security.HelloClient
client-jar=${example}-client.jar

portable-war=${example}-portable.war
deployable-war=${context-path}.war
war-path=${tut-
root}/tutorial/examples/jaxrpc/${example}/dist/${deployable-
war}

trust-store=${tut-root}/tutorial/examples/jaxrpc/security/cli-
ent.keystore

trust-store-password=changeit
```

Step 5: Setting Security Properties in the Client Code

The source code for the client is in the `HelloClient.java` file of the *<JWSDP_HOME>*`/docs/tutorial/examples/jaxrpc/security` directory. For basic authentication over SSL, the client code must set several security-related properties.

- `trust-store` Property - The value of the `trust-store` property is the fully qualified name of the client keystore file:

<*JWSDP_HOME*>/docs/tutorial/examples/jaxrpc/security/client.keystore

- `trust-store-password` Property - The `trust-store-password` property is the password of the keystore. The default value of this password is `changeit`.
- `username` and `password` Properties - The username and password properties correspond to the `manager` role. (See Security Roles, page 703.)

The client sets the aforementioned security properties as follows. The code in **bold** is the code that had been added from the original version of the `jaxrpc/hello` example application.

```
package security;

import javax.xml.rpc.Stub;

public class HelloClient {

    public static void main(String[] args) {

        if (args.length !=4) {
        System.out.println("Usage: ant run-security");
        System.exit(1);
        }

        String trustStore=args[0];
        String trustStorePassword=args[1];
        String username=args[2];
        String password=args[3];

        System.out.println("trustStore: " + trustStore);
        System.out.println("trustStorePassword: " +
          trustStorePassword);
        System.out.println("username: " + username);
        System.out.println("password: " + password);

     try {
        Stub stub = createProxy();
        System.setProperty("javax.net.ssl.trustStore",
          trustStore);
        System.setProperty("javax.net.ssl.trustStorePassword",
          trustStorePassword);
     stub._setProperty(javax.xml.rpc.Stub.USERNAME_PROPERTY,
          username);
     stub._setProperty(javax.xml.rpc.Stub.PASSWORD_PROPERTY,
          password);
          HelloIF hello = (HelloIF)stub;
          System.out.println(hello.sayHello("Duke! I feel
             secure!"));
           } catch (Exception ex) {
               ex.printStackTrace();
           }
      }

      private static Stub createProxy() {
         // Note: MyHelloService_Impl is implementation-specific.
          return (Stub)(new MyHello_Impl().getHelloIFPort());
      }
   }
```

Step 6: Create a New Ant Target for Running this Example

The existing target for running the hello example application is not sufficient for running the secure version of the application. You need to pass information on the keystore and its password, as well as the user name and its password. The following target has been added to the file *<JWSDP_HOME>*/docs/examples/jaxrpc/security/build.xml to faciliate running the secure JAX-RPC example:

```
<target name="run-security"
     description="Runs a client with authentication
     over ssl">
     <echo message="Running the ${client-class} program...." />
     <java
        fork="on"
        classpath="${dist}/${client-jar}:${jwsdp-jars}"
        classname="${client-class}" >
        <arg value="${trust-store}" />
        <arg value="${trust-store-password}" />
        <arg value="${username}" />
        <arg value="${password}" />
     </java>
</target>
```

Step 7: Building and Running this Example

To build and run this JAX-RPC example over SSL, perform the following steps:

1. If you haven't already done so, follow the instructions in Setting Up (page 69), download the example code for this tutorial, and complete Steps 1-2 before proceeding, as these steps are specific to your machine and implementation.
2. Make sure that Tomcat is running.
3. Go to the *<JWSDP_HOME>*/docs/tutorial/examples/jaxrpc/security directory.
4. Type the following commands:

```
ant build
ant package
ant deploy
ant build-static
ant run-security
```

The client should display the following output:

```
% ant run-security
Buildfile: build.xml

run-security:
     [echo] Running the security.HelloClient program...
     [java] trustStore: <JWSDP_HOME>/docs/tutorial/examples/
     jaxrpc/security/client.keystore
     [java] trustStorePassword: changeit
     [java] username: your_name
     [java] password: your_password
[java] Hello - secure Duke! I feel secure!

BUILD SUCCESSFUL
```

Enabling Mutual Authentication Over SSL

The section Security for JAX-RPC (page 734) discusses setting up server-side authentication. This section discusses setting up client-side authentication. When both server and client-side authentication are enabled, this is called mutual, or two-way, authentication. In client authentication, clients are required to submit certificates that are issued by a certificate authority that you choose to accept. There are at least two ways to enable client authentication.

1. Configure the SSL Socket Factory to enable client authentication. For example, to configure the SSL Socket Factory for Tomcat, you would set `clientAuth="true"`, as shown in **bold** in the code sample below. By enabling client authentication in this way, client authentication is required for all the requests going through the specified SSL port. As with all changes to the Web server configuration file, you must stop and restart the Web server for this change to become effective.

```
<!-- SSL Connector on Port 8443 -->
<Connector   className="org.apache.coyote.tomcat4.CoyoteCon-
nector"
     port="8443" minProcessors="5" maxProcessors="75"
     enableLookups="false"
     acceptCount="10"        debug="0"        scheme="https"
secure="true">
<Factory className=
"org.apache.coyote.tomcat4.CoyoteServerSocketFactory">
keystoreFile=
     "<JWSDP_HOME>/docs/tutorial/examples/jaxrpc/security/
     server.keystore"
     keystorePass="changeit"
     clientAuth="true" protocol="TLS" debug="0" />
</Connector>
```

2. Set the method of authentication in the `web.xml` file to `CLIENT-CERT`, as shown in **bold** below. By enabling client authentication in this way, client authentication is enabled for a specific application.

```
<login-config>
     <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

3. When client authentication is enabled in both ways mentioned above, client authentication will be performed twice.

Configuring Mutual Authentication for the JAX-RPC Security Example

To configure and create a JAX-RPC service with mutual authentication, follow all of the steps in the section Security for JAX-RPC (page 734) up to and including the command `ant build-static`. Then, follow these steps:

1. Generate a client certificate, export it, and then import the client certificate into the server's keystore, as discussed in Creating a Client Certificate for Mutual Authentication (page 727).
2. Edit `web.xml` to change the method of authentication to `CLIENT-CERT` in the login configuration section of the deployment descriptor.
   ```
   <login-config>
        <auth-method>CLIENT-CERT</auth-method>
   </login-config>
   ```
3. Run the application:
   ```
   ant run-security
   ```

The client should display the following line:

```
Hello Duke! I feel secure!
```

Acknowledgement: This section includes material from the "Web Services Security Configuration" white paper, written by Rahul Sharma and Beth Stearns.

EIS-Tier Security

In the EIS tier, an application component requests a connection to an EIS resource. As part of this connection, the EIS may require a sign-on to the resource. The application component provider has two choices for the design of the EIS sign-on:

- With the container-managed sign-on approach, the application component lets the container take the responsibility of configuring and managing the EIS sign-on. The container determines the user name and password for establishing a connection to an EIS instance.
- With the component-managed sign-on approach, the application component code manages EIS sign-on by including code that performs the sign-on process to an EIS.

Container-Managed Sign-On

With container-managed sign-on, an application component does not have to pass any security information for signing on to the resource to the `getConnection()` method. The security information is supplied by the container, as shown in the following example.

```
// Business method in an application component
Context initctx = new InitialContext();

// Perform JNDI lookup to obtain a connection factory
javax.resource.cci.ConnectionFactory cxf =
     (javax.resource.cci.ConnectionFactory)initctx.lookup(
      “java:comp/env/eis/MainframeCxFactory”);

// Invoke factory to obtain a connection. The security
// information is not passed in the getConnection method
javax.resource.cci.Connection cx = cxf.getConnection();
...
```

Component-Managed Sign-On

With component-managed sign-on, an application component is responsible for passing the security information that is needed for signing on to the resource to the `getConnection()` method. Security information could be a user name and password, for example, as shown here:

```
// Method in an application component
Context initctx = new InitialContext();

// Perform JNDI lookup to obtain a connection factory
javax.resource.cci.ConnectionFactory cxf =
     (javax.resource.cci.ConnectionFactory)initctx.lookup(
       “java:comp/env/eis/MainframeCxFactory”);

// Get a new ConnectionSpec
com.myeis.ConnectionSpecImpl properties = //..

// Invoke factory to obtain a connection
properties.setUserName(“...”);
properties.setPassword(“...”);
javax.resource.cci.Connection cx =
  cxf.getConnection(properties);
...
```

19 The Coffee Break Application

Stephanie Bodoff, Maydene Fisher, Dale Green, Kim Haase

The introduction to this tutorial introduced a scenario in which a Web application (The Coffee Break) is constructed using Web services. Now that we have discussed all the technologies necessary to build Web applications and Web services, this chapter describes an implementation of the scenario described in Chapter 1.

Coffee Break Overview

The Coffee Break sells coffee on the Internet. Customers communicate with the Coffee Break server to order coffee online. The server consists of Java Servlets, JSP pages, and JavaBeans components. A customer enters the quantity of each coffee to order and clicks the “Submit” button to send the order.

The Coffee Break does not maintain any inventory. It handles customer and order management and billing. Each order is filled by forwarding suborders to one or more coffee distributors. This process is depicted in Figure 19–1.

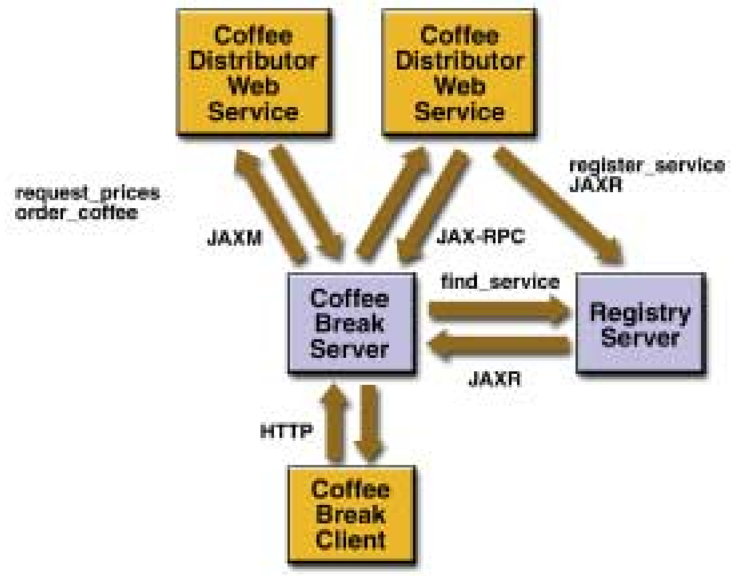


Figure 19–1 Coffee Break Application Flow

The Coffee Break server obtains the coffee varieties it sells and their prices by querying distributors at startup and on demand.

1. The Coffee Break server uses JAXM messaging to communicate with one of its distributors. It has been dealing with this distributor for some time and has previously made the necessary arrangements for doing request-response JAXM messaging. The two parties have agreed to exchange four kinds of XML messages and have set up the DTDs those messages will follow.
2. The Coffee Break server uses JAXR to send a query searching for coffee distributors that support JAX-RPC to the Registry Server.
3. The Coffee Break server requests price lists from each of the coffee distributors. The server makes the appropriate remote procedure calls and waits for the response, which is a JavaBeans component representing a price list. The JAXM distributor returns price lists as XML documents.
4. Upon receiving the responses, the Coffee Break server processes the price lists from the JavaBeans components returned by calls to the distributors.
5. The Coffee Break Server creates a local database of distributors.
6. When an order is placed, suborders are sent to one or more distributors using the distributor's preferred protocol.

JAX-RPC Distributor Service

The Coffee Break server is also a client— it makes remote calls on the JAX-RPC distributor service. The service code consists of the service interface, service implementation class, and several JavaBeans components that are used for method parameters and return types.

Service Interface

The service interface, `SupplierIF`, defines the methods that can be called by remote clients. The parameters and return types of these methods are JavaBeans components:

- `AddressBean` - shipping information for customer
- `ConfirmationBean` - order id and ship date
- `CustomerBean` - customer contact information
- `LineItemBean` - order item
- `OrderBean` - order id, customer, address, list of line items, total price
- `PriceItemBean` - price list entry (coffee name and wholesale price)
- `PriceListBean` - price list

Because these components are shared by other programs, their source code resides in the <*JWSDP_HOME*>`/docs/tutorial/examples/cb/common/src` directory. The source code for the `SupplierIF` interface, which follows, resides in the <*JWSDP_HOME*>`/docs/tutorial/examples/cb/jaxrpc/src` directory.

```
package com.sun.cb;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface SupplierIF extends Remote {

    public ConfirmationBean placeOrder(OrderBean order)
        throws RemoteException;
   public PriceListBean getPriceList() throws RemoteException;
}
```

Service Implementation

The SupplierImpl class implements the placeOrder and getPriceList methods, which are defined by the SupplierIF interface. So that you can focus on the code related to JAX-RPC, these methods are short and simplistic. In a real-world application, these methods would access databases and interact with other services, such as shipping, accounting, and inventory.

The placeOrder method accepts as input a coffee order and returns a confirmation for the order. To keep things simple, the placeOrder method confirms every order and sets the ship date in the confirmation to the next day. (This date is calculated by DateHelper, a utility class that resides in the cb/common subdirectory.) The source code for the placeOrder method follows:

```
public ConfirmationBean placeOrder(OrderBean order) {

    Date tomorrow =
        com.sun.cb.DateHelper.addDays(new Date(), 1);
    ConfirmationBean confirmation =
        new ConfirmationBean(order.getId(), tomorrow);
    return confirmation;
}
```

The getPriceList method returns a PriceListBean object, which lists the name and price of each type of coffee that can be ordered from this service. The getPriceList method creates the PriceListBean object by invoking a private method named loadPrices. In a production application, the loadPrices method would fetch the prices from a database. However, our loadPrices method takes a shortcut by getting the prices from the SupplierPrices.properties file. Here are the getPriceList and loadPrices methods:

```
public PriceListBean getPriceList() {

   PriceListBean priceList = loadPrices();
   return priceList;
}

private PriceListBean loadPrices() {

   String propsName = “com.sun.cb.SupplierPrices”;
   Date today = new Date();
   Date endDate = DateHelper.addDays(today, 30);

   PriceItemBean[] priceItems =
       PriceLoader.loadItems(propsName);
```

```
        PriceListBean priceList =
            new PriceListBean(today, endDate, priceItems);

        return priceList;
    }
```

Publishing the Service in the Registry

Because we want customers to find our service, we will to publish it in a registry. The programs that publish and remove our service are called `OrgPublisher` and `OrgRemover`. These programs are not part of the service's Web application. They are stand-alone programs that are run by the `ant set-up-service` command. (See Building and Installing the JAX-RPC Service, page 778.) Immediately after the service is installed, it's published in the registry. And in like manner, right before the service is removed, it's removed from the registry.

The `OrgPublisher` program begins by loading `String` values from the `CoffeeRegistry.properties` file. Next, the program instantiates a utility class named `JAXRPublisher`. `OrgPublisher` connects to the registry by invoking the `makeConnection` method of `JAXRPublisher`. To publish the service, `OrgPublisher` invokes the `executePublish` method, which accepts as input `username`, `password`, and `endpoint`. The `username` and `password` values are required by the Registry Server. The `endpoint` value is the URL that remote clients will use to contact our JAX-RPC service. The `executePublish` method of `JAXRPublisher` returns a key that uniquely identifies the service in the registry. `OrgPublisher` saves this key in a text file named `orgkey.txt`. The `OrgRemover` program will read the key from `orgkey.txt` so that it can delete the service. (See Deleting the Service From the Registry, page 755.) The source code for the `OrgPublisher` program follows.

```
package com.sun.cb;

import javax.xml.registry.*;
import java.util.ResourceBundle;
import java.io.*;

public class OrgPublisher {

    public static void main(String[] args) {

        ResourceBundle registryBundle =
           ResourceBundle.getBundle
           ("com.sun.cb.CoffeeRegistry");
```

```
            String queryURL =
                registryBundle.getString("query.url");
            String publishURL =
                registryBundle.getString("publish.url");
            String username =
                registryBundle.getString("registry.username");
            String password =
                registryBundle.getString("registry.password");
           String endpoint = registryBundle.getString("endpoint");
           String keyFile = registryBundle.getString("key.file");

            JAXRPublisher publisher = new JAXRPublisher();
            publisher.makeConnection(queryURL, publishURL);
            String key = publisher.executePublish
                (username, password, endpoint);

            try {
                FileWriter out = new FileWriter(keyFile);
                out.write(key);
                out.flush();
                out.close();
            } catch (IOException ex) {
                System.out.println(ex.getMessage());
            }
        }

    }
```

The `JAXRPublisher` class is almost identical to the sample program `JAXRPublish.java`, which is described in Managing Registry Data (page 551).

First, the `makeConnection` method creates a connection to the Registry Server. See Establishing a Connection (page 542) for more information. To do this, it first specifies a set of connection properties using the query and publish URLs passed in from the `CoffeeRegistry.properties` file. For the Registry Server, the query and publish URLs are actually the same.

```
Properties props = new Properties();
props.setProperty("javax.xml.registry.queryManagerURL",
    queryUrl);
props.setProperty("javax.xml.registry.lifeCycleManagerURL",
    publishUrl);
```

Next, the makeConnection method creates the connection, using the connection properties:

```
    ConnectionFactory factory = ConnectionFactory.newInstance();
    factory.setProperties(props);
    connection = factory.createConnection();
```

The executePublish method takes three arguments: a username, a password, and an endpoint. It begins by obtaining a RegistryService object, then a BusinessQueryManager object and a BusinessLifeCycleManager object, which enable it to perform queries and manage data:

```
    rs = connection.getRegistryService();
    blcm = rs.getBusinessLifeCycleManager();
    bqm = rs.getBusinessQueryManager();
```

Because it needs password authentication in order to publish data, it then uses the username and password arguments to establish its security credentials:

```
    PasswordAuthentication passwdAuth =
        new PasswordAuthentication(username,
            password.toCharArray());
    Set creds = new HashSet();
    creds.add(passwdAuth);
    connection.setCredentials(creds);
```

It then creates an Organization object with the name "JAXRPCCoffeeDistributor," then a User object that will serve as the primary contact. It gets the data from the resource bundle instead of hardcoding it as strings, but otherwise this code is almost identical to that shown in the JAXR chapter.

```
    ResourceBundle bundle =
        ResourceBundle.getBundle("com.sun.cb.CoffeeRegistry");

    // Create organization name and description
    Organization org =
        blcm.createOrganization(bundle.getString("org.name"));
    InternationalString s =
        blcm.createInternationalString
        (bundle.getString("org.description"));
    org.setDescription(s);

    // Create primary contact, set name
```

```
User primaryContact = blcm.createUser();
PersonName pName =
    blcm.createPersonName(bundle.getString("person.name"));
primaryContact.setPersonName(pName);
```

It adds a telephone number and email address for the user, then makes the user the primary contact:

```
org.setPrimaryContact(primaryContact);
```

It gives JAXRPCCoffeeDistributor a classification using the North American Industry Classification System (NAICS). In this case it uses the classification “Other Grocery and Related Products Wholesalers”.

```
Classification classification = (Classification)
    blcm.createClassification(cScheme,
        bundle.getString("classification.name"),
        bundle.getString("classification.value"));
Collection classifications = new ArrayList();
classifications.add(classification);
org.addClassifications(classifications);
```

Next, it adds the JAX-RPC service, called “JAXRPCCoffee Service,” and its service binding. The access URI for the service binding contains the endpoint URL that remote clients will use to contact our service:

```
http://localhost:8080/jaxrpc-coffee-supplier/jaxrpc/SupplierIF
```

JAXR validates each URI, so an exception is thrown if the service was not installed before you ran this program.

```
Collection services = new ArrayList();
Service service =
    blcm.createService(bundle.getString("service.name"));
InternationalString is =
    blcm.createInternationalString
    (bundle.getString("service.description"));
service.setDescription(is);

// Create service bindings
Collection serviceBindings = new ArrayList();
ServiceBinding binding = blcm.createServiceBinding();
is = blcm.createInternationalString
    (bundle.getString("service.binding"));
binding.setDescription(is);
try {
        binding.setAccessURI(endpoint);
    } catch (JAXRException je) {
        throw new JAXRException("Error: Publishing this " +
        "service in the registry has failed because " +
        "the service has not been installed on Tomcat.");
    }
    serviceBindings.add(binding);

    // Add service bindings to service
    service.addServiceBindings(serviceBindings);

    // Add service to services, then add services to organization
    services.add(service);
    org.addServices(services);
```

Then it saves the organization to the registry:

```
    Collection orgs = new ArrayList();
    orgs.add(org);
    BulkResponse response = blcm.saveOrganizations(orgs);
```

The `BulkResponse` object returned by `saveOrganizations` includes the `Key` object containing the unique key value for the organization. The `executePublish` method first checks to make sure the `saveOrganizations` call succeeded.

If the call succeeded, the method extracts the value from the `Key` object and displays it:

```
    Collection keys = response.getCollection();
    Iterator keyIter = keys.iterator();
    if (keyIter.hasNext()) {
        javax.xml.registry.infomodel.Key orgKey =
            (javax.xml.registry.infomodel.Key) keyIter.next();
        id = orgKey.getId();
        System.out.println("Organization key is " + id);
    }
```

Finally, the method returns the string `id` so that the `OrgPublisher` program can save it in a file for use by the `OrgRemover` program.

Deleting the Service From the Registry

The `OrgRemover` program deletes the service from the Registry Server immediately before the service is removed. Like the `OrgPublisher` program, the `OrgRemover` program starts by fetching values from the `CoffeeRegistry.prop-`

erties file. One these values, keyFile, is the name of the file that contains the key that uniquely identifies the service. OrgPublisher reads the key from the file, connects to the Registry Server by invoking makeConnection, and then deletes the service from the registry by calling executeRemove. Here is the source code for the OrgRemover program:

```
package com.sun.cb;

import java.util.ResourceBundle;
import javax.xml.registry.*;
import javax.xml.registry.infomodel.Key;
import java.io.*;

public class OrgRemover {

    Connection connection = null;

    public static void main(String[] args) {

        String keyStr = null;

        ResourceBundle registryBundle =
            ResourceBundle.getBundle
            (“com.sun.cb.CoffeeRegistry”);

        String queryURL =
            registryBundle.getString(“query.url”);
        String publishURL =
            registryBundle.getString(“publish.url”);
        String username =
            registryBundle.getString(“registry.username”);
        String password =
            registryBundle.getString(“registry.password”);
        String keyFile = registryBundle.getString(“key.file”);

        try {
            FileReader in = new FileReader(keyFile);
            char[] buf = new char[512];
            while (in.read(buf, 0, 512) >= 0) { }
            in.close();
            keyStr = new String(buf).trim();
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }

        JAXRRemover remover = new JAXRRemover();
        remover.makeConnection(queryURL, publishURL);
```

```
            javax.xml.registry.infomodel.Key modelKey = null;
            modelKey = remover.createOrgKey(keyStr);
            remover.executeRemove(modelKey, username, password);
        }
    }
```

Instantiated by the OrgRemover program, the JAXRRemover class contains the makeConnection, createOrgKey, and executeRemove methods. It is almost identical to the sample program JAXRDelete.java, which is described in Removing Data from the Registry (page 556).

The makeConnection method is identical to the JAXRPublisher method of the same name.

The createOrgKey method is a utility method that takes one argument, the string value extracted from the key file. It obtains the RegistryService object and the BusinessLifeCycleManager object, then creates a Key object from the string value.

The executeRemove method takes three arguments: a username, a password, and the Key object returned by the createOrgKey method. It uses the username and password arguments to establish its security credentials with the Registry Server, just as the executePublish method does.

The method then wraps the Key object in a Collection and uses the Business-LifeCycleManager object's deleteOrganizations method to delete the organization.

```
    Collection keys = new ArrayList();
    keys.add(key);
    BulkResponse response = blcm.deleteOrganizations(keys);
```

The deleteOrganizations method returns the keys of the organizations it deleted, so the executeRemove method then verifies that the correct operation was performed and displays the key for the deleted organization.

```
    Collection retKeys = response.getCollection();
    Iterator keyIter = retKeys.iterator();
    javax.xml.registry.infomodel.Key orgKey = null;
    if (keyIter.hasNext()) {
        orgKey =
            (javax.xml.registry.infomodel.Key) keyIter.next();
        id = orgKey.getId();
        System.out.println("Organization key was " + id);
    }
```

JAXM Distributor Service

The JAXM distributor service is simply the arrangements that the distributor and the Coffee Break have made regarding their exchange of XML documents. These arrangements include what kinds of messages they will send, the form of those messages, and what kind of JAXM messaging they will do. If they had agreed to do one-way messaging, they would also have had to use messaging providers that talk to each other and had to use the same profile. In this scenario, the parties have agreed to use request-response messaging, so a messaging provider is not needed.

The Coffee Break server sends two kinds of messages:

- Requests for current wholesale coffee prices
- Customer orders for coffee

The JAXM coffee supplier responds with two kinds of messages:

- Current price lists
- Order confirmations

All of the messages they send conform to an agreed-upon XML structure, which is specified in a DTD for each kind of message. This allows them to exchange messages even though they use different document formats internally.

The four kinds of messages exchanged by the Coffee Break server and the JAXM distributor are specified by the following DTDs:

- `request-prices.dtd`
- `price-list.dtd`
- `coffee-order.dtd`
- `confirm.dtd`

These DTDs may be found at

```
<JWSDP_HOME>/docs/tutorial/examples/cb/jaxm/dtds
```

The `dtds` directory also contains a sample of what the XML documents specified in the DTDs might look like. The corresponding XML files for each of the DTDs are as follows:

- `request-prices.xml`
- `price-list.xml`
- `coffee-order.xml`
- `confirm.xml`

Because of the DTDs, both parties know ahead of time what to expect in a particular kind of message and can therefore extract its content using the JAXM API.

Code for the client and server applications is in the following directory:

```
<JWSDP_HOME>/docs/tutorial/examples/cb/jaxm/src/com/sun/cb
```

JAXM Client

The Coffee Break server, which is the JAXM client in this scenario, sends requests to its JAXM distributor. Because the request-response form of JAXM messaging is being used, the client applications use the `SOAPConnection` method `call` to send messages.

```
SOAPMessage response = con.call(request, endpoint);
```

Accordingly, the client code has two major tasks. The first is to create and send the request; the second is to extract the content from the response. These tasks are handled by the classes `PriceListRequest` and `OrderRequest`.

Sending the Request

This section covers the code for creating and sending the request for an updated price list. This is done in the `getPriceList` method of `PriceListRequest`, which follows the DTD `price-list.dtd`.

The `getPriceList` method begins by creating the connection that will be used to send the request. Then it gets the default `MessageFactory` object so that it can create the `SOAPMessage` object `msg`.

```
SOAPConnectionFactory scf =
            SOAPConnectionFactory.newInstance();
SOAPConnection con = scf.createConnection();

MessageFactory mf = MessageFactory.newInstance();
SOAPMessage msg = mf.createMessage();
```

The next step is to access the message's `SOAPEnvelope` object, which will be used to create a `Name` object for each new element that is created. It is also used to access the `SOAPBody` object, to which the message's content will be added.

```
SOAPPart part = msg.getSOAPPart();
SOAPEnvelope envelope = part.getEnvelope();
SOAPBody body = envelope.getBody();
```

The file `price-list.dtd` specifies that the top-most element inside the body is `request-prices` and that it contains the element `request`. The text node added to `request` is the text of the request being sent. Every new element that is added to the message must have a `Name` object to identify it, which is created by the `Envelope` method `createName`. The following lines of code create the top-level element in the `SOAPBody` object `body`. The first element created in a `SOAPBody` object is always a `SOAPBodyElement` object.

```
Name bodyName = envelope.createName("request-prices",
       "RequestPrices", "http://sonata.coffeebreak.com");
SOAPBodyElement requestPrices =
                    body.addBodyElement(bodyName);
```

In the next few lines, the code adds the element `request` to the element `request-prices` (represented by the `SOAPBodyElement` `requestPrices`.) Then the code adds a text node containing the text of the request. Next, because there are no other elements in the request, the code calls the method `saveChanges` on the message to save what has been done.

```
Name requestName = envelope.createName("request");
SOAPElement request =
               requestPrices.addChildElement(requestName);
request.addTextNode("Send updated price list.");

msg.saveChanges();
```

With the creation of the request message completed, the code sends the message to the JAXM coffee supplier. The message being sent is the `SOAPMessage` object `msg`, to which the elements created in the previous code snippets were added. The endpoint is the URI for the JAXM coffee supplier. The `SOAPConnection` object con is used to send the message, and because it is no longer needed, it is closed.

```
URL endpoint = new URL(
 "http://localhost:8080/jaxm-coffee-supplier/getPriceList");
SOAPMessage response = con.call(msg, endpoint);
con.close();
```

When the call method is executed, Tomcat executes the servlet PriceList-Servlet. This servlet creates and returns a SOAPMessage object whose content is the JAXM distributor's price list. (PriceListServlet is discussed in Returning the Price List, page 767.) Tomcat knows to execute PriceListServlet because the web.xml file at <*JWSDP*>/docs/tutorial/examples/cb/jaxm/web/ maps the given endpoint to that servlet.

Extracting the Price List

This section demonstrates (1) retrieving the price list that is contained in response, the SOAPMessage object returned by the method call, and (2) returning the price list as a PriceListBean.

The code creates an empty Vector object that will hold the coffee-name and price elements that are extracted from response. Then the code uses response to access its SOAPBody object, which holds the message's content. Notice that the SOAPEnvelope object is not accessed separately because it is not needed for creating Name objects, as it was in the previous section.

```
Vector list = new Vector();

SOAPBody responseBody = response.getSOAPPart().
                          getEnvelope().getBody();
```

The next step is to retrieve the SOAPBodyElement object. The method getChildElements returns an Iterator object that contains all of the child elements of the element on which it is called, so in the following lines of code, it1 contains the SOAPBodyElement object bodyEl, which represents the price-list element.

```
Iterator it1 = responseBody.getChildElements();
while (it1.hasNext()) {
  SOAPBodyElement bodyEl = (SOAPBodyElement)it1.next();
```

The Iterator object it2 holds the child elements of bodyEl, which represent coffee elements. Calling the method next on it2 retrieves the first coffee element in bodyEl. As long as it2 has another element, the method next will return the next coffee element.

```
      Iterator it2 = bodyEl.getChildElements();
      while (it2.hasNext()) {
         SOAPElement child2 = (SOAPElement)it2.next();
```

The next lines of code drill down another level to retrieve the coffee-name and price elements contained in it3. Then the message getValue retrieves the text (a coffee name or a price) that the JAXM coffee distributor added to the coffee-name and price elements when it gave content to response. The final line in the following code fragment adds the coffee name or price to the Vector object list. Note that because of the nested while loops, for each coffee element that the code retrieves, both of its child elements (the coffee-name and price elements) are retrieved.

```
         Iterator it3 = child2.getChildElements();
         while (it3.hasNext()) {
          SOAPElement child3 = (SOAPElement)it3.next();
          String value = child3.getValue();
          list.addElement(value);
         }
      }
   }
```

The last code fragment adds the coffee names and their prices (as a PriceListItem) to the ArrayList priceItems, and prints each pair on a separate line. Finally it constructs and returns a PriceListBean.

```
   ArrayList priceItems = new ArrayList();

   for (int i = 0; i < list.size(); i = i + 2) {
      priceItems.add(
           new PriceItemBean(list.elementAt(i).toString(),
           new BigDecimal(list.elementAt(i + 1).toString())));
      System.out.print(list.elementAt(i) + "        ");
      System.out.println(list.elementAt(i + 1));
   }

   Date today = new Date();
   Date endDate = DateHelper.addDays(today, 30);
   PriceListBean plb =
            new PriceListBean(today, endDate, priceItems);
```

Ordering Coffee

The other kind of message that the Coffee Break server can send to the JAXM distributor is an order for coffee. This is done in the `placeOrder` method of `OrderRequest`, which follows the DTD `coffee-order.dtd`.

Creating the Order

As with the client code for requesting a price list, the `placeOrder` method starts out by creating a `SOAPConnection` object, creating a `SOAPMessage` object, and accessing the message's `SOAPEnvelope` and `SOAPBody` objects.

```
SOAPConnectionFactory scf =
                       SOAPConnectionFactory.newInstance();
SOAPConnection con = scf.createConnection();
MessageFactory mf = MessageFactory.newInstance();
SOAPMessage msg = mf.createMessage();

SOAPPart part = msg.getSOAPPart();
SOAPEnvelope envelope = part.getEnvelope();
SOAPBody body = envelope.getBody();
```

Next the code creates and adds XML elements to form the order. As is required, the first element is a `SOAPBodyElement`, which in this case is `coffee-order`.

```
Name bodyName = envelope.createName("coffee-order", "PO",
                  "http://sonata.coffeebreak.com");
SOAPBodyElement order = body.addBodyElement(bodyName);
```

The application then adds the next level of elements, the first of these being `orderID`. The value given to `orderID` is extracted from the `OrderBean` object passed to the `OrderRequest.placeOrder` method.

```
Name orderIDName = envelope.createName("orderID");
SOAPElement orderID = order.addChildElement(orderIDName);
orderID.addTextNode(orderBean.getId());
```

The next element, `customer`, has several child elements that give information about the customer. This information is also extracted from the `Customer` component of `OrderBean`.

```
Name childName = envelope.createName("customer");
SOAPElement customer = order.addChildElement(childName);

childName = envelope.createName("last-name");
```

```
SOAPElement lastName = customer.addChildElement(childName);
lastName.addTextNode(orderBean.getCustomer().
  getLastName());

childName = envelope.createName("first-name");
SOAPElement firstName = customer.addChildElement(childName);
firstName.addTextNode(orderBean.getCustomer().
  getFirstName());

childName = envelope.createName("phone-number");
SOAPElement phoneNumber = customer.addChildElement(childName);
phoneNumber.addTextNode(orderBean.getCustomer().
  getPhoneNumber());

childName = envelope.createName("email-address");
SOAPElement emailAddress =
            customer.addChildElement(childName);
emailAddress.addTextNode(orderBean.getCustomer().
  getEmailAddress());
```

The `address` element, added next, has child elements for the street, city, state, and zip code. This information is extracted from the `Address` component of `OrderBean`.

```
childName = envelope.createName("address");
SOAPElement address = order.addChildElement(childName);

childName = envelope.createName("street");
SOAPElement street = address.addChildElement(childName);
street.addTextNode(orderBean.getAddress().getStreet());

childName = envelope.createName("city");
SOAPElement city = address.addChildElement(childName);
city.addTextNode(orderBean.getAddress().getCity());

childName = envelope.createName("state");
SOAPElement state = address.addChildElement(childName);
state.addTextNode(orderBean.getAddress().getState());

childName = envelope.createName("zip");
SOAPElement zip = address.addChildElement(childName);
zip.addTextNode(orderBean.getAddress().getZip());
```

The element `line-item` has three child elements: `coffeeName`, `pounds`, and `price`. This information is extracted from the `LineItems` list contained in `OrderBean`.

```
for (Iterator it = orderBean.getLineItems().iterator();
                                   it.hasNext(); ; ) {
  LineItemBean lib = (LineItemBean)it.next();

  childName = envelope.createName("line-item");
  SOAPElement lineItem =
       order.addChildElement(childName);

  childName = envelope.createName("coffeeName");
  SOAPElement coffeeName =
       lineItem.addChildElement(childName);
  coffeeName.addTextNode(lib.getCoffeeName());

  childName = envelope.createName("pounds");
  SOAPElement pounds =
       lineItem.addChildElement(childName);
  pounds.addTextNode(lib.getPounds().toString());

  childName = envelope.createName("price");
  SOAPElement price =
       lineItem.addChildElement(childName);
  price.addTextNode(lib.getPrice().toString());

}

  //total
  childName = envelope.createName("total");
  SOAPElement total =
       order.addChildElement(childName);
  total.addTextNode(orderBean.getTotal().toString());
}
```

With the order complete, the application sends the message and closes the connection.

```
URL endpoint = new URL(
  "http://localhost:8080/jaxm-coffee-supplier/orderCoffee");
SOAPMessage reply = con.call(msg, endpoint);
con.close();
```

Because the `web.xml` file maps the given endpoint to `ConfirmationServlet`, Tomcat executes that servlet (discussed in Returning the Order Confirmation, page 772) to create and return the `SOAPMessage` object `reply`.

Retrieving the Order Confirmation

The rest of the `placeOrder` method retrieves the information returned in *`reply`*. The client knows what elements are in it because they are specified in `confirm.dtd`. After accessing the `SOAPBody` object, the code retrieves the `confirmation` element and gets the text of the `orderID` and `ship-date` elements. Finally, it constructs and returns a `ConfirmationBean` with this information.

```
SOAPBody sBody = reply.getSOAPPart().getEnvelope().getBody();
Iterator bodyIt = sBody.getChildElements();
SOAPBodyElement sbEl = (SOAPBodyElement)bodyIt.next();
Iterator bodyIt2 = sbEl.getChildElements();

SOAPElement ID = (SOAPElement)bodyIt2.next();
String id = ID.getValue();

SOAPElement sDate = (SOAPElement)bodyIt2.next();
String shippingDate = sDate.getValue();

SimpleDateFormat df = new
          SimpleDateFormat("EEE MMM dd HH:mm:ss z yyyy");
Date date = df.parse(shippingDate);
ConfirmationBean cb = new ConfirmationBean(id, date);
```

JAXM Service

The JAXM coffee distributor, the JAXM server in this scenario, provides the response part of the request-response paradigm. When JAXM messaging is being used, the server code is a servlet. The core part of each servlet is made up of three `javax.servlet.HttpServlet` methods: `init`, `doPost`, and `onMessage`. The `init` and `doPost` methods set up the response message, and the `onMessage` method gives the message its content.

Returning the Price List

This section takes you through the servlet PriceListServlet. This servlet creates the message with the current price list that is returned to the method call, invoked in PriceListRequest.

Any servlet extends a javax.servlet class. Being part of a Web application, this servlet extends HttpServlet. It first creates a static MessageFactory object that will be used later to create the SOAPMessage object that is returned. Then it declares the MessageFactory object msgFactory, which will be used to create a SOAPMessage object that has the headers and content of the original request message.

```
public class PriceListServlet extends HttpServlet {
  static MessageFactory fac = null;
  static {
     try {
        fac = MessageFactory.newInstance();
     } catch (Exception ex) {
        ex.printStackTrace();
     }
  };

  MessageFactory msgFactory;
```

Every servlet has an init method. This init method initializes the servlet with the configuration information that Tomcat passed to it. Then it simply initializes msgFactory with the default implementation of the MessageFactory class.

```
public void init(ServletConfig servletConfig)
                       throws ServletException {
  super.init(servletConfig);
  try {
     // Initialize it to the default.
     msgFactory = MessageFactory.newInstance();
  } catch (SOAPException ex) {
     throw new ServletException(
        "Unable to create message factory" + ex.getMessage());
  }
}
```

The next method defined in PriceListServlet is doPost, which does the real work of the servlet by calling the onMessage method. (The onMessage method is discussed later in this section.) Tomcat passes the doPost method two arguments. The first argument, the HttpServletRequest object req, holds the content of the message sent in PriceListRequest. The doPost method gets the content from req and puts it in the SOAPMessage object msg so that it can pass it to the onMessage method. The second argument, the HttpServletResponse object resp, will hold the message generated by executing the method onMessage.

In the following code fragment, doPost calls the methods getHeaders and putHeaders, defined immediately after doPost, to read and write the headers in req. It then gets the content of req as a stream and passes the headers and the input stream to the method MessageFactory.createMessage. The result is that the SOAPMessage object msg contains the request for a price list. Note that in this case, msg does not have any headers because the message sent in PriceListRequest did not have any headers.

```
public void doPost( HttpServletRequest req, HttpServletResponse
      resp) throws ServletException, IOException {
  try {
    // Get all the headers from the HTTP request.
    MimeHeaders headers = getHeaders(req);

    // Get the body of the HTTP request.
    InputStream is = req.getInputStream();

    // Now internalize the contents of the HTTP request and
    // create a SOAPMessage
    SOAPMessage msg = msgFactory.createMessage(headers, is);
```

Next, the code declares the SOAPMessage object reply and populates it by calling the method onMessage.

```
    SOAPMessage reply = null;
    reply = onMessage(msg);
```

If reply has anything in it, its contents are saved, the status of resp is set to OK, and the headers and content of reply are written to resp. If reply is empty, the status of resp is set to indicate that there is no content.

```
    if (reply != null) {
    // Need to call saveChanges because we're going to use the
    // MimeHeaders to set HTTP response information. These
    // MimeHeaders are generated as part of the save.
```

```
            if (reply.saveRequired()) {
               reply.saveChanges();
            }

            resp.setStatus(HttpServletResponse.SC_OK);

            putHeaders(reply.getMimeHeaders(), resp);
            // Write out the message on the response stream.
            OutputStream os = resp.getOutputStream();
            reply.writeTo(os);
            os.flush();
         } else
            resp.setStatus(HttpServletResponse.SC_NO_CONTENT);

      } catch (Exception ex) {
            throw new ServletException( "JAXM POST failed " +
                              ex.getMessage());
      }
   }
```

The methods `getHeaders` and `putHeaders` are not standard methods in a servlet the way `init`, `doPost`, and `onMessage` are. The method `doPost` calls `getHeaders` and passes it the `HttpServletRequest` object `req` that Tomcat passed to it. It returns a `MimeHeaders` object populated with the headers from `req`.

```
   static MimeHeaders getHeaders(HttpServletRequest req) {

      Enumeration enum = req.getHeaderNames();
      MimeHeaders headers = new MimeHeaders();

      while (enum.hasMoreElements()) {
         String headerName = (String)enum.nextElement();
         String headerValue = req.getHeader(headerName);

         StringTokenizer values = new StringTokenizer(
                                    headerValue, ",");
         while (values.hasMoreTokens()) {
            headers.addHeader(headerName,
               values.nextToken().trim());
         }
      }

      return headers;
   }
```

The doPost method calls putHeaders and passes it the MimeHeaders object headers, which was returned by the method getHeaders. The method putHeaders writes the headers in headers to res, the second argument passed to it. The result is that res, the response that Tomcat will return to the method call, now contains the headers that were in the original request.

```
static void putHeaders(MimeHeaders headers,
                            HttpServletResponse res) {
   Iterator it = headers.getAllHeaders();
   while (it.hasNext()) {
      String[] values = headers.getHeader(header.getName());
      if (values.length == 1)
         res.setHeader(header.getName(),
         header.getValue());
      else {
         StringBuffer concat = new StringBuffer();
         int i = 0;
         while (i < values.length) {
            if (i != 0) concat.append(',');
            concat.append(values[i++]);
         }
         res.setHeader(header.getName(), concat.toString());
      }
   }
}
```

The method onMessage is the application code for responding to the message sent by PriceListRequest and internalized into msg. It uses the static MessageFactory object fac to create the SOAPMessage object message and then populates it with the distributor's current coffee prices.

The method doPost invokes onMessage and passes it msg. In this case, onMessage does not need to use msg because it simply creates a message containing the distributor's price list. The onMessage method in ConfirmationServlet (Returning the Order Confirmation, page 772), on the other hand, uses the message passed to it to get the order ID.

```
public SOAPMessage onMessage(SOAPMessage msg) {
   SOAPMessage message = null;
   try {
      message = fac.createMessage();

      SOAPPart part = message.getSOAPPart();
      SOAPEnvelope envelope = part.getEnvelope();
      SOAPBody body = envelope.getBody();
```

```
    Name bodyName = envelope.createName("price-list",
            "PriceList", "http://sonata.coffeebreak.com");
    SOAPBodyElement list = body.addBodyElement(bodyName);

    coffee Name coffeeN = envelope.createName("coffee");
    SOAPElement coffee = list.addChildElement(coffeeN);

    Name coffeeNm1 = envelope.createName("coffee-name");
    SOAPElement coffeeName =
                    coffee.addChildElement(coffeeNm1);
    coffeeName.addTextNode("Arabica");

    Name priceName1 = envelope.createName("price");
    SOAPElement price1 = coffee.addChildElement(priceName1);
    price1.addTextNode("4.50");

    Name coffeeNm2 = envelope.createName("coffee-name");
    SOAPElement coffeeName2 =
                         coffee.addChildElement(coffeeNm2);
    coffeeName2.addTextNode("Espresso");

    Name priceName2 = envelope.createName("price");
    SOAPElement price2 = coffee.addChildElement(priceName2);
    price2.addTextNode("5.00");

    Name coffeeNm3 = envelope.createName("coffee-name");
    SOAPElement coffeeName3 =
                         coffee.addChildElement(coffeeNm3);
    coffeeName3.addTextNode("Dorada");

    Name priceName3 = envelope.createName("price");
    SOAPElement price3 = coffee.addChildElement(priceName3);
    price3.addTextNode("6.00");

    Name coffeeNm4 = envelope.createName("coffee-name");
    SOAPElement coffeeName4 =
                         coffee.addChildElement(coffeeNm4);
    coffeeName4.addTextNode("House Blend");

    Name priceName4 = envelope.createName("price");
    SOAPElement price4 = coffee.addChildElement(priceName4);
    price4.addTextNode("5.00");

    message.saveChanges();

  } catch(Exception e) {
    e.printStackTrace();
```

```
      }
      return message;
      }
   }
```

Returning the Order Confirmation

`ConfirmationServlet` creates the confirmation message that is returned to the `call` method that is invoked in `OrderRequest`. It is very similar to the code in `PriceListServlet` except that instead of building a price list, its `onMessage` method builds a confirmation with the order number and shipping date.

The `onMessage` method for this servlet uses the `SOAPMessage` object passed to it by the `doPost` method to get the order number sent in `OrderRequest`. Then it builds a confirmation message with the order ID and shipping date. The shipping date is calculated as today's date plus two days.

```
   public SOAPMessage onMessage(SOAPMessage message) {

      SOAPMessage confirmation = null;

      try {

         //retrieve the orderID elementfrom the message received
         SOAPBody sentSB = message.getSOAPPart().
                              getEnvelope().getBody();
         Iterator sentIt = sentSB.getChildElements();
         SOAPBodyElement sentSBE =
                           (SOAPBodyElement)sentIt.next();
         Iterator sentIt2 = sentSBE.getChildElements();
         SOAPElement sentSE = (SOAPElement)sentIt2.next();

         //get the text for orderID to put in confirmation
         String sentID = sentSE.getValue();

         //create the confirmation message
         confirmation = fac.createMessage();
         SOAPPart sp = confirmation.getSOAPPart();
         SOAPEnvelope env = sp.getEnvelope();
         SOAPBody sb = env.getBody();
         Name newBodyName = env.createName("confirmation",
                "Confirm", "http://sonata.coffeebreak.com");
         SOAPBodyElement confirm =
                sb.addBodyElement(newBodyName);

         //create the orderID element for confirmation
```

```
        Name newOrderIDName = env.createName("orderId");
        SOAPElement newOrderNo =
                    confirm.addChildElement(newOrderIDName);
        newOrderNo.addTextNode(sentID);

        //create ship-date element
        Name shipDateName = env.createName("ship-date");
        SOAPElement shipDate =
                  confirm.addChildElement(shipDateName);

        //create the shipping date
        Date today = new Date();
        long msPerDay = 1000 * 60 * 60 * 24;
        long msTarget = today.getTime();
        long msSum = msTarget + (msPerDay * 2);
        Date result = new Date();
        result.setTime(msSum);
        String sd = result.toString();
        shipDate.addTextNode(sd);

        confirmation.saveChanges();

      } catch (Exception ex) {
        ex.printStackTrace();
      }
      return confirmation;
    }
```

Coffee Break Server

The Coffee Break Server uses servlets, JSP pages, and JavaBeans components to dynamically construct HTML pages for consumption by a Web browser client. The JSP pages use the template tag library discussed in A Template Tag Library (page 665) to achieve a common look and feel among the HTML pages, and many of the JSTL custom tags discussed in Chapter 17 to minimize the use of scripting.

The Coffee Break Server implementation is organized along the Model-View-Controller design pattern. The `Dispatcher` servlet is the controller. It examines the request URL, creates and initializes model JavaBeans components, and dispatches requests to view JSP pages. The JavaBeans components contain the business logic for the application—they call the Web services and perform computations on the data returned from the services. The JSP pages format the data stored in the JavaBeans components. The mapping between JavaBeans components and pages is summarized in Table 19–1.

Table 19–1 Model and View Components

Function	JSP Page	JavaBeans Component
Update order data	`orderForm`	`ShoppingCart`
Update delivery and billing data	`checkoutForm`	`CheckoutFormBean`
Display order confirmation	`checkoutAck`	`OrderConfirmations`

JSP Pages

orderForm

`orderForm` displays the current contents of the shopping cart. The first time the page is requested, the quantities of all the coffees are 0. Each time the customer changes the coffees amounts and clicks the Update button, the request is posted back to `orderForm`. The `Dispatcher` servlet updates the values in the shopping cart, which are then redisplayed by orderForm. When the order is complete, the customer proceeds to the `checkoutForm` page by clicking the Checkout link.

checkoutForm

`checkoutForm` is used to collect delivery and billing information for the customer. When the Submit button is clicked, the request is posted to the `checkoutAck` page. However, the request is first handled by the `Dispatcher`, which invokes the `validate` method of `checkoutFormBean` If the validation does not succeed, the requested page is reset to `checkoutForm`, with error notifications in each invalid field. If the validation succeeds, `checkoutFormBean` submits suborders to each distributor and stores the result in the request-scoped OrderConfirmations JavaBeans component and control is passed to `checkoutAck`.

checkoutAck

checkoutAck simply displays the contents of the OrderConfirmations JavaBeans component, which is a list of the suborders comprising an order and the ship dates of each suborder.

JavaBeans Components

RetailPriceList

RetailPriceList is a list of retail price items. A retail price item contains a coffee name, a wholesale price per pound, a retail price per pound, and a distributor. This data is used for two purposes: it contains the price list presented to the end user and is used by CheckoutFormBean when it constructs the suborders dispatched to coffee distributors.

It first performs a JAXR lookup to determine the JAX-RPC service endpoints. It then queries each JAX-RPC service for a coffee price list. Finally it queries the JAXM service for a price list. The two price lists are combined and a retail price per pound is determined by adding a markup of 35% to the wholesale prices.

Discovering the JAX-RPC Service

Instantiated by RetailPriceList, JAXRQueryByName connects to the registry server and searches for coffee distributors registered with the name JAXRPCCoffeeDistributor in the executeQuery method. The method returns a collection of organizations which contain services. Each service is accessible via a service binding or URI.RetailPriceList makes a JAX-RPC call to each URI.

ShoppingCartItem

ShoppingCart is a list of shopping cart items. A shopping cart item contains a retail price item, the number of pounds of that item, and the total price for that item.

OrderConfirmation

`OrderConfirmations` is a list of order confirmation objects. An order confirmation contains order and confirmation objects, already discussed in Service Interface (page 749).

CheckoutFormBean

`CheckoutFormBean` checks the completeness of information entered into `checkoutForm`. If the information is incomplete, the bean populates error messages and `Dispatcher` redisplays `checkoutForm` with the error messages. If the information is complete, order requests are constructed from the shopping cart and the information supplied to `checkoutForm` and are sent to each distributor. As each confirmation is received, an order confirmation is created and added to `OrderConfirmations`.

```
if (allOk) {
  String orderId = CCNumber;

  AddressBean address = new AddressBean(street, city,
    state, zip);
  CustomerBean customer = new CustomerBean(firstName, lastName,
    "(" + areaCode+ ") " + phoneNumber, email);

  for(Iterator d = rpl.getDistributors().iterator();
    d.hasNext(); ) {
    String distributor = (String)d.next();
    System.out.println(distributor);
    ArrayList lis = new ArrayList();
    BigDecimal price = new BigDecimal("0.00");
    BigDecimal total = new BigDecimal("0.00");
    for(Iterator c = cart.getItems().iterator();
      c.hasNext(); ) {
      ShoppingCartItem sci = (ShoppingCartItem) c.next();
      if ((sci.getItem().getDistributor()).
          equals(distributor) &&
          sci.getPounds().floatValue() > 0) {
        price = sci.getItem().
          getWholesalePricePerPound().
          multiply(sci.getPounds());
        total = total.add(price);
        LineItemBean li = new LineItemBean(
          sci.getItem().getCoffeeName(), sci.getPounds(),
          sci.getItem().getWholesalePricePerPound());
        lis.add(li);
```

```
            }
         }

         if (!lis.isEmpty()) {
            OrderBean order = new OrderBean(orderId,
               customer, lis, total, address);

            String JAXMOrderURL =
               "http://localhost:8080/
                  jaxm-coffee-supplier/orderCoffee";

            if (distributor.equals(JAXMOrderURL)) {
               OrderRequest or = new OrderRequest(JAXMOrderURL);
               confirmation = or.placeOrder(order);
            } else {
               OrderCaller ocaller = new OrderCaller(distributor);
               confirmation = ocaller.placeOrder(order);
            }
            OrderConfirmation oc = new OrderConfirmation(order,
               confirmation);
            ocs.add(oc);
         }
      }
   }
```

RetailPriceListServlet

The `RetailPriceListServlet` responds to requests to reload the price list via the URL `/loadPriceList`. It simply creates a new `RetailPriceList` and a new `ShoppingCart`.

Since this servlet would be used by administrators of the Coffee Break Server, it is a protected Web resource. In order to load the price list, a user must authenticate (using basic authentication) and the authenticated user must be in the `admin` role.

Building, Installing, and Running the Application

The source code for the Coffee Break application is located in the directory *`<JWSDP_HOME>`*`/docs/tutorial/examples/cb`. Within the `cb` directory are subdirectories for each Web application—`jaxm`, `jaxrpc`, `server`—and a directory, `common`, for classes shared by the Web applications. Each subdirectory contains a `build.xml` and `build.properties` file. The Web application subdirectories in turn contain a `src` subdirectory for Java classes and a `web` subdirectory for Web resources and the Web application deployment descriptor.

Note: The Web applications are installed into Tomcat using `ant install` task. Before you can use the `install` task you must create a file named `build.properties` in your home directory that contains the user name and password you provided when you installed the Java WSDP. See Running Manager Commands Using Ant Tasks (page 826).

Building the Common Classes

To build the common classes:

1. In a terminal window, go to *<JWSDP_HOME>*`/docs/tutorial/examples/cb/common`.
2. Run `ant build`.

Building and Installing the JAX-RPC Service

To build the JAX-RPC service and client library and install the JAX-RPC service:

1. In a terminal window, go to *<JWSDP_HOME>*`/docs/tutorial/examples/cb/jaxrpc.`
2. Run `ant build`. This task generates the JAX-RPC ties and stubs, creates the JAXR and client libraries, compiles the server classes, and copies them into the correct location for installation.
3. Start Tomcat and Xindice, if they are not already running. This starts the Registry Server.
4. Run ant `set-up-service`. This task installs the JAX-RPC service into Tomcat and registers the service with the Registry Server. The registration process can take some time, so wait until you see the following output before proceeding to the next step:

```
run-jaxr-publish:
[echo] Running OrgPublisher.
```

```
     [echo] Note: Remember to start the registry server
before
     running this program.
     [java] Created connection to registry
     [java] Got registry service, query manager, and life
cycle
     manager
     [java] Established security credentials
     [java] Organization saved
     [java] Organization key is edeed14d-5eed-eed1-31c2-
     aa789a472fe0
```

5. You can test that the JAX-RPC service has been installed correctly by running one or both of the test programs: execute `ant run-test-price` or `ant run-test-order`. Here is what you should see when you run `ant run-test-price`:

```
run-test-price:
run-test-client:
[java] 05/21/02 06/20/02
[java] Kona 6.50
[java] French Roast 5.00
[java] Wake Up Call 5.50
[java] Mocca 4.00
```

   Later on, you may remove the JAX-RPC service by running `ant take-down-service`. This command deletes the service from the Registry Server and then uninstalls the service from Tomcat. Do not remove the service at this time.

Building and Installing the JAXM Service

To build the JAXM service and client library and install the JAXM service:

1. In a terminal window, go to *<JWSDP_HOME>*`/docs/tutorial/examples/cb/jaxm`.
2. Run `ant build`. This task creates the client library and compiles the server classes and copies them into the correct location for installation.
3. Make sure Tomcat is started.
4. Run ant `install`. This task installs the JAXM service into Tomcat.

5. You can test that the JAXM service has been installed correctly by running one or both of the test programs: execute `ant run-test-price` or `ant run-test-order`.

Building and Installing the Coffee Break Server

To build and install the Coffee Break server:

1. In a terminal window, go to *<JWSDP_HOME>*`/docs/tutorial/examples/cb/server`.
2. Run `ant build`. This task compiles the server classes and copies the classes, JSP pages, client libraries, and tag libraries into the correct location for installation. Note that the Coffee Break server depends on the client libraries generated by the JAX-RPC (`jaxrpc-client.jar`) and JAXM (`jaxm-client.jar`) build process.
3. Make sure Tomcat is started.
4. Run `ant install`.

Running the Coffee Break Client

After you have installed all the Web applications, check that all the applications are running by executing `ant list` in a terminal window or opening `http://localhost:8080/manager/list`. In a browser, you should see something like:

```
OK - Listed applications for virtual host localhost
/manager:running:0:../server/webapps/manager
/jaxm-translator:running:0:D:\jwsdp-1_0\webapps\jaxm-
translator.war
/jaxm-coffee-supplier:running:0:D:/jwsdp-1_0/docs/tuto-
rial/examples/cb/jaxm/build
/jaxm-soaprp:running:0:D:\jwsdp-1_0\webapps\jaxm-soaprp.war
/saaj-simple:running:0:D:\jwsdp-1_0\webapps\saaj-simple.war
/jaxm-remote:running:0:D:\jwsdp-1_0\webapps\jaxm-remote.war
/jstl-examples:running:0:D:\jwsdp-1_0\webapps\jstl-
examples.war
/registry-server:running:0:D:\jwsdp-1_0\webapps
egistry-server.war
/jaxmtags:running:0:D:\jwsdp-1_0\webapps\jaxmtags.war
/jaxm-simple:running:0:D:\jwsdp-1_0\webapps\jaxm-simple.war
```

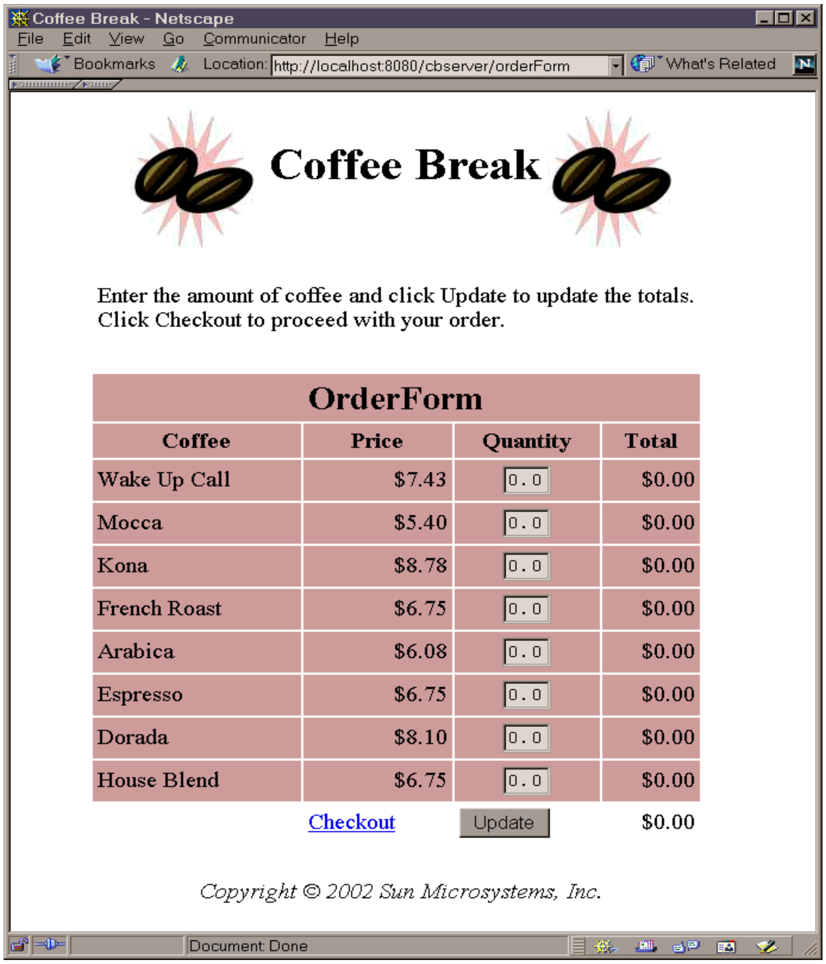
```
/jaxrpc-coffee-supplier:running:0:D:/jwsdp-1_0/docs/tuto-
rial/examples/cb/jaxrpc/build
/cbserver:running:1:D:/jwsdp-1_0/docs/tutorial/exam-
ples/cb/server/build
/:running:0:D:\jwsdp-1_0\webapps\ROOT
/admin:running:0:../server/webapps/admin
```

The highlighted applications are the Coffee Break server and the JAX-RPC and JAXM services.

Then, to run the Coffee Break client, open the Coffee Break server URL in a Web browser:

```
http://localhost:8080/cbserver/orderForm
```

You should see a page something like the one shown in Figure 19–2.


Coffee Break - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://localhost:8080/cbserver/orderForm What's Related
Document: Done

Coffee Break

Enter the amount of coffee and click Update to update the totals. Click Checkout to proceed with your order.

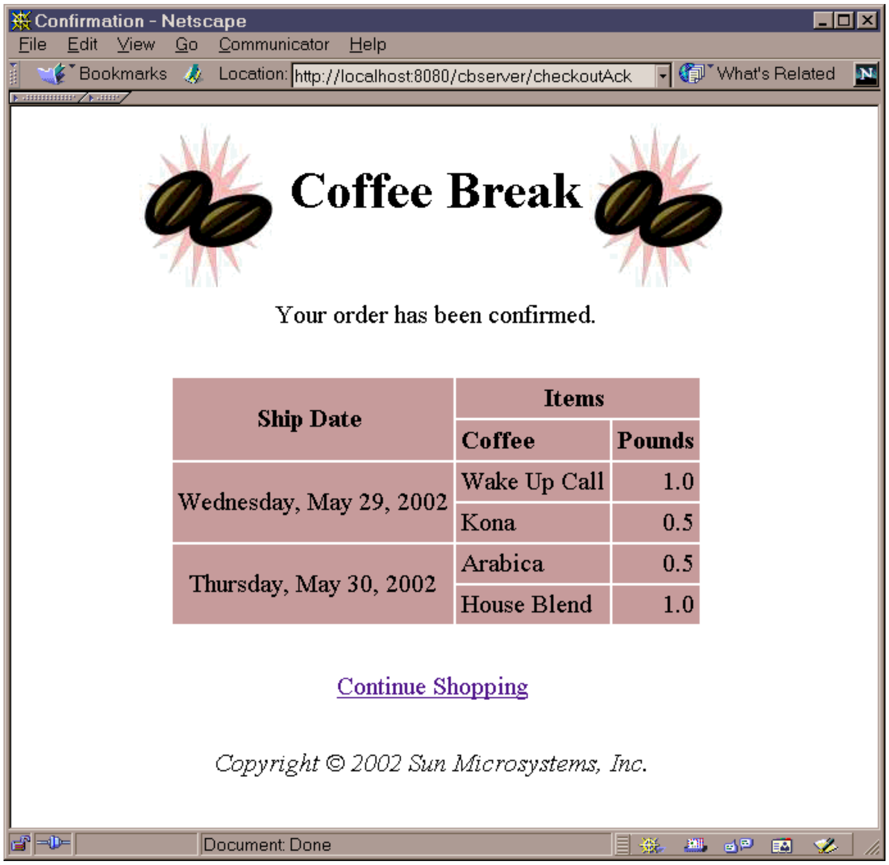
OrderForm

Coffee	Price	Quantity	Total
Wake Up Call	$7.43	0.0	$0.00
Mocca	$5.40	0.0	$0.00
Kona	$8.78	0.0	$0.00
French Roast	$6.75	0.0	$0.00
Arabica	$6.08	0.0	$0.00
Espresso	$6.75	0.0	$0.00
Dorada	$8.10	0.0	$0.00
House Blend	$6.75	0.0	$0.00
	Checkout	Update	$0.00

Copyright © 2002 Sun Microsystems, Inc.

Figure 19–2 Order Form

After you have gone through the application screens, you will get an order confirmation that looks like the one shown in Figure 19–3.


Confirmation - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://localhost:8080/cbserver/checkoutAck What's Related
Coffee Break
Document: Done

Your order has been confirmed.

Ship Date	Items	
	Coffee	Pounds
Wednesday, May 29, 2002	Wake Up Call	1.0
	Kona	0.5
Thursday, May 30, 2002	Arabica	0.5
	House Blend	1.0

Continue Shopping

Copyright © 2002 Sun Microsystems, Inc.

Figure 19–3 Order Confirmation

Deploying the Coffee Break Application

The instructions in the previous section described how to install and run the Coffee Break application. However, an installed application is not available when Tomcat is restarted. To permanently deploy the application:

1. Remove the JAXRPC and JAXM services and the Coffee Break server by executing `ant remove` in each Web application directory.
2. Package the applications into WAR files by executing `ant package` in each Web application directory.
3. Deploy the application by executing `ant deploy` in each Web application directory.

A

Tomcat Administration Tool

Debbie Carson

This appendix contains information about the Tomcat Web Server Administration Tool. The Tomcat Web Server Administration Tool is referred to as `admintool` throughout this section for ease of reference.

The `admintool` utility is used to configure the behavior of the Tomcat Java Servlet/JSP container while it is running. Changes made to Tomcat using `admintool` can be saved persistently so that the changes remain when Tomcat is restarted, or the changes can be attributed to the current session only.

Running admintool

The `admintool` Web application can be used to manipulate Tomcat while it is running. For example, you can add a context or set up users and roles for container-managed security.

To start admintool, follow these steps.

1. Start Tomcat by calling its startup script from the command line, as follows:

```
<JWSDP_HOME>/bin/startup.sh           (Unix platform)

<JWSDP_HOME>\bin\start startup.bat    (Microsoft Windows)
```

2. Start a Web browser.
3. In the Web browser, point to the following URL:

   ```
   http://localhost:8080/admin
   ```

   This command invokes the Web application with the context of `admin`.

4. Log in to `admintool` using the user name and password combination defined when you installed the Java WSDP.

   This user name and password combination is assigned the roles of `admin`, `manager`, and `provider` by default. To use `admintool`, you must log in with a user name and password combination that has been assigned the role of `admin`.

   If you've forgotten the user name and password, you can find them in the file *`<JWSDP_HOME>`*`/conf/tomcat-users.xml`, which is viewable with any text editor. This file contains an element `<user>` for each individual user, which might look something like this:

   ```
   <user name="your_name" password="your_password"
   roles="admin,manager,provider" />
   ```

The `admintool` Web application displays in the Web browser window:

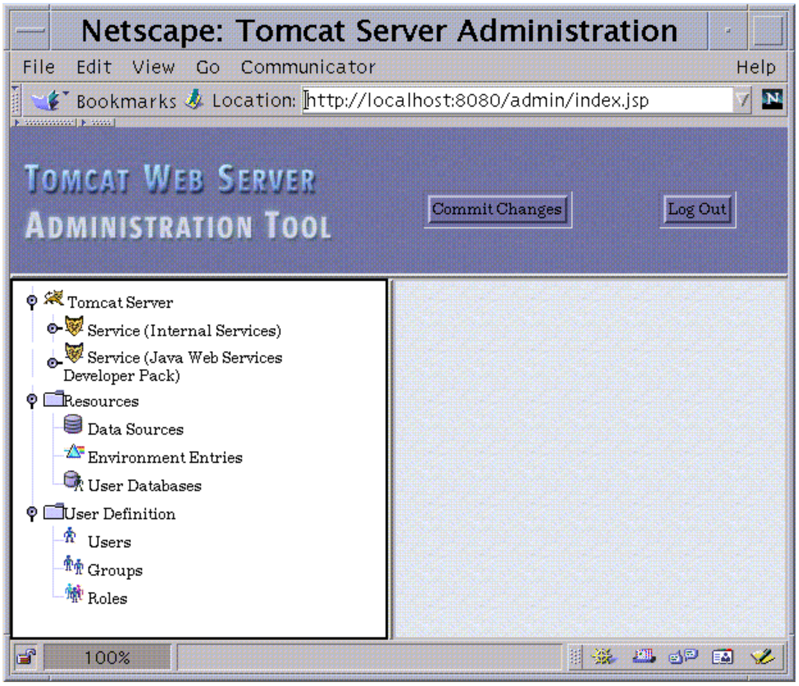

Netscape: Tomcat Server Administration
File Edit View Go Communicator Help
Bookmarks Location: http://localhost:8080/admin/index.jsp
TOMCAT WEB SERVER ADMINISTRATION TOOL
Commit Changes
Log Out
Tomcat Server
Service (Internal Services)
Service (Java Web Services Developer Pack)
Resources
Data Sources
Environment Entries
User Databases
User Definition
Users
Groups
Roles
100%

Figure A–1 The Tomcat Server Administration Tool

5. Perform Tomcat Web Server Administration tasks.

   After you have made changes to Tomcat, select the Save button on that page to save the attributes for the current Tomcat process. Select the Commit Changes button to write the changes to the <*JWSDP_HOME*>`/conf/server.xml` file so that the changes to the Tomcat server are persistent and will be retrieved when Tomcat is restarted.

   The previous version of `server.xml` is backed up in the same directory, with an extension indicating when the file was backed up, for example, `server.xml.2003-02-15.12-11-54`. To restore a previous configuration, shut down Tomcat, rename the file to `server.xml`, and restart Tomcat.

6. Log out of `admintool` by selecting Log Out when you are finished.
7. Shut down Tomcat by calling its shutdown script from the command line, as follows:

```
<JWSDP_HOME>/bin/shutdown.sh          (Unix platform)

<JWSDP_HOME>\bin\shutdown.bat         (Microsoft Windows)
```

This document contains information about using `admintool` to configure the behavior of Tomcat. For more information on these configuration elements, read the Tomcat Configuration Reference, which can be found at `<`*`JWSDP_HOME`*`>/docs/tomcat/config/index.html`.

This document does not attempt to describe which configurations should be used to perform specific tasks. For information of this type, refer to the documents listed in Further Information (page 824).

Configuring Tomcat

As you can see in Figure A–1, `admintool` presents a hierarchy of elements that can be configured to customize the Tomcat JSP/Servlet container to your needs. The Server element represents the characteristics of the entire JSP/Servlet container.

Setting Server Properties

Select Tomcat Server in the left pane. The Server Properties display in the right pane. The Server element represents the entire JSP/Servlet container. The server properties are shown in Table A–1.

Table A–1 Server Properties

Property	**Description**
Port Number	The TCP/IP port number on which this server waits for a shutdown command. This connection must be initiated from the same server computer that is running this instance of Tomcat. The default value is 8005. Values less than 1024 will generate a warning, as special software capabilities are required when using this port
Debug Level	The level of debugging detail logged by this server. Higher numbers generate more detailed output. If not specified, the default debugging detail level is zero (0).
Shutdown	The command string that must be received via a TCP/IP connection to the specified port number in order to shut down Tomcat. The value for this property must contain at least 6 characters. The default value is SHUTDOWN.

Configuring Services

Service elements are nested with the Server element. The Service element represents the combination of one or more Connector components that share a single engine component for processing incoming requests. The default configuration for Tomcat includes an Internal Service and a Java Web Services Developer Pack Service.

- The Internal Service uses port 8081. This service is used internally by Tomcat Web applications such as JAXM `provider` and JAXM `provideradmin` contexts. These contexts are used by the JAXM Web applications contexts in the JWSDP Service.
- The Java Web Services Developer Pack Service uses port 8080, the standard port on which users can deploy their Web applications. For Java Servlet and JSP pages developers, this is the service to use.

It is possible to use `admintool` to add other services, which might use a different port. To create a new service,

1. Select Tomcat Server in the left pane.
2. Select Create New Service from the drop-down list in the right pane.
3. Enter the values for Service Name, Engine Name, Debug Level, and Default Hostname.

   The Service Name is the display name of this Service, which will be included in log messages if you choose a Logger (see Configuring Logger Elements, page 802).

Note: The name of each Service associated with a particular Server must be unique.

For each Service element defined, you can create or delete the following elements:

- **Connector** elements represent the interface between the Service and external clients that send requests to it and receive responses from it. See Configuring Connector Elements (page 790) for more information.
- **Host** elements represent a virtual host, which is an association of a network name for a server (such as `www.mycompany.com`) with the particular

server on which Tomcat is running. See Configuring Host Elements (page 795) for more information.

- **Logger** elements represent a destination for logging, debugging, and error messages (including stack tracebacks) for Tomcat (Engine, Host, or Context). See Configuring Logger Elements (page 802) for more information.
- User **Realm** elements represent a database of user names, passwords, and roles assigned to those users. See Configuring Realm Elements (page 805) for more information.
- **Valve** elements represent a component that will be inserted into the request processing pipeline for the associated container (Engine, Host, or Context). See Configuring Valve Elements (page 812) for more information.

Configuring Connector Elements

Connector elements represent the interface between external clients sending requests to (and receiving responses from) a particular Service.

To edit a connector,

1. Expand the Service element in the left pane.
2. Select the Connector to edit.
3. Edit the values in the right pane.
4. Select Save to save the changes for this session. Select Commit Changes to save the changes for when Tomcat is restarted.

To create a new connector for a service,

1. Select the Service element in the left pane. It is highly recommended that you only modify the Java Web Services Developer Pack Service, or a service that you have created.
2. Select Create New Connector from the Available Actions list.
3. Enter the preferred values for the Connector. See Connector Attributes (page 792) for more information on the options.
4. Select Save to save the changes for this session. Select Commit Changes to save the changes for when Tomcat is restarted.

To learn more about Connectors, read the documents titled *Coyote HTTP/1.1 Connectors* at `<JWSDP_HOME>/docs/tomcat/config/coyote.html` or the document titled *JK 2 Connectors* at `<JWSDP_HOME>/docs/tomcat/config/jk2.html`.

Types of Connectors

Using `admintool`, you can create the following types of Connectors:

- HTTP

  Selecting HTTP enables you to create a Connector component that supports the HTTP/1.1 protocol. It enables Tomcat to function as a stand-alone Web server in addition to its ability to execute Java Servlets and JSP pages. A particular instance of this component listens for connections on a specific TCP port number on the server. One or more such Connectors can be configured as part of a single Service, each forwarding to the associated Engine to perform request processing and create the response.

- HTTPS

  Selecting HTTPS enables you to create an SSL HTTP/1.1 Connector. Secure Socket Layer (SSL) technology enables Web browsers and Web servers to communicate over a secure connection. In order to implement SSL, a Web server must have an associated keystore certificate for each external interface (IP address) that accepts secure connections. Installing and Configuring SSL Support (page 721) contains detailed instructions on setting up an HTTPS connector.

- AJP

  Selecting AJP enables you to create a Connector component that communicates with a Web connector via the Apache JServ Protocol ("AJP"). This is used for cases where you wish to invisibly integrate Tomcat into an existing (or new) Apache installation, and you want Apache to handle the static content contained in the Web application, and/or utilize Apache's SSL processing. In many application environments, this will result in better overall performance than running your applications under Tomcat stand-alone using the HTTP/1.1 Connector. However, the only way to know for sure whether it will provide better performance for your application is to try it both ways.

Connector Attributes

When you create or modify any type of Connector, the attributes shown in Table A–2 may be set, as needed.

Table A–2 Common Connector Attributes

Attribute	**Description**
Accept Count	The maximum queue length for incoming connection requests when all possible request processing threads are in use. Any requests received when the queue is full will be refused. The default value is 10.
Connection Timeout	The number of milliseconds this Connector will wait, after accepting a connection, for the request URI line to be presented. The default value is 60000 (i.e. 60 seconds).
Debug Level	The debugging detail level of log messages generated by this component, with higher numbers creating more detailed output. If not specified, this attribute is set to zero (0).
Default Buffer Size	The size (in bytes) of the buffer to be provided for input streams created by this connector. By default, buffers of 2048 bytes will be provided.
Enable DNS Lookups	Whether or not you want calls to `request.getRemoteHost()` to perform DNS lookups in order to return the actual host name of the remote client. Set to True if you want calls to `request.getRemoteHost()` to perform DNS lookups in order to return the actual host name of the remote client. Set to False to skip the DNS lookup and return the IP address in String form instead (thereby improving performance).
IP Address	Specifies which address will be used for listening on the specified port, for servers with more than one IP address. By default, this port will be used on all IP addresses associated with the server.

Table A–2 Common Connector Attributes (Continued)

Attribute	Description
Port Number	The TCP port number on which this Connector will create a server socket and await incoming connections. Your operating system will allow only one server application to listen to a particular port number on a particular IP address.
Redirect Port Number	The port number where Tomcat will automatically redirect the request if this Connector is supporting non-SSL requests, and a request is received for which a matching security constraint requires SSL transport.
Minimum	The number of request processing threads that will be created when this Connector is first started. This attribute should be set to a value smaller than that set for Maximum. The default value is 5.
Maximum	The maximum number of request processing threads to be created by this Connector, which therefore determines the maximum number of simultaneous requests that can be handled. If not specified, this attribute is set to 75.

When the Connector is of type HTTP or HTTPS, additional attributes are also available, as shown in Table A–3.

Table A–3 Attributes of HTTP/HTTPS Connectors

Attribute	Description
Proxy Name	The server name to be returned for calls to `request.getServerName()` if this Connector is being used in a proxy configuration.
Proxy Port Number	The server port to be returned for calls to `request.getServerPort()` if this Connector is being used in a proxy configuration.

When the type of Connector is HTTPS, additional attributes as outlined in Table A–4 may also be set.

Table A–4 HTTPS Attributes

Attribute	**Description**
Client Authentication	Whether or not you want the SSL stack to require a valid certificate chain from the client before accepting a connection. Set to True if you want the SSL stack to require a valid certificate chain from the client before accepting a connection. A False value (which is the default) will not require a certificate chain unless the client requests a resource protected by a security constraint that uses client-certificate authentication.
Keystore Filename	The path to and name of the keystore file where you have stored the server certificate to be loaded. By default, the file name is `.keystore` and the path name is the operating system home directory of the user that is running Tomcat. If you are using default values for the file name and path, you can leave this field blank. If you specify a keystore file name without specifying a path, `admintool` looks for the file in the <*JWSDP_HOME*> directory.
Keystore Password	The password used to access the server certificate from the specified keystore file. The default value is `changeit`.

Note: In order to use an SSL connector, you must use `keytool` to generate a keystore file. If you have generated a keystore file with the default name (`.keystore`) in the default directory (the operating system home directory of the user that is running Tomcat) with default password (`changeit`), you can leave the Keystore Filename and Keystore Password attributes empty when creating an SSL Connector. When the two properties are left empty, `admintool` will look for the keystore file with the default name (`.keystore`) and the default password (`changeit`) in the default location (the operating system home directory of the user that is running Tomcat). If you specify a keystore file name without specifying a path, `admintool` looks for the file in the <*JWSDP_HOME*> directory. Installing and Configuring SSL Support (page 721) contains detailed instructions on setting up an HTTPS connector.

Configuring Host Elements

The Host element represents a virtual host, which is an association of a network name for a server (such as `www.mycompany.com`) with the particular server on which Tomcat is running. In order to be effective, this name must be registered in the Domain Name Service (DNS) server that manages the Internet domain to which you belong.

In many cases, system administrators wish to associate more than one network name (such as `www.mycompany.com` and `company.com`) with the same virtual host and applications. This can be accomplished using the Host Name Aliases feature described in Host Name Aliases (page 797).

One or more Host elements are nested inside a Service. Exactly one of the Hosts associated with each Service MUST have a name matching the `defaultHost` attribute of that Service. Inside the Host element, you can nest any of the following elements:

- Context elements, which are discussed in Configuring Context Elements (page 797).
- Logger Elements, which are discussed in Configuring Logger Elements (page 802).
- Valve Elements, which are discussed in Configuring Valve Elements (page 812).
- Host Aliases, which are discussed in Host Name Aliases (page 797).

To edit a Host,

1. Expand the Service element in the left pane.
2. Expand the Host element in the left pane.
3. Select the Host, or any of its Contexts, Valves, Loggers, or Aliases, to edit.
4. Edit the values in the right pane.
5. Select Save to save the changes for this session. Select Commit Changes to save the changes for when Tomcat is restarted.

To create a new Host for a service,

1. Select the Service element in the left pane. It is highly recommended that you only modify the Java Web Services Developer Pack Service, or a service that you have created.

2. Select Create New Host from the Available Actions list.
3. Enter the preferred values for the Host. See Host Attributes (page 796) for more information on the options.
4. Select Save to save the changes for this session. Select Commit Changes to save the changes for when Tomcat is restarted.

To learn more about Hosts, read the document titled "Host Container" at *<JWSDP_HOME>*`/docs/tomcat/config/host.html`.

Host Attributes

The attributes shown in Table A–5 may be viewed, set, or modified for a Host.

Table A–5 Host Attributes

Attribute	**Description**
Name	The network name of this virtual host, as registered in your Domain Name Service server. One of the Hosts nested within an Engine MUST have a name that matches the `defaultHost` setting for that Engine.
Application Base	The Application Base directory for this virtual host. This is the path name of a directory that may contain Web applications to be deployed on this virtual host. You may specify an absolute path name for this directory, or a path name that is relative to the directory under which Tomcat is installed.
Debug Level	The level of debugging detail logged by this Engine to the associated Logger. Higher numbers generate more detailed output. If not specified, the default debugging detail level is zero (0).
Unpack WARs	Whether or not you want Web applications that are deployed into this virtual host from a Web Application Archive (WAR) file to be unpacked into a disk directory structure. Set to True if you want Web applications that are deployed into this virtual host from a Web Application Archive (WAR) file to be unpacked into a disk directory structure or False to run the application directly from a WAR file. The default value is False.

Host Name Aliases

In many server environments, Network Administrators have configured more than one network name (in the Domain Name Service (DNS) server) that resolve to the IP address of the same server. Normally, each such network name would be configured as a separate Host element with its own set of Web applications.

However, in some circumstances it is desirable for two or more network names to resolve to the same virtual host, running the same set of applications. A common use case for this scenario is a corporate Web site where users should be able to utilize either `www.mycompany.com` or `company.com` to access exactly the same content and applications.

Tomcat supports virtual hosts, which are multiple "hosts + domain names" mapped to a single IP. Usually, each host name is mapped to a host in Tomcat, for example, `www.foo.com` is mapped to `localhost`, or `www.foo1.com` is mapped to `localhost1`. In some cases, various host names can be mapped to the same host, for example `www.foo.com` and `www.foo1.com` can both be mapped to `localhost`. In this situation, you will see both of these aliases listed under `localhost` in `admintool`.

To use Host Aliases, the DNS server must have the host names registered to the IP of the server on which Tomcat will be running.

To create a new Host alias,

1. Select the Host element in the left pane.
2. Select Create New Aliases from the Available Actions list.
3. Enter the name for the Alias.
4. Select Save to save the changes for this session. Select Commit Changes to save the changes for when Tomcat is restarted.

Configuring Context Elements

The Context element represents a Web application that is run within a particular virtual host. Each Web application is based on a Web Application Archive (WAR) file or a directory containing the Web application in its unpacked form. For more information about WAR files, see Web Application Archives (page 96).

When an HTTP request is received, Tomcat selects the Web application that will be used to process the request. To select the Web application, Tomcat matches the longest prefix of the Request URI against the context path of each defined

Context. Once a Context is selected, it selects an appropriate Servlet to process the incoming request, based on the Servlet mappings defined in the Web application deployment descriptor, which must be located at `<web_app_root>/WEB-INF/web.xml`.

You can define as many Context elements within a Host element as you wish, but each must have a unique context path. At least one Context must include a context path equal to a zero-length string. This Context becomes the default Web application for this virtual host and is used to process all requests that do not match any other Context's context path.

To edit a Context,

1. Expand the Service element in the left pane.
2. Expand the Host element in the left pane.
3. Select the Context to edit.
4. Edit the values in the right pane.
5. Select Save to save the changes for this session. Select Commit Changes to save the changes for when Tomcat is restarted.

To create a new Context for a service,

1. Select the Service element in the left pane.
2. Select the Host element in the left pane to which you want to add the Context.
3. Select Create New Context from the Available Actions list.
4. Enter the preferred values for the Context. See Context Attributes (page 798) for more information on the options.
5. Select Save to save the changes for this session. Select Commit Changes to save the changes for when Tomcat is restarted.

To learn more about Contexts, read the document titled "The Context Container" at `<JWSDP_HOME>/docs/tomcat/config/context.html`.

Context Attributes

The Context element page contains three types of properties:

- Context Properties, described in Table A–6.
- Loader Properties, described in Table A–7.
- Session Manager Properties, described in Table A–8.

The attributes shown in Table A–6 may be viewed, set, or modified for Context properties.

Table A–6 Context Properties

Attribute	Description
Cookies	Set to True if you want cookies to be used for session identifier communication if supported by the client. Set to False if you want to disable the use of cookies for session identifier communication and rely only on URL rewriting by the application. The default value is True.
Cross Context	Set to True if you want calls to `ServletContext.getContext()` within this application to successfully return a request dispatcher for other Web applications running on this virtual host. Set to False in security-conscious environment to make `getContext()` always return `null`. The default value is False.
Debug Level	The level of debugging detail logged by this Engine to the associated Logger. Higher numbers generate more detailed output. If not specified, the default debugging detail level is zero (0).
Document Base	The Document Base (also known as the Context Root) directory for this Web application is the path to the Web Application Archive file (if this Web application is being executed directly from the WAR file). You may specify an absolute path name for this directory or WAR file, or a path name that is relative to the application base directory of the owning Host.
Override	Set to True to have explicit settings in this Context element override any corresponding settings in the `DefaultContext` element associated with the owning Host. By default, settings in the `DefaultContext` element will be used.The default value is False.

Table A–6 Context Properties

Attribute	Description
Path	The context path of this Web application, which is matched against the beginning of each request URI to select the appropriate Web application for processing. All of the context paths within a particular Host must be unique. If you specify a context path of an empty string (""), you are defining the default Web application for this Host, which will process all requests not assigned to other Contexts.
Reloadable	Set to True if you want Tomcat to monitor classes in `/WEB-INF/classes/` and `/WEB-INF/lib` for changes and automatically reload the Web application if a change is detected. This feature is very useful during application development, but it requires significant runtime overhead and is not recommended for use on deployed production applications. You can use the Manager Web application to trigger reloads of deployed applications on demand. The default value is False.
Use Naming	Set to True to have Tomcat enable a JNDI `InitialContext` for this Web application that is compatible with Java2 Enterprise Edition (J2EE) platform conventions.The default value is False.
Working Directory	Path to a scratch directory for temporary read-write use by Servlets within the associated Web application. This directory will be made visible to Servlets in the Web application by a Servlet context attribute (of type `java.io.File`) named `javax.servlet.context.tempdir` as described in the Servlet Specification. If not specified, a suitable directory underneath `<`*`JWSDP_HOME`*`>/work` will be provided.

The Loader Properties section enables you to configure the Web application class loader that will be used to load Servlet and JavaBeans classes for this Web application. Normally, the default configuration of the class loader will be sufficient. The attributes shown in Table A–7 may be viewed, set, or modified for Loader properties.

Table A–7 Loader Properties

Attribute	**Description**
Check Interval	The number of seconds between checks for modified classes and resources if Reloadable has been set to True. The default value is 15 seconds.
Debug Level	The level of debugging detail logged by this Engine to the associated Logger. Higher numbers generate more detailed output. If not specified, the default debugging detail level is zero (0).
Reloadable	Set to True if you want Tomcat to monitor classes in `/WEB-INF/classes/` and `/WEB-INF/lib` for changes and automatically reload the Web application if a change is detected. This feature is very useful during application development, but it requires significant runtime overhead and is not recommended for use on deployed production applications. You can use the Manager Web application when you need to trigger reloads of deployed applications on demand.The default value is False.

The Session Manager Properties enable you to configure the session manager that will be used to create, destroy, and persist HTTP sessions for this Web application. Normally, the default configuration of the session manager will be sufficient. The attributes shown in Table A–8 may be viewed, set, or modified for Session Manager properties.

Table A–8 Session Manager Properties

Attribute	**Description**
Check Interval	The number of seconds between checks for expired sessions for this manager. The default value is 60 seconds.

Table A–8 Session Manager Properties

Attribute	Description
Debug Level	The level of debugging detail logged by this Manager to the associated Logger. Higher numbers generate more detailed output. If not specified, the default debugging detail level is zero (0).
Session ID Initializer	Tomcat provides two standard implementations of Session Managers. `org.apache.catalina.session.StandardManager` stores active sessions. `org.apache.catalina.session.PersistentManager` persistently stores active sessions that have been swapped out (in addition to saving sessions across a restart of Tomcat) in a storage location that is selected via the use of an appropriate `Store` nested element. In addition to the usual operations of creating and deleting sessions, a `PersistentManager` has the capability to swap active (but idle) sessions out to a persistent storage mechanism, as well as to save all sessions across a normal restart of Tomcat. The actual persistent storage mechanism that is used is selected by your choice of a Store element nested inside the Manager element - this is required for use of `PersistentManager`.
Maximum Active Sessions.	The maximum number of active sessions that will be created by this Manager, or -1 (the default) for no limit.

Configuring Logger Elements

A Logger element represents a destination for logging, debugging, and error messages (including stack tracebacks) for Tomcat.

If you are interested in producing access logs as a Web server does (for example, to run hit count analysis software), you will want to configure an Access Log Valve component on your Engine, Host, or Context.

Using `admintool`, you can create 3 types of loggers:

- `SystemOutLogger`

The Standard Output Logger records all logged messages to the stream to which the standard output of Tomcat is pointed. The default Tomcat startup script points this at the file `logs/catalina.out` relative to the directory where Tomcat is installed.

- `SystemErrLogger`
  The Standard Error Logger records all logged messages to the stream to which the standard error output of Tomcat is pointed. The default Tomcat startup script points this at the file `logs/catalina.out` relative to the directory where Tomcat is installed.
- `FileLogger`
  The File Logger records all logged messages to disk file(s) in a specified directory. The actual filenames of the log files are created from a configured prefix, the current date in YYYY-MM-DD format, and a configured suffix. On the first logged message after midnight each day, the current log file will be closed and a new file opened for the new date, without your having to shut down Tomcat in order to perform this switch.

To edit a Logger,

1. Expand the Service element in the left pane.
2. Select the Logger to edit.
3. Edit the values in the right pane.
4. Select Save to save the changes for this session. Select Commit Changes to save the changes for when Tomcat is restarted.

To create a new Logger for a service,

1. Select the Service element in the left pane. It is highly recommended that you only modify the Java Web Services Developer Pack Service, or a service that you have created.
2. Select Create New Logger from the Available Actions list.
3. Enter the preferred values for the Logger. See Logger Attributes (page 804) for more information on the options.
4. Select Save to save the changes for this session. Select Commit Changes to save the changes for when Tomcat is restarted.

To learn more about Loggers, read the document titled “Logger Component” at `<JWSDP_HOME>/docs/tomcat/config/logger.html`.

Logger Attributes

Common attributes for all of the Logger types are outlined in Table A–9.

Table A–9 Logger Attributes

Attribute	Description
Type	The type of Logger to create: `SystemOutLogger`, `SystemErrLogger`, or `FileLogger`.
Debug Level	The level of debugging detail logged by this Logger. Higher numbers generate more detailed output. If not specified, the default debugging detail level is zero (0).
Verbosity Level	The verbosity level for this logger. Messages with a higher verbosity level than the specified value will be silently ignored. Available levels are 0 (fatal messages only), 1 (errors), 2 (warnings), 3 (information), and 4 (debug). The default value is 0 (fatal messages only).

If you are using a Logger of type `FileLogger`, additional attributes that may be set are shown in Table A–10.

Table A–10 FileLogger Attributes

Attribute	Description
Directory	The absolute or relative path name of a directory in which log files created by this logger will be placed. If a relative path is specified, it is interpreted as relative to the directory in which Tomcat is installed. If no directory attribute is specified, the default value is `logs` (relative to the directory in which Tomcat is installed).
Prefix	The prefix added to the start of each log file's name. If not specified, the default value is `catalina`. To specify no prefix, use a zero-length string.

Table A–10 FileLogger Attributes (Continued)

Attribute	Description
Suffix	The suffix added to the end of each log file's name. If not specified, the default value is `.log`. To specify no suffix, use a zero-length string.
Timestamp	Whether or not all logged messages are to be date and time stamped. Set to True to cause all logged messages to be date and time stamped. Set to False to skip date/time stamping.

Configuring Realm Elements

A Realm element represents a database of user names, passwords, and roles (similar to Unix groups) assigned to those users. Different implementations of Realm allow Tomcat to be integrated into environments where such authentication information is already being created and maintained, and then to utilize that information to implement container managed security (as described in the Java Servlet Specification, available online at `http://java.sun.com/products/servlet/download.html`).

The Realm created inside the Service in which Tomcat is running can not be edited or deleted, and no other Realm can be added to this service. In the Java WSDP, this is the Service (Java Web Services Developer Pack). You can create a Realm inside a Service you have defined and added to Tomcat.

To edit a Realm,

1. Expand the Service element in the left pane.
2. Select the Realm to edit.
3. Edit the values in the right pane.
4. Select Save to save the changes for this session. Select Commit Changes to save the changes for when Tomcat is restarted.

A Realm can be created inside any container Engine, Host, or Context. There can be only one instance of a Realm under each of these. Realms associated with an Engine or a Host are automatically inherited by lower-level containers, unless explicitly overridden. To add a new Realm,

1. Select the service, host, or context under which the new Realm is to be created.

2. Select the Create New User Realm option from the Available Actions list. Select the type of Realm. Depending on the type of Realm you choose, the attributes vary.

There are several standard Realm implementations available, including:

- `JDBCRealm`

  The JDBC Database Realm connects Tomcat to a relational database, accessed through an appropriate JDBC driver, to perform lookups of user names, passwords, and their associated roles. Because the lookup is done each time it is required, changes to the database will be immediately reflected in the information used to authenticate new logins. Attributes for the JDBC Database Realm implementation are shown in JDBCRealm Attributes (page 807).

- `JNDIRealm`

  The JNDI Directory Realm connects Tomcat to an LDAP Directory, accessed through an appropriate JNDI driver, to perform lookups of user names, passwords, and their associated roles. Because the lookup is done each time it is required, changes to the directory will be immediately reflected in the information used to authenticate new logins. Attributes for the JNDI Database Realm implementation are shown in JNDIRealm Attributes (page 808).

- `MemoryRealm`

  The Memory Based Realm is a simple Realm implementation that reads an XML file to configure valid users, passwords, and roles. The file format and default file location are identical to those currently supported by Tomcat 3.x. This implementation is intended solely to get up and running with container managed security - it is NOT intended for production use. As such, there are no mechanisms for updating the in-memory collection of users when the content of the underlying data file is changed. Attributes for the Memory Realm implementation are shown in MemoryRealm Attributes (page 811).

- `UserDatabaseRealm`

  `UserDatabaseRealm` is an implementation of Realm based on an implementation of `UserDatabase` made available through the global JNDI resources configured for the instance of Tomcat. The Resource Name parameter is set to the global JNDI resources name for the configured instance of `UserDatabase` to be consulted. Attributes for the User Database Realm implementation are shown in UserDatabaseRealm Attributes (page 810).

To learn more about Realms, read the document titled *Realm Component* at *<JWSDP_HOME>*/docs/tomcat/config/realm.html or *Realm Configuration How To* at *<JWSDP_HOME>*/docs/tomcat/realm-howto.html.

JDBCRealm Attributes

The JDBC Database Realm connects Tomcat to a relational database, accessed through an appropriate JDBC driver, to perform lookups of use names, passwords, and their associated roles. Because the lookup is done each time it is required, changes to the database will be immediately reflected in the information used to authenticate new logins. Attributes for the JDBC Database Realm implementation are shown in Table A–11.

Table A–11 JDBCRealm Attributes

Attribute	**Description**
Database Driver	Fully qualified Java class name of the JDBC driver to be used to connect to the authentication database.
Database Password	The database password to use when establishing the JDBC connection.
Database URL	The connection URL to be passed to the JDBC driver when establishing a database connection.
Database User Name	The database user name to use when establishing the JDBC connection.
Debug Level	The level of debugging detail logged by this Engine. Higher numbers generate more detailed output. If not specified, the default debugging detail level is zero (0).
Digest Algorithm	The name of the `MessageDigest` algorithm used to encode user passwords stored in the database. If not specified, user passwords are assumed to be stored in text.
Password Column	Name of the column in the users table that contains the user's credentials (i.e. password). If a value for Digest Algorithm is specified, the component will assume that the passwords have been encoded with the specified algorithm. Otherwise, they will be assumed to be in clear text.

Table A–11 JDBCRealm Attributes

Attribute	**Description**
Role Name Column	Name of the column, in the user roles table, which contains a role name assigned to the corresponding user.
User Name Column	Name of the column, in the users and user roles table, that contains the user's user name.
User Role Table	Name of the user roles table, which must contain columns named by the User Name Column and Role Name Column attributes.
User Table	Name of the users table, which must contain columns named by the User Name Column and Password Column attributes.

JNDIRealm Attributes

The JNDI Directory Realm connects Tomcat to an LDAP Directory, accessed through an appropriate JNDI driver, to perform lookups of user names, passwords, and their associated roles. Because the lookup is done each time it is required, changes to the directory will be immediately reflected in the information used to authenticate new logins.

A rich set of attributes lets you configure the required connection to the underlying directory, as well as the element and attribute names used to retrieve the required information. Attributes for the JNDI Directory Realm implementation are shown in Table A–12.

Table A–12 JNDIRealm Attributes

Attribute	**Description**
Connection Name	The directory user name to use when establishing the JNDI connection. This attribute is required if you specify the User Password attribute, and is not used otherwise.
Connection Password	The directory password to use when establishing the JNDI connection. This attribute is required if you specify the User Password property, and is not used otherwise.

Table A–12 JNDIRealm Attributes

Attribute	Description
Connection URL	The connection URL to be passed to the JNDI driver when establishing a connection to the directory.
Context Factory	Fully qualified Java class name of the factory class used to acquire our JNDI `InitialContext`. By default, assumes that the standard JNDI LDAP provider will be utilized.
Debug Level	The level of debugging detail logged by this Engine. Higher numbers generate more detailed output. If not specified, the default debugging detail level is zero (0).
Digest Algorithm	The name of the `MessageDigest` algorithm used to encode user passwords stored in the database. If not specified, user passwords are assumed to be stored in clear text.
Role Base Element	The base directory element for performing role searches.
Role Name	The name of the directory attribute to retrieve when selecting the assigned roles for a user. If not specified, use the User Role Name attribute to specify the name of an attribute in the user's entry that contains zero or more role names assigned to this user.
Role Search Pattern	The LDAP search expression to use when selecting roles for a particular user, with {0} marking where the actual user name should be inserted. For more information on patterns, see Values for the Pattern Attribute (page 813).
Search Role Subtree	Set to True to search subtrees of the elements selected by the Role Search Pattern expression. Set to False to not search subtrees. The default value is False.
User Role Name	The name of a directory attribute in the user's entry containing zero or more values for the names of roles assigned to this user. If not specified, use the Role Name attribute to specify the name of a particular attribute that is retrieved from individual role entries associated with this user.
User Base	The entry that is the base of the subtree containing users. If not specified, the search base is the top-level context. This option is not used when User Pattern is specified.

Table A–12 JNDIRealm Attributes

Attribute	**Description**
Search User Subtree	Set to True if you are using the User Search Pattern to search for authenticated users and you want to search subtrees of the element specified by the User Base Element. The default value of False causes only the specified level to be searched. Not used if you are using the User Pattern expression.
User Password	Name of the LDAP element containing the user's password. If you specify this value, `JNDIRealm` will bind to the directory using the values specified by the Connection Name and Connection Password attributes and retrieve the corresponding attribute for comparison to the value specified by the user being authenticated. If you do not specify this value, `JNDIRealm` will attempt to bind to the directory using the user name and password specified by the user, with a successful bind being interpreted as an authenticated user.
User Pattern	The LDAP search expression to use when retrieving the attributes of a particular user, with `{0}` marking where the actual user name should be inserted. Use this attribute instead of User Search Pattern if you want to select a particular single entry based on the user name.
User Search	The LDAP search expression to use when retrieving the attributes of a particular user, with {0} marking where the actual user name should be inserted. Use this attribute instead of User Pattern to search the entire directory (instead of retrieving a particular named entry) under the optional additional control of the User Base Element and Search User Subtree attributes.

UserDatabaseRealm Attributes

`UserDatabaseRealm` is an implementation of Realm based on an implementation of `UserDatabase` made available through the global JNDI resources configured for the instance of Tomcat. The `resourceName` parameter is set to the global JNDI resources name for the configured instance of `UserDatabase` to be consulted. Attributes for the User Database Realm implementation are shown in Table A–13.

Table A–13 UserDataBaseRealm Attributes

Attribute	**Description**
Debug Level	The level of debugging detail logged by this Engine. Higher numbers generate more detailed output. If not specified, the default debugging detail level is zero (0).
Resource Name	The global JNDI resources name for the configured instance of `UserDatabase` to be consulted.

MemoryRealm Attributes

The Memory Based Realm is a simple Realm implementation that reads an XML file to configure valid users, passwords, and roles. The file format and default file location are identical to those currently supported by Tomcat 3.x. This implementation is intended solely to get up and running with container managed security - it is NOT intended for production use. As such, there are no mechanisms for updating the in-memory collection of users when the content of the underlying data file is changed. Attributes for the Memory Realm implementation are shown in Table A–14.

Table A–14 MemoryRealm Attributes

Attribute	**Description**
Debug Level	The level of debugging detail logged by this Engine. Higher numbers generate more detailed output. If not specified, the default debugging detail level is zero (0).
Path Name	The path to the XML file containing user information. The path is specified absolute or relative to *<JWSDP_HOME>*. If no path name is specified, the default value is *<JWSDP_HOME>*`/conf/tomcat-users.xml`.

Configuring Valve Elements

A Valve element represents a component that will be inserted into the request processing pipeline for Tomcat. Individual Valves have distinct processing capabilities, and are described individually below.

To edit a Valve,

1. Expand the Service element in the left pane.
2. Select the Valve to edit.
3. Edit the values in the right pane.
4. Select Save to save the changes for this session. Select Commit Changes to save the changes for when Tomcat is restarted.

To create a new Valve for a service,

1. Select the Service element in the left pane. It is highly recommended that you only modify the Java Web Services Developer Pack Service, or a service that you have created.
2. Select Create New Valve from the Available Actions list.
3. Enter the preferred values for the Valve. See Valve Attributes (page 812) for more information on the options.
4. Select Save to save the changes for this session. Select Commit Changes to save the changes for when Tomcat is restarted.

To learn more about Valves, read the document titled “Valve Component” at `<JWSDP_HOME>/docs/tomcat/config/valve.html`.

Valve Attributes

There are 5 types of Valves available in this release, and each has its own set of attributes, listed in the following sections.

AccessLogValve Attributes

The Access Log Valve creates log files in the same format as those created by standard Web servers. These logs can later be analyzed by standard log analysis tools to track page hit counts, user session activity, and so on. The Access Log Valve shares many of the configuration and behavior characteristics of the File Logger, including the automatic rollover of log files at midnight each night. An Access Log Valve can be associated with any Tomcat container, and will record

ALL requests processed by that container. Attributes for `AccessLogValve` are shown in Table A–15.

Table A–15 AccessLogValve Attributes

Attribute	**Description**
Debug Level	The level of debugging detail logged by this Logger. Higher numbers generate more detailed output. If not specified, the default debugging detail level is zero (0).
Directory	The absolute or relative path name of a directory in which log files created by this valve will be placed. If a relative path is specified, it is interpreted as relative to *`<JWSDP_HOME>`*. The default value is `logs` (relative to *`<JWSDP_HOME>`*).
Pattern	A formatting layout identifying the various information fields from the request and response to be logged, or the word `common` or `combined` to select a standard format. See Values for the Pattern Attribute (page 813) for more information.
Prefix	The prefix added to the start of each log file's name. The default value is `access_log`. To specify no prefix, use a zero-length string.
Resolve Hosts	Whether or not to convert the IP address of the remote host into the corresponding host name via a DNS lookup. Set to True to convert the IP address of the remote host into the corresponding host name via a DNS lookup. Set to False to skip this lookup, and report the remote IP address instead. The default is False.
Suffix	The suffix added to the end of each log file's name. If not specified, the default value is `""`. To specify no suffix, use a zero-length string.

Values for the Pattern Attribute

Values for the pattern attribute are made up of literal text strings, combined with pattern identifiers prefixed by the "%" character to cause replacement by the corresponding variable value from the current request and response. The following pattern codes are supported:

- `%a` - Remote IP address
- `%A` - Local IP address
- `%b` - Bytes sent, excluding HTTP headers, or '-' if zero
- `%B` - Bytes sent, excluding HTTP headers
- `%h` - Remote host name (or IP address if `resolveHosts` is false)
- `%H` - Request protocol
- `%l` - Remote logical user name from `identd` (always returns '-')
- `%m` - Request method (GET, POST, etc.)
- `%p` - Local port on which this request was received
- `%q` - Query string (prepended with a '?' if it exists)
- `%r` - First line of the request (method and request URI)
- `%s` - HTTP status code of the response
- `%S` - User session ID
- `%t` - Date and time, in Common Log Format
- `%u` - Remote user that was authenticated (if any), else '-'
- `%U` - Requested URL path
- `%v` - Local server name

The shorthand pattern name `common` (which is also the default) corresponds to `%h %l %u %t "%r" %s %b`. The shorthand pattern name `combined` appends the values of the Referrer and User-Agent headers, each in double quotes, to the `common` pattern.

RemoteAddrValve Attributes

Remote Address Valve allows you to compare the IP address of the client that submitted this request against one or more regular expressions, and either allow the request to continue or refuse to process the request from this client. A Remote Address Valve must accept any request presented to this container for processing before it will be passed on.

Attributes for this Valve are listed in Table A–16.

Table A–16 RemoteAddrValve Attributes

Attribute	**Description**
Allow IP Addresses	A comma-separated list of regular expression patterns that the remote client's IP address is compared to. If this attribute is specified, the remote address MUST match for this request to be accepted. If this attribute is not specified, all requests will be accepted UNLESS the remote address matches a deny pattern.
Deny IP Addresses	A comma-separated list of regular expression patterns that the remote client's IP address is compared to. If this attribute is specified, the remote address MUST NOT match for this request to be accepted. If this attribute is not specified, request acceptance is governed solely by the Allow IP Addresses attribute.

RemoteHostValve Attributes

The Remote Host Valve allows you to compare the host name of the client that submitted this request against one or more regular expressions, and either allow the request to continue or refuse to process the request from this client. A Remote Host Valve must accept any request presented to this container for processing before it will be passed on.

Attributes for the `RemoteHostValve` are outlined in Table A–17.

Table A–17 RemoteHostValve Attributes

Attribute	**Description**
Allow these Hosts	A comma-separated list of regular expression patterns that the remote client's host name is compared to. If this attribute is specified, the remote hostname MUST match for this request to be accepted. If this attribute is not specified, all requests will be accepted UNLESS the remote host name matches a deny pattern.

Table A–17 RemoteHostValve Attributes (Continued)

Attribute	Description
Deny these Hosts	A comma-separated list of regular expression patterns that the remote client's host name is compared to. If this attribute is specified, the remote host name MUST NOT match for this request to be accepted. If this attribute is not specified, request acceptance is governed solely by the Allow These Hosts attribute.

RequestDumperValve Attributes

The Request Dumper Valve is a useful tool in debugging interactions with a client application (or browser) that is sending HTTP requests to your Tomcat-based server. When configured, it causes details about each request processed by its associated Engine, Host, or Context to be logged to the Logger that corresponds to that container. This Valve has no specific attributes.

SingleSignOn Attributes

The Single Sign On Valve is utilized when you wish to give users the ability to sign on to any one of the Web applications associated with your virtual host, and then have their identity recognized by all other Web applications on the same virtual host. This Valve has a Debug Level attribute.

Configuring Resources

The Resources node represents the Global Naming Resources component. The elements under this node represent the global JNDI resources which are defined for the Server. The following resources can be used to configure the resource manager (or object factory) used to return objects when a Web application performs a JNDI lookup operation on the corresponding resource name:

- Data Sources
- Environment Entries
- User Databases

For more information on configuring Global Naming Resources, read the document titled *GlobalNamingResources Component,* available from `<`*`JWSDP_HOME`*`>/docs/tomcat/config/globalresources.html.`

Configuring Data Sources

Many Web applications need to access a database via a JDBC driver to support the functionality required by that application. The J2EE Platform Specification requires J2EE Application Servers to make a Data Source implementation (that is, a connection pool for JDBC connections) available for this purpose. Tomcat offers the same support so that database-based applications developed on Tomcat using this service will run unchanged on any J2EE server.

To edit a Data Source,

1. Expand the Resources element in the left pane.
2. Select the Data Source to edit.
3. Edit the values in the right pane.
4. Select Save to save the changes for this session. Select Commit Changes to save the changes for when Tomcat is restarted.

To create a new Data Source for Tomcat,

1. Create a `Context` in `server.xml` and add a `ResourceLink` within the context . The `global` field of the Resource Link should be the name of the Global Resource you wish to add using `admintool`. The name field of the `ResourceLink` should be the name you use as a Resource Reference in `web.xml`. The following example from a `server.xml` for Tomcat shows a simple Resource Reference and Context.

   ```
   <Context
     className="org.apache.catalina.core.StandardContext"
     cachingAllowed="true"charsetMapperClass="org.apache.
     catalina.util.CharsetMapper" cookies="true"
     crossContext="false" debug="0" displayName="GSApp"
     docBase="/home/your_name/work/Standard
       Engine\localhost\manager\gs.war"
     mapperClass="org.apache.catalina.core.
     StandardContextMapper" path="/GSApp"
     privileged="false"reloadable="false" useNaming="true"
     wrapperClass="org.apache.catalina.core.StandardWrapper">
     <ResourceLink global="jdbc/ActivityDB" name="ActivityDB"/>
   </Context>
   ```

2. Add the resource reference to the deployment descriptor for the application, `web.xml`. The following code, which shows an example of a resource reference in bold, is an example from the Getting Started application.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
'-//Sun Microsystems, Inc.//DTD Web Application 2.3//
EN' 'http://java.sun.com/dtd/web-app_2_3.dtd'>

<web-app>
  <display-name>GSApp</display-name>
  <servlet>
    <servlet-name>index</servlet-name>
    <display-name>index</display-name>
    <jsp-file>/index.jsp</jsp-file>
  </servlet>
  <resource-ref>
    <res-ref-name>ActivityDB</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
</web-app>
```

The Resource created here is linked to the context via the Resource Link element.

3. In `admintool`, select the Data Source element in the left pane.
4. Select Create New Data Source from the Available Actions list.
5. Set the Data Source attributes. See Data Source Attributes (page 820) for more information on the options. The JNDI Name you specify in `admintool` should match the Resource Reference name from the `web.xml` file and the Resource Link name from `server.xml`. Add the Driver Name, URL, User Name, and Password.
6. Select Save to save the changes for this session. Select Commit Changes to save the changes for when Tomcat is restarted.

If you select Commit Changes, the `<`*`JWSDP_HOME`*`>/conf/server.xml` file will be updated with an entry for `ResourceParams` within the `GlobalNamingResources` element, which might look like this:

```
<GlobalNamingResources>
  <Environment description="Absolute Path name of the JWSDP"
    Installation" name="jwsdp.home"
    override="true" type="java.lang.String"
    value="/home/your_name/jwsdp-1.1"/>
  <Resource auth="Container" description="Users and Groups
    Database" name="UserDatabase"
    scope="Shareable"
    type="org.apache.catalina.UserDatabase"/>
  <Resource name="jdbc/ActivityDB" scope="Shareable"
     type="javax.sql.DataSource"/>
  <ResourceParams name="UserDatabase">
    <parameter>
      <name>factory</name>
      <value>org.apache.catalina.users.
        MemoryUserDatabaseFactory</value>
    </parameter>
    <parameter>
      <name>pathname</name>
      <value>conf/tomcat-users.xml</value>
    </parameter>
  </ResourceParams>
  <ResourceParams name="jdbc/ActivityDB">
    <parameter>
      <name>validationQuery</name>
      <value></value>
    </parameter>
    <parameter>
      <name>user</name>
      <value>your_user_name</value>
    </parameter>
    <parameter>
      <name>maxWait</name>
      <value>5000</value>
    </parameter>
    <parameter>
      <name>maxActive</name>
      <value>4</value>
    </parameter>
    <parameter>
      <name>password</name>
      <value>your_password</value>
    </parameter>
    <parameter>
      <name>url</name>
      <value>jdbc:pointbase:server://localhost/ActivityDB</val
ue>
    </parameter>
    <parameter>
      <name>driverClassName</name>
      <value>com.pointbase.jdbc.jdbcUniversalDriver</value>
    </parameter>
    <parameter>
      <name>maxIdle</name>
```

```
            <value>2</value>
          </parameter>
        </ResourceParams>
      </GlobalNamingResources>
```

Data Source Attributes

Note: In order to use a Data Source, you must have a JDBC driver installed and configured.

The attributes outlined in Table A–18 may be viewed, set, or modified for a Data Source.

Table A–18 Data Source Attributes

Attribute	**Description**
JNDI Name	The JNDI name under which you will look up pre-configured data sources. By convention, all such names should resolve to the `jdbc` subcontext (relative to the standard `java:comp/env` naming context that is the root of all provided resource factories.) For example, this entry might look like `jdbc/EmployeeDB`.
Data Source URL	The connection URL to be passed to the JDBC driver. One example is `jdbc:HypersonicSQL:database`.
JDBC Driver Class	The fully-qualified Java class name of the JDBC driver to be used. One example is `org.hsql.jdbcDriver`.
User Name	The database user name to be passed to the JDBC driver.
Password	The database password to be passed to the JDBC driver.
Max. Active Connections	The maximum number of active instances that can be allocated from this pool at the same time. Default value is 4.

Table A–18 Data Source Attributes (Continued)

Attribute	Description
Max. Idle Connections	The maximum number of connections that can sit idle in this pool at the same time. Default value is 2.
Max. Wait for Connections	The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception. Default value is 5000.
Validation Query	A SQL query that can be used by the pool to validate connections before they are returned to the application. If specified, this query MUST be an SQL SELECT statement that returns at least one row.

Configuring Environment Entries

Use this element to configure or delete named values that will be made visible to Web applications as environment entry resources. An example of an environment entry that might be useful is the absolute path to the Java WSDP installation, which is already defined as an Environment Entry.

To edit an Environment Entry,

1. Expand the Resources element in the left pane.
2. Select Environment Entries in the left pane.
3. Select the Environment Entry to edit in the right pane. By default, an environment entry for the absolute path to the Java WSDP installation displays.
4. Edit the values in the right pane.
5. Select Save to save the changes for this session. Select Commit Changes to save the changes for when Tomcat is restarted.

To create a new Environment Entry for Tomcat,

1. Select the Environment Entries element in the left pane.
2. Select Create New Env Entry from the Available Actions list.
3. Set the Environment Entries attributes. See Environment Entries Attributes (page 822) for more information on the options.
4. Select Save to save the changes for this session. Select Commit Changes to save the changes for when Tomcat is restarted.

Environment Entries Attributes

The valid attributes for an Environment element are outlined in Table A–19.

Table A–19 Environment Entries Attributes

Attribute	Description
Name	The name of the environment entry to be created, relative to the `java:comp/env` context. For example, `jwsdp.home`.
Type	The fully qualified Java class name expected by the Web application for this environment entry: `java.lang.Boolean`, `java.lang.Byte`, `java.lang.Character`, `java.lang.Double`, `java.lang.Float`, `java.lang.Integer`, `java.lang.Long`, `java.lang.Short`, or `java.lang.String`.
Value	The parameter value that will be presented to the application when requested from the JNDI context. This value must be convertible to the Java type defined by the type attribute. For example, <*path_to_home_directory*>`/jwsdp-1_0`.
Override Application Level Entries	Whether or not you want an Environment Entry for the same environment entry name, found in the Web application's deployment descriptor, to override the value specified here. Unselect this option if you do not want an Environment Entry for the same environment entry name, found in the Web application's deployment descriptor, to override the value specified here. By default, overrides are allowed.
Description	An optional, human-readable description of this environment entry.

Configuring User Databases

Use this Resource to configure and edit a database of users for this server. The default database, <*JWSDP_HOME*>`/conf/tomcat-users.xml`, is already defined.

To edit a User Database,

1. Expand the Resources element in the left pane.
2. Select User Databases in the left pane.

3. Select the User Database to edit in the right pane. By default, a user database for Tomcat displays.
4. Edit the values in the right pane.
5. Select Save to save the changes for this session. Select Commit Changes to save the changes for when Tomcat is restarted.

To create a new User Database for Tomcat,

1. Select the User Databases element in the left pane.
2. Select Create New User Database from the Available Actions list.
3. Set the User Database attributes. See User Database Attributes (page 823) for more information on the options.
4. Select Save to save the changes for this session. Select Commit Changes to save the changes for when Tomcat is restarted.

User Database Attributes

Configure a User Database with the attributes outlined in Table A–20.

Table A–20 User Database Attributes

Attribute	Description
Name	The name of the user database to be created, for example, `UserDatabase`.
Location	The location where the user database should be created, for example, `conf/tomcat-users.xml`.
Factory	The type of factory to use for this database, `org.apache.catalina.users.MemoryUserDatabaseFactory`.
Description	A human-readable description of what type of data the database holds, for example, Users and Groups Database.

Administering Roles, Groups, and Users

Information about understanding, adding, removing, and editing roles, groups, and users can be found in the Web Application Security topic Users, Groups, and Roles (page 702).

Further Information

- *Tomcat Server Configuration Reference*. For further information on the elements that can be used to configure the behavior of Tomcat, read the Tomcat Configuration Reference, which can be found at `<`*`JWSDP_HOME`*`>/docs/tomcat/config/index.html`.
- *JNDI Resources How-To*. This document discusses configuring JNDI Resources, Tomcat Standard Resource Factories, JDBC Data Sources, and Custom Resource Factories. This document can be found at `<`*`JWSDP_HOME`*`>/docs/tomcat/jndi-resources-howto.html`.
- *Manager Application How-To*. This document describes using the Manager Application to deploy a new Web application, undeploy an existing application, or reload an existing application without having to shut down and restart Tomcat. This document can be found at `<`*`JWSDP_HOME`*`>/docs/tomcat/manager-howto.html`.
- *Proxy Support How-To*. This document discusses running behind a proxy server (or a web server that is configured to behave like a proxy server). In particular, this document discusses how to manage the values returned by the calls from Web applications that ask for the server name and port number to which the request was directed for processing. This document can be found at `<`*`JWSDP_HOME`*`>/docs/tomcat/proxy-howto.html`.
- *Realm Configuration How-To*. This document discusses how to configure Tomcat to support container-managed security by connecting to an existing database of user names, passwords, and user roles. This document can be found at `<`*`JWSDP_HOME`*`>/docs/tomcat/realm-howto.html`.
- *Security Manager How-To*. This document discusses the use of a `SecurityManager` while running Tomcat to protect your server from unauthorized servlets, JSPs, JSP beans, and tag libraries. This document can be found at `<`*`JWSDP_HOME`*`>/docs/tomcat/security-manager-howto.html`.
- *SSL Configuration How-To*. This document discusses how to install and configure SSL support on Tomcat. Configuring SSL support on Tomcat using Java WSDP is discussed in Installing and Configuring SSL Support (page 721). The Tomcat documentation at `<`*`JWSDP_HOME`*`>/docs/tomcat/ssl-howto.html` also discusses this topic, however, the information in this tutorial is more up-to-date for the version of Tomcat shipped with the Java WSDP.

B

Tomcat Web Application Manager

Stephanie Bodoff

THE Tomcat Web Application Manager is used to list, install, reload, deploy, and remove Web applications from Tomcat. The Tomcat Web Application Manager is referred to as `manager` throughout this section for ease of reference.

Running the Web Application Manager

The `manager` is itself a Web application that is preinstalled into Tomcat, so Tomcat must be running in order to use it. You invoke a `manager` command via one of the URLs listed in Table B–1.

Table B–1 Tomcat Web Application Manager Commands

Function	**Command**
list	`http://<host>:8080/manager/list`

Table B–1 Tomcat Web Application Manager Commands

Function	Command

install	`http://<host>:8080/manager/install?path=/mywebapp&war=file:/path/to/mywebapp`
	
	`http://<host>:8080/manager/install?path=/mywebapp&war=jar:file:/path/to/mywebapp.war!/`

reload	`http://<host>:8080/manager/reload?path=/mywebapp`
remove	`http://<host>:8080/manager/remove?path=/mywebapp`

Since the `manager` pages are protected Web resources, the first time you invoke a `manager` command, an authentication dialog will appear. You must log in to the `manager` with the user name and password you provided when you installed the Java WSDP.

The document *Manager App HOW-TO*, distributed with the Java WSDP at `<JWSDP_HOME>/docs/tomcat/manager-howto.html`, contains reference information about the manager application.

Running Manager Commands Using Ant Tasks

The version of `Ant` distributed with the Java WSDP supports tasks that invoke `manager` commands, thus allowing you to run the commands from a terminal window. The tasks are summarized in Table B–2.

Table B–2 Ant Web Application Manager Tasks

Function	Ant Task Syntax
list	`<list url="url" username="username" password="password" />`

Table B–2 Ant Web Application Manager Tasks

Function	**Ant Task Syntax**

install	`<install url="`*`url`*`" path="`*`mywebapp`*`" war=""`
	`  username="`*`username`*`" password="`*`password`*`" />`
	
	The value of the `war` attribute can be a WAR file (`jar:file:`*`/path/to/mywebapp`*`.war!/`) or an unpacked directory (`file:`*`/path/to/mywebapp`*`)`.

reload	`<reload url="`*`url`*`" path="`*`mywebapp`*`"`
	`  username="`*`username`*`" password="`*`password`*`" />`

deploy	`<deploy url="`*`url`*`" path="`*`mywebapp`*`"`
	`  war="file:`*`/path/to/mywebapp`*`.war"`
	`  username="`*`username`*`" password="`*`password`*`" />`

undeploy	`<undeploy url="`*`url`*`" path="`*`mywebapp`*`"`
	`  username="`*`username`*`" password="`*`password`*`" />`

remove	`<remove url="`*`url`*`" path="`*`mywebapp`*`"`
	`  username="`*`username`*`" password="`*`password`*`" />`

Note: An application that is installed is not available after Tomcat is restarted. To make an application permanently available, use the `deploy` task.

Since a user of the `manager` is required to be authenticated, the `Ant` tasks take `username` and `password` attributes in addition to the URL. Instead of embedding these in the `Ant` build file, you can use the approach followed by the tutorial examples. You set the `username` and `password` properties in a file named `build.properties` in your home directory as follows:

```
username=ManagerName
password=ManagerPassword
```

Replace *`ManagerName`* and *`ManagerPassword`* with the values you specified for the user name and password when you installed the Java WSDP.

Note: On Windows, your home directory is the directory where your Windows profile is stored. For example, on Windows 2000 it would be `C:\Documents and Settings\`*`yourProfile`*.

The Ant build files import these properties with the following element:

```
<property file="${user.home}/build.properties"/>
```

C

The Java WSDP Registry Server

Kim Haase

A registry offers a mechanism for humans or software applications to advertise and discover Web services. The Java Web Services Developer Pack (Java WSDP) Registry Server implements Version 2 of the Universal Description, Discovery and Integration (UDDI) project to provide a UDDI registry for Web services in a private environment. You can use it with the Java WSDP APIs as a test registry for Web services application development.

You can use the Registry Server to test applications that you develop that use the Java API for XML Registries (JAXR), described in Publishing and Discovering Web Services with JAXR (page 537). You can also use the JAXR Registry Browser sample application provided with the Java WSDP to perform queries and updates on Registry Server data; see Registry Browser (page 839) for details.

The release of the Registry Server that is part of the Java WSDP includes the following:

- A Web application, a servlet, that implements UDDI Version 2 functionality
- A database based on the native XML database Xindice, which is part of the Apache XML project. This database provides the persistent store for registry data.

The Registry Server does not support messages defined in the UDDI Version 2.0 Replication Specification.

This chapter describes how to start the Registry Server and how to use JAXR to access it. It also describes how to access the Registry Server using a command-line client script that is provided as a sample application.

Starting the Registry Server

In order to use the Java WSDP Registry Server, you must start Tomcat. Starting Tomcat automatically starts both the Registry Server and the Xindice database.

See Starting Tomcat (page 80) for information on how to start Tomcat.

See Shutting Down Tomcat (page 83) for information on how to stop Tomcat.

Using JAXR to Access the Registry Server

You can access the Registry Server by using the sample programs in the `<JWSDP_HOME>/docs/tutorial/examples/jaxr` directory. For details on how these examples work and how to run them, see Running the Client Examples (page 562).

Before you compile the examples, you need to edit the file `JAXRExamples.properties` as follows.

1. If necessary, edit the following lines in the `JAXRExamples.properties` file to specify the Registry Server. The default registry is the Registry Server, so if you are using the examples for the first time you do not need to perform this step. The lines should look something like this:

```
## Uncomment one pair of query and publish URLs.
## IBM:
#query.url=http://uddi.ibm.com/testregistry/inquiryapi
#publish.url=https://uddi.ibm.com/testregistry/protect/
publishapi
## Microsoft:
#query.url=http://uddi.microsoft.com/inquire
#publish.url=https://uddi.microsoft.com/publish
## Registry Server:
query.url=http://localhost:8080/RegistryServer
publish.url=http://localhost:8080/RegistryServer
```

If the Registry Server is running on a system other than your own, specify the fully qualified host name instead of `localhost`. Do not use `https:` for the `publishURL`.

2. If necessary, edit the following lines in the `JAXRExamples.properties` file to specify the user name and password you will be using. The default is the Registry Server default password:

```
## Specify username and password if needed
## testuser/testuser are defaults for Registry Server
registry.username=testuser
registry.password=testuser
```

3. You can leave the following lines in the `JAXRExamples.properties` file as they are. You do not use a proxy to access the Registry Server, so these values are not used. If you previously filled in the host values, you can leave them filled in.

```
## HTTP and HTTPS proxy host and port;
##   ignored by Registry Server
http.proxyHost=
http.proxyPort=8080
https.proxyHost=
https.proxyPort=8080
```

4. Feel free to change any of the organization data in the remainder of the file. This data is used by the publishing and postal address examples.

Using the Command Line Client Script to Access the Registry Server

You will find shell scripts in the `<`*`JWSDP_HOME`*`>/registry-server-1.0_04/samples/` directory called `registry-server-test.sh` (for UNIX systems) and `registry-server-test.bat` (for Microsoft Windows systems).

The script uses XML files in the `xml` subdirectory to send messages to the Registry Server.

To use the script, go to the directory where the script resides. Make sure the script is executable (make it so if it is not).

You can use the script to perform the following tasks:

- Obtaining authentication
- Saving a business
- Finding a business
- Obtaining business details
- Deleting a business
- Validating messages
- Retrieving a particular user's businesses
- Sending any kind of UDDI message
- Adding a new user to the registry
- Deleting a user from the registry

Obtaining Authentication

Before you can perform other tasks, you must obtain authentication as a user of the Registry Server.

To obtain authentication, you use the file `GetAuthToken.xml` in the `xml` subdirectory. By default, the registry accepts a default user named `testuser` with a password of `testuser`. To create other users, follow the instructions in Adding a New User to the Registry (page 836), then edit the `GetAuthToken.xml` file to specify the user name and password you created.

To obtain authentication, enter the following command on one line:

Windows:

```
registry-server-test run-cli-request
   -Drequest=xml\GetAuthToken.xml
```

UNIX:

```
registry-server-test.sh run-cli-request
   -Drequest=xml/GetAuthToken.xml
```

When the script runs, it returns an `<authToken>` tag that contains an `<authInfo>` tag. You will use the value in this tag in the next step.

The value in this tag is valid for one hour. You can rerun the script after it expires.

Saving a Business

To save (that is, to add) a business, you use the file `SaveBusiness.xml` in the `xml` subdirectory. Before you run the script, edit the `<authInfo>` tag in this file and replace the existing contents with the contents of the `<authInfo>` tag returned in the previous step. Feel free to modify other values specified in the file.

To save the business, enter the following command on one line:

Windows:

```
registry-server-test run-cli-request
   -Drequest=xml\SaveBusiness.xml
```

UNIX:

```
registry-server-test.sh run-cli-request
   -Drequest=xml/SaveBusiness.xml
```

Output appears in the terminal window in which you run the command.

Finding a Business

To find a business by name, you use the file `FindBusiness.xml` in the `xml` subdirectory.

Before you run the script this time, edit the file by changing the value in the `<name>` tag to the name you specified in the `SaveBusiness.xml` file (or the first few characters of the name).

To find the business, use the following command:

Windows:

```
registry-server-test run-cli-request
   -Drequest=xml\FindBusiness.xml
```

UNIX:

```
registry-server-test.sh run-cli-request
   -Drequest=xml/FindBusiness.xml
```

Output appears in the terminal window. Notice the `businessKey` value returned in the `<businessEntity>` tag. You will use it in the next step.

Obtaining Business Details

To obtain details about a business, you use the file `GetBusinessDetail.xml` in the `xml` subdirectory.

Before you run the script this time, edit this file by copying the `businessKey` value from the output of the command in the previous step into the `<businessKey>` tag.

To obtain details about the business you saved, use the following command:

Windows:

```
registry-server-test run-cli-request
   -Drequest=xml\GetBusinessDetail.xml
```

UNIX:

```
registry-server-test.sh run-cli-request
   -Drequest=xml/GetBusinessDetail.xml
```

Output appears in the terminal window.

Deleting a Business

To delete a business you saved, you use the file `DeleteBusiness.xml` in the `xml` subdirectory.

Before you run the script this time, edit the file as follows:

1. Change the value of the `<authInfo>` tag to the value you used for `SaveBusiness.xml`.
2. Change the value of the `<businessKey>` tag to the business key value of the business you want to delete.

To delete the business, use the following command:

Windows:

```
registry-server-test run-cli-request
   -Drequest=xml\DeleteBusiness.xml
```

UNIX:

```
registry-server-test.sh run-cli-request
   -Drequest=xml/DeleteBusiness.xml
```

Validating UDDI Messages

To validate a UDDI message against the UDDI V2.0 XML schema before you send it, use the following command:

Windows:

```
registry-server-test run-validate -Dinstance=XML_file_name
```

UNIX:

```
registry-server-test.sh run-validate -Dinstance=XML_file_name
```

If a file contains errors, the error messages have the following format:

```
file:line:column:message
```

Retrieving a User's Businesses

To obtain a summary of all items published by a user, you use the file `GetRegisteredInfo.xml` in the `xml` subdirectory.

Before you run the script this time, edit this file by copying the `<authInfo>` string that you entered in the `SaveBusiness.xml` or `DeleteBusiness.xml` file into the `<authInfo>` tag of `GetRegisteredInfo.xml`.

To obtain details about the business you saved, use the following command:

Windows:

```
registry-server-test run-cli-request
   -Drequest=xml\GetRegisteredInfo.xml
```

UNIX:

```
registry-server-test.sh run-cli-request
   -Drequest=xml/GetRegisteredInfo.xml
```

The command returns information not only about your business, but about all other businesses and Tmodels owned by `testuser`.

Sending UDDI Request Messages

To send any UDDI request to the server, use the following command:

Windows:

```
registry-server-test run-cli-request -Drequest=name_of_file
```

UNIX:

```
registry-server-test.sh run-cli-request -Drequest=name_of_file
```

where *`name_of_file`* is an XML file containing a UDDI message. It is a good idea to validate the message before you send it.

The `xml` subdirectory contains numerous messages you can edit and use in addition to those described here. You can also create your own messages.

Adding a New User to the Registry

To add a new user to the Registry Server database, you first generate a hash password for the user. Then you use the file `UserInfo.xml` in the `xml` subdirectory. Perform the following steps:

1. To generate a hash password for the user, specify the actual password as the *`value`* argument in the following command line:

   Windows:

   ```
   registry-server-test run-md5 -Dpassword=value
   ```

   UNIX:

   ```
   registry-server-test.sh run-md5 -Dpassword=value
   ```

   For example, if you specify a password value of `mypass`, you get output like the following:

   ```
   registry-server-test run-md5 -Dpassword=mypass
   Buildfile:
   D:\jwsdp-1.1\registry-server-1.0_04\samples\test-build.xml

   run-md5:
   ```

```
[echo] -- Running md5 for auth --
[java]
[java]  The Value of the MD5 Hash is: a029d0df84eb5549
```

2. Open the file `xml/UserInfo.xml` in an editor.
3. Change the values in the `<fname>`, `<lname>`, and `<uid>` tags to the first name, last name, and unique user ID (UID) of the new user. The `<uid>` tag is commonly the user's login name. It must be unique.
4. Enter the hash value from the `run-md5` command as the value of the `<passwd>` tag in `UserInfo.xml`. Do not modify the `<tokenExpiration>` or `<authInfo>` tag.
5. Save and close the `UserInfo.xml` file.
6. Enter the following command (all on one line):

   Windows:

   ```
   registry-server-test run-cli-request
        -Drequest=xml\UserInfo.xml
   ```

   UNIX:

   ```
   registry-server-test.sh run-cli-request
        -Drequest=xml/UserInfo.xml
   ```

Deleting a User from the Registry

To delete a user from the registry, you use the file `UserDelete.xml` in the `xml` subdirectory.

Before you run the script this time, edit this file by modifying the values in the `<fname>`, `<lname>`, `<uid>`, and `<passwd>` tags.

To delete the user, use the following command:

Windows:

```
registry-server-test run-cli-request
   -Drequest=xml\UserDelete.xml
```

UNIX:

```
registry-server-test.sh run-cli-request
   -Drequest=xml/UserDelete.xml
```

Further Information

For more information about UDDI registries, JAXR, and Web services, see the following:

- Universal Description, Discovery, and Integration (UDDI) project:
  `http://www.uddi.org/`
- JAXR home page:
  `http://java.sun.com/xml/jaxr/index.html`
- Java Web Services Developer Pack (Java WSDP):
  `http://java.sun.com/webservices/webservicespack.html`
- Java Technology and XML:
  `http://java.sun.com/xml/`
- Java Technology & Web Services:
  `http://java.sun.com/webservices/index.html`

D Registry Browser

Kim Haase

THE Registry Browser is both a working example of a JAXR client and a simple GUI tool that enables you to search registries and submit data to them.

The Registry Browser source code is in the directory `<JWSDP_HOME>/jaxr-1.0_03/samples/jaxr-browser`. Much of the source code implements the GUI. The JAXR code is in the file `JAXRClient.java`.

The Registry Browser allows access to any registry, but includes as preset URLs the IBM and Microsoft UDDI test registries. You may also use the Registry Server (see The Java WSDP Registry Server, page 829).

Starting the Browser

To start the browser, go to the directory `<JWSDP_HOME>/jaxr-1.0_03/bin` or place this directory in your path.

The following commands show how to start the browser on a UNIX system and a Microsoft Windows system, respectively:

```
jaxr-browser.sh

jaxr-browser
```

In order to access the Registry Server through the browser, you must make sure to start Tomcat before you perform any queries or submissions to the browser; see Starting the Registry Server (page 830) for details.

In order to access external registries, the browser needs to know your Web proxy settings. By default, the browser uses the settings you specified when you installed the Java WSDP. These are defined in the file `<JWSDP_HOME>/conf/jwsdp.properties`. If you want to override these settings, you can edit this file or specify proxy information on the browser command line.

To use the same proxy server for both HTTP and HTTPS access, specify a non-default proxy host and proxy port as follows. The port is usually 8080. The following command shows how to start the browser on a UNIX system:

```
jaxr-browser.sh httpHost httpPort
```

For example, if your proxy host is named `websys` and it is in the `south` subdomain, you would enter

```
jaxr-browser.sh websys.south 8080
```

To use different proxy servers for HTTP and HTTPS access, specify the hosts and ports as follows. (If you do not know whether you need two different servers, specify just one. It is relatively uncommon to need two.) On a Microsoft Windows system, the syntax is as follows:

```
jaxr-browser httpHost httpPort httpsHost httpsPort
```

After the browser starts, enter the URL of the registry you want to use in the Registry Location combo box, or select a URL from the drop-down menu in the combo box. The menu allows you to choose among the IBM and Microsoft registries. To access the Registry Server, you normally enter the following URL in the Registry Location combo box:

```
http://localhost:8080/RegistryServer
```

Specify `localhost` if the Registry Server is on your own system. Otherwise, specify the fully qualified hostname of the system where the Registry Server is running. You specify the same URL for both queries and updates.

There may be a delay of a few seconds while a busy cursor is visible.

When the busy cursor disappears, you have a connection to the URL. However, you do not establish a connection to the registry itself until you perform a query or update, so JAXR will not report an invalid URL until then.

The browser contains two main panes, Browse and Submissions.

Querying a Registry

You use the Browse pane to query a registry.

Note: In order to perform queries on the Microsoft registry, you must be connected to the `inquire` URL. To perform queries on the IBM registry, you may be connected to either the `inquiryapi` URL or the `publishapi` URL.

Querying by Name

To search for organizations by name, perform the following steps.

1. Click the Browse tab if it is not already selected.
2. In the Find By panel on the left side of the Registry Browser window, do the following:
   a. Select Name in the Find By combo box if it is not already selected.
   b. Enter a string in the text field.
   c. Press Enter or click the Search button in the toolbar.

After a few seconds, the organizations whose names match the text string appear in the right side of the Registry Browser window. An informational dialog box appears if no matching organizations are found.

Queries are not case-sensitive. If you enter a plain text string (*`string`*), organization names match if they *begin* with the text string you entered. Enclose the string in percent signs (`%`*`string`*`%`) for wildcard searches.

Double-click on an organization to show its details. An Organization dialog box appears. In this dialog box, you can click Show Services to display the Services dialog box for the organization. In the Services dialog box, you can click Show ServiceBindings to display the ServiceBindings dialog box for that service.

Querying by Classification

To query a registry by classification, perform the following steps.

1. Select Classification in the Find By combo box.
2. In the Classifications pane that appears below the combo box, double-click a classification scheme.

3. Continue to double-click until you reach the node you want to search on.
4. Click the Search button in the toolbar.

After a few seconds, one or more organizations in the chosen classification may appear in the right side of the Registry Browser window. An informational dialog box appears if no matching organizations are found.

Managing Registry Data

You use the Submissions pane to add organizations to the registry.

To go to the Submissions pane, click the Submissions tab.

Adding an Organization

To add an organization, use the Organization panel on the left side of the Submissions pane.

Use the Organization Information fields as follows:

- Name: Enter the name of the organization.
- Id: You cannot enter or modify data in this field; the ID value is returned by the registry when you submit the data.
- Description: Enter a description of the organization.

Use the Primary Contact Information fields as follows:

- Name: Enter the name of the primary contact person for the organization.
- Phone: Enter the primary contact's phone number.
- Email: Enter the primary contact's email address.

Note: With the Registry Server, none of these fields is required; it is possible (though not advisable) to add an organization that has no data. With the IBM and Microsoft registries, an organization must have a name.

For information on adding or removing classifications, see Adding and Removing Classifications (page 844).

Adding Services to an Organization

To add information about an organization's services, Use the Services panel on the right side of the Submissions pane.

To add a service, click the Add Services button in the toolbar. A subpanel for the service appears in the Services panel. Click the Add Services button more than once to add more services in the Services panel.

Each service subpanel has the following components:

- Name, Id, and Description fields
- Edit Bindings and Remove Service buttons
- A Classifications panel

Use these components as follows:

- Name field: Enter a name for the service.
- Id field: You cannot enter or modify data in this field for a level 0 JAXR provider.
- Description field: Enter a description of the service.
- Click the Edit Bindings button to add service bindings for the service. An Edit ServiceBindings dialog box appears. See the next section, Adding Service Bindings to a Service, for details.
- Click the Remove Service button to remove this service from the organization. The service subpanel disappears from the Services panel.
- To add or remove classifications, use the Classifications panel. See Adding and Removing Classifications (page 844) for details.

Adding Service Bindings to a Service

To add service bindings for a service, click the Edit Bindings button in a service subpanel in the Submissions pane. The Edit ServiceBindings dialog box appears.

If there are no existing service bindings when the dialog box first appears, it contains an empty Service Bindings panel and two buttons, Add Binding and Done. If the service already has service bindings, the Service Bindings panel contains a subpanel for each service binding.

Click Add Binding to add a service binding. Click Add Binding more than once to add multiple service bindings.

After you click Add Binding, a new service binding subpanel appears. It contains three text fields and a Remove Binding button.

Use the text fields as follows:

- Description: Enter a description of the service binding.
- Access URI: Enter the URI used to access the service. The URI must be valid; if it is not, the submission will fail.

Use the Remove Binding button to remove the service binding from the service.

Click Done to close the dialog box when you have finished adding or removing service bindings.

Adding and Removing Classifications

To add classifications to, or remove classifications from, an organization or service, use a Classifications panel. A Classifications panel appears in an Organization panel or service subpanel.

To add a classification:

1. Click Add.
2. In the Select Classifications dialog, double-click one of the classification schemes.
    - If you clicked ntis-gov:naics:1997 or unspsc-org:unspsc:3-1, you can add the classification at any level of the taxonomy hierarchy. When you reach the level you want, click Add.
    - If you clicked uddi-org:iso-ch:3166:1999 (geography), locate the appropriate leaf node (the country) and click Add.

The classification appears in a table in the Classifications panel below the buttons.

To add multiple classifications to the organization or service, you can repeat these steps more than once. Alternatively, you can click on the classification schemes while pressing the control or shift key, then click Add.

Click Close to dismiss the window when you have finished.

To remove a classification, select the appropriate table row in the Classifications panel and click Remove. The classification disappears from the table.

Submitting the Data

When you have finished entering the data you want to add, click the Submit button in the toolbar.

An authentication dialog box appears. To continue with the submission, enter your user name and password and click OK. To close the window without submitting the data, click Cancel.

If you are using the Registry Server, the default username and password are both `testuser`.

If the submission is successful, an information dialog box appears with the organization key in it. Click OK to continue. The organization key also appears in the ID field of the Submissions pane.

Note: If you submit an organization, return to the Browse pane, then return to the Submissions pane, you will find that the organization is still there. If you click the Submit button again, a new organization is created, whether or not you modify the organization data.

Deleting an Organization

To delete an organization:

1. Use the Browse pane to locate an organization you wish to delete.
2. Connect to a URL that allows you to publish data. If you were previously using a URL that only allows queries, change the URL to the publish URL.
3. Right-click on the organization and choose Delete RegistryObject from the pop-up menu.
4. In the authentication dialog box that appears, enter your user name and password and click OK. To close the window without deleting the organization, click Cancel.

Stopping the Browser

To stop the Registry Browser, choose Exit from the File menu.

E

Provider Administration Tool

Maydene Fisher

THE Provider Administration tool is a convenient means of configuring a messaging provider. A messaging provider, a third party service, handles the behind-the-scenes details of the routing and transmission of JAXM messages. The JAXM tutorial gives more information about messaging providers in the section Messaging Providers (page 492).

Note that Tomcat must be running in order to use the Provider Administration tool. Follow these steps to use it:

1. With Tomcat running, set your browser window to

   ```
   http://localhost:8080/index.html
   ```

2. Click on the link “JAXM Provider Administration Tool”

   A window will come up with text boxes for your user name and password. Enter the same user name and password you supplied to the installation wizard when you installed this release of the Java WSDP.

3. Follow the instructions given on the page that comes up

The Provider Administration tool is normally used by System Administrators, but others may use it as well. Exploring this tool gives you more of an idea of what a messaging provider needs to know. For example, a messaging provider maintains a list of the endpoints to which you can send messages. You can add a new endpoint to this list using the Provider Administration tool. If a message is not delivered successfully on the first try, a messaging provider will continue attempting to deliver it. You can specify the number of times the messaging provider should attempt delivery by supplying a retry limit. Setting this limit is another thing you can do with the Provider Administration tool.

The following lists the ways you can use the tool to set a messaging provider's properties.

- To add, modify, or delete an endpoint
- To change the number of retries (the number of times the provider will try to send a message)
- To change the retry interval (the amount of time the provider will wait before trying to send a message again)
- To change the directory where the provider logs messages
- To set the number of messages per log file

F

HTTP Overview

Stephanie Bodoff

MOST Web clients use the HTTP protocol to communicate with a J2EE server. HTTP defines the requests that a client can send to a server and responses that the server can send in reply. Each request contains a URL, which is a string that identifies a Web component or a static object such as an HTML page or image file.

The J2EE server converts an HTTP request to an HTTP request object and delivers it to the Web component identified by the request URL. The Web component fills in an HTTP response object, which the server converts to an HTTP response and sends to the client.

This appendix provides some introductory material on the HTTP protocol. For further information on this protocol, see the Internet RFCs: HTTP/1.0 - RFC 1945, HTTP/1.1 - RFC 2616, which can be downloaded from

```
http://www.rfc-editor.org/rfc.html
```

HTTP Requests

An HTTP request consists of a request method, a request URL, header fields, and a body. HTTP 1.1 defines the following request methods:

- `GET` - retrieves the resource identified by the request URL.
- `HEAD` - returns the headers identified by the request URL.
- `POST` - sends data of unlimited length to the Web server.
- `PUT` - stores a resource under the request URL.
- `DELETE` - removes the resource identified by the request URL.
- `OPTIONS` - returns the HTTP methods the server supports.
- `TRACE` - returns the header fields sent with the TRACE request.

HTTP 1.0 includes only the GET, HEAD, and POST methods. Although J2EE servers are only required to support HTTP 1.0, in practice many servers, including the Java WSDP, support HTTP 1.1.

HTTP Responses

An HTTP response contains a result code, header fields, and a body.

The HTTP protocol expects the result code and all header fields to be returned before any body content.

Some commonly used status codes include:

- `404` - indicates that the requested resource is not available.
- `401` - indicates that the request requires HTTP authentication.
- `500` - indicates an error inside the HTTP server which prevented it from fulfilling the request.
- `503` - indicates that the HTTP server is temporarily overloaded, and unable to handle the request.

G

Java Encoding Schemes

This appendix describes the character-encoding schemes that are supported by the Java platform.

US-ASCII
US-ASCII is a 7-bit encoding scheme that covers the English-language alphabet. It is not large enough to cover the characters used in other languages, however, so it is not very useful for internationalization.

ISO-8859-1
This is the character set for Western European languages. It's an 8-bit encoding scheme in which every encoded character takes exactly 8-bits. (With the remaining character sets, on the other hand, some codes are reserved to signal the start of a multi-byte character.)

UTF-8
UTF-8 is an 8-bit encoding scheme. Characters from the English-language alphabet are all encoded using an 8-bit bytes. Characters for other languages are encoded using 2, 3 or even 4 bytes. UTF-8 therefore produces compact documents for the English language, but for other languages, documents tend to be half again as large as they would be if they used UTF-16. If the majority of a document's text is in a Western European language, then UTF-8 is generally a good choice because it allows for internationalization while still minimizing the space required for encoding.

UTF-16
UTF-16 is a 16-bit encoding scheme. It is large enough to encode all the characters from all the alphabets in the world. It uses 16-bits for most characters, but includes 32-bit characters for ideogram-based languages like Chinese. A Western European-language document that uses UTF-16 will be

twice as large as the same document encoded using UTF-8. But documents written in far Eastern languages will be far smaller using UTF-16.

Note: UTF-16 depends on the system's byte-ordering conventions. Although in most systems, high-order bytes follow low-order bytes in a 16-bit or 32-bit "word", some systems use the reverse order. UTF-16 documents cannot be interchanged between such systems without a conversion.

Further Information

For a complete list of the encodings that can be supported by the Java 2 platform, see:

http://java.sun.com/j2se/1.3/docs/guide/intl/encoding.doc.html

Glossary

A *B C D E F G H I J K L M N O P Q R S T U V W X Y Z*

access control
The methods by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints.

ACID
The acronym for the four properties guaranteed by transactions: atomicity, consistency, isolation, and durability.

admintool
A tool used to manipulate Tomcat while it is running.

anonymous access
Accessing a resource without authentication.

Ant
A Java-based, and thus cross-platform, build tool that can be extended using Java classes. The configuration files are XML-based, calling out a target tree where various tasks get executed.

Apache Software Foundation
Through the Jakarta Project, creates and maintains open source solutions on the Java platform for distribution to the public at no charge. Tomcat and Ant are two products developed by Apache and provided with the Java Web Services Developer Pack.

applet
A component that typically executes in a Web browser, but can execute in a variety of other applications or devices that support the applet programming model.

Application Deployment Tool
A tool for creating WAR files for application deployment and handling security issues.

archiving
Saving the state of an object and restoring it.

attribute
A qualifier on an XML tag that provides additional information.

authentication
The process that verifies the identity of a user, device, or other entity in a computer system, usually as a prerequisite to allowing access to resources in a system. Java WSDP requires three types of authentication: *basic*, *form-based,* and *mutual*, and supports *digest* authentication.

authorization
The process by which access to a method or resource is determined. Authorization depends upon the determination of whether the principal associated with a request through authentication is in a given security role. A security role is a logical grouping of users defined by the person who assembles the application. A deployer maps security roles to security identities. Security identities may be principals or groups in the operational environment.

authorization constraint
An authorization rule that determines who is permitted to access a Web resource collection.

B2B
Business-to-business.

basic authentication
An authentication mechanism in which a Web server authenticates an entity with a user name and password obtained using the Web application's built-in authentication mechanism.

binary entity
See *unparsed entity.*

binding
Construction of the code needed to process a well-defined bit of XML data.

binding compiler
A compiler that transforms, or binds, a source XML schema and optional customizing binding declarations to a set of Java content classes.

binding declarations

By default, the JAXB binding compiler binds Java classes and packages to a source XML schema based on rules defined in the *JAXB Specification*. In most cases, the default binding rules are sufficient to generate a robust set of schema-derived classes from a wide range of schemas. There may be times, however, when the default binding rules are not sufficient for your needs. JAXB supports customizations and overrides to the default binding rules by means binding declarations made inline in a source schema.

binding framework

A runtime API that provides interfaces for unmarshalling, marshalling, and validating XML content in a Java application.

build file

The XML file that contains one project that contains one or more targets. A target is a set of tasks you want to be executed. When starting Ant, you can select which target(s) you want to have executed. When no target is given, the project's default is used.

build properties file

A file named `build.properties` that contains properties in

business logic

The code that implements the functionality of an application.

A B C ***D E F G H I J K L M N O P Q R S T U V W X Y Z***

callback methods

Component methods called by the container to notify the component of important events in its life cycle.

CDATA

A predefined XML tag for Character DATA that means don't interpret these characters, as opposed to Parsed Character Data (PCDATA), in which the normal rules of XML syntax apply (for example, angle brackets demarcate XML tags, tags define XML elements, etc.). CDATA sections are typically used to show examples of XML syntax.

certificate authority

A trusted organization that issues public key certificates and provides identification to the bearer.

client certificate authentication

An authentication mechanism that uses HTTP over SSL, in which the server and, optionally, the client authenticate each other with a public key certificate that conforms to a standard that is defined by X.509 Public Key Infrastructure (PKI).

comment

Text in an XML document that is ignored, unless the parser is specifically told to recognize it.

commit

The point in a transaction when all updates to any resources involved in the transaction are made permanent.

component

An application-level software unit supported by a *container*. Components are configurable at deployment time. See also *Web components*.

component contract

The contract between a component and its container. The contract includes: life cycle management of the component, a context interface that the instance uses to obtain various information and services from its container, and a list of services that every container must provide for its components.

component-managed sign-on

Security information needed for signing on to the resource to the `getConnection()` method is provided by an application component.

connection

See *resource manager connection*.

connection factory

See *resource manager connection factory.*

connector

A standard extension mechanism for containers to provide connectivity to enterprise information systems. A connector is specific to an enterprise information system and consists of a resource adapter and application development tools for enterprise information system connectivity. The resource adapter is plugged in to a container through its support for system-level contracts defined in the connector architecture.

Connector element

A representation of the interface between external clients sending requests to a particular service.

container

An entity that provides life cycle management, security, deployment, and runtime services to *components*.

container-managed sign-on
: Security information needed for signing on to the resource to the `getConnection()` method is supplied by the container.

content
: The part of an XML document that occurs after the prolog, including the root element and everything it contains.

content tree
: An XML document is marshalled into a tree of Java objects. The objects in a content tree are manipulated by means of the schema-derived JAXB classes, so that programmers are able to work with XML data as Java objects rather than XML text.

context attribute
: An object bound into the context associated with a servlet.

Context element
: A representation of a Web application that is run within a particular virtual host.

context root
: A name that gets mapped to the *document root* of a Web application.

credentials
: The information describing the security attributes of a *principal*.

CSS
: Cascading Style Sheet. A stylesheet used with HTML and XML documents to add a style to all elements marked with a particular tag, for the direction of browsers or other presentation mechanisms.

data
: The contents of an element, generally used when the element does not contain any subelements. When it does, the more general term content is generally used. When the only text in an XML structure is contained in simple elements, and elements that have subelements have little or no data mixed in, then that structure is often thought of as XML data, as opposed to an XML document.

data binding
: An XML data-binding facility contains a binding compiler that binds components of a source schema to schema-derived Java content classes. Each class provides access to the content of the corresponding schema component via a set of JavaBeans-style access (i.e., `get` and `set`) methods. Binding dec-

larations provides a capability to customize the binding from schema components to Java representation. Such a facility also provides a binding framework, a runtime API that, in conjunction with the derived classes, supports unmarshal, marshal, and validate operations.

document
In general, an XML structure in which one or more elements contains text intermixed with subelements. See also *data*.

DDP
Document-Driven Programming. The use of XML to define applications.

declaration
The very first thing in an XML document, which declares it as XML. The minimal declaration is <?xml version="1.0"?>. The declaration is part of the document *prolog*.

declarative security
Mechanisms used in an application that are expressed in a declarative syntax in a deployment descriptor.

delegation
An act whereby one *principal* authorizes another principal to use its identity or privileges with some restrictions.

deploy task
A Tomcat manager application task. Requires a WAR, but not necessarily on the same server. Uploads the WAR to Tomcat, which then unpacks it into the `<`*`JWSDP_HOME`*`>/webapps` directory and loads the application. Useful when you want to deploy an application into a running production server. Restarts of Tomcat will remember that the application exists because it exists in the `/webapps` directory.

deployment
The process whereby software is installed into an operational environment.

deployment descriptor
An XML file provided with each module and application that describes how they should be deployed. The deployment descriptor directs a deployment tool to deploy a module or application with specific container options and describes specific configuration requirements that a deployer must resolve.

digest authentication
An authentication mechanism in which a Web application authenticates to a Web server by sending the server a message digest along its HTTP request message. The digest is computed by employing a one-way hash algorithm to a concatenation of the HTTP request message and the client’s password. The

digest is typically much smaller than the HTTP request, and doesn't contain the password.

distributed application

An application made up of distinct components running in separate runtime environments, usually on different platforms connected via a network. Typical distributed applications are two-tier (client-server), three-tier (client-middleware-server), and multitier (client-multiple middleware-multiple servers).

document root

The top-level directory of a *WAR*. The document root is where JSP pages, client-side classes and archives, and static Web resources are stored.

DOM

The Document Object Model. An API for accessing and manipulating XML documents as tree structures. DOM provides platform-neutral, language-neutral interfaces that enables programs and scripts to dynamically access and modify content and structure in XML documents.

DTD

Document Type Definition. An optional part of the document prolog, as specified by the *XML* standard. The DTD specifies constraints on the valid tags and tag sequences that can be in the document. The DTD has a number of shortcomings however, which has led to various schema proposals. For example, the DTD entry <!ELEMENT username (#PCDATA)> says that the XML element called username contains Parsed Character DATA— that is, text alone, with no other structural elements under it. The DTD includes both the local subset, defined in the current file, and the external subset, which consists of the definitions contained in external .dtd files that are referenced in the local subset using a parameter entity.

ebXML

Electronic Business XML. A group of specifications designed to enable enterprises to conduct business through the exchange of XML-based messages. It is sponsored by OASIS and the United Nations Centre for the Facilitation of Procedures and Practices in Administration, Commerce and Transport (U.N./CEFACT).

element

A unit of XML data, delimited by tags. An XML element can enclose other elements.

empty tag

A tag that does not enclose any content.

enterprise bean

A component that implements a business task or business entity and resides in an EJB container; either an *entity bean*, *session bean*, or *message-driven bean*.

enterprise information system

The applications that comprise an enterprise's existing system for handling company-wide information. These applications provide an information infrastructure for an enterprise. An enterprise information system offers a well defined set of services to its clients. These services are exposed to clients as local and/or remote interfaces. Examples of enterprise information systems include: enterprise resource planning systems, mainframe transaction processing systems, and legacy database systems.

enterprise information system resource

An entity that provides enterprise information system-specific functionality to its clients. Examples are: a record or set of records in a database system, a business object in an enterprise resource planning system, and a transaction program in a transaction processing system.

entity

A distinct, individual item that can be included in an XML document by referencing it. Such an entity reference can name an entity as small as a character (for example, "&lt;", which references the less-than symbol, or left-angle bracket (<). An entity reference can also reference an entire document, or external entity, or a collection of DTD definitions (a parameter entity).

entity bean

An enterprise bean that represents persistent data maintained in a database. An entity bean can manage its own persistence or can delegate this function to its container. An entity bean is identified by a primary key. If the container in which an entity bean is hosted crashes, the entity bean, its primary key, and any remote references survive the crash.

entity reference

A reference to an entity that is substituted for the reference when the XML document is parsed. It may reference a predefined entity like &lt; or it may reference one that is defined in the DTD. In the XML data, the reference could be to an entity that is defined in the local subset of the DTD or to an external XML file (an external entity). The DTD can also carve out a segment of DTD specifications and give it a name so that it can be reused (included) at multiple points in the DTD by defining a parameter entity.

error

A SAX parsing error is generally a validation error—in other words, it occurs when an XML document is not valid, although it can also occur if the declaration specifies an XML version that the parser cannot handle. See also: *fatal error, warning*.

Extensible Markup Language

A markup language that makes data portable.

external entity

An entity that exists as an external XML file, which is included in the XML document using an *entity reference*.

external subset

That part of the DTD that is defined by references to external .dtd files.

A B C D E* F *G H I J K L M N O P Q R S T U V W X Y Z

fatal error

A fatal error occurs in the SAX parser when a document is not well formed, or otherwise cannot be processed. See also: *error, warning*.

filter

An object that can transform the header and/or content of a request or response. Filters differ from *Web components* in that they usually do not themselves create responses but rather they modify or adapt the requests for a resource, and modify or adapt responses from a resource. A filter should not have any dependencies on a Web resource for which it is acting as a filter so that it can be composable with more than one type of Web resource.

filter chain

A concatenation of XSLT transformations in which the output of one transformation becomes the input of the next.

form-based authentication

An authentication mechanism in which a Web container provides an application-specific form for logging in. This form of authentication uses Base64 encoding and can expose user names and passwords unless all connections are over SSL.

A B C D E F G *H I J K L M N O P Q R S T U V W X Y Z*

general entity

An entity that is referenced as part of an XML document's content, as distinct from a parameter entity, which is referenced in the DTD. A general entity can be a parsed entity or an unparsed entity.

group

An authenticated set of users classified by common traits such as job title or customer profile. Groups are also associated with a set of roles, and every user that is a member of a group inherits all of the roles assigned to that group.

A B C D E F G H *I J K L M N O P Q R S T U V W X Y Z*

Host element

A representation of a virtual host.

HTML

Hypertext Markup Language. A markup language for hypertext documents on the Internet. HTML enables the embedding of images, sounds, video streams, form fields, references to other objects with URLs and basic text formatting.

HTTP

Hypertext Transfer Protocol. The Internet protocol used to fetch hypertext objects from remote hosts. HTTP messages consist of requests from client to server and responses from server to client.

HTTPS

HTTP layered over the SSL protocol.

A B C D E F G H I *J K L M N O P Q R S T U V W X Y Z*

impersonation

An act whereby one entity assumes the identity and privileges of another entity without restrictions and without any indication visible to the recipients of the impersonator’s calls that delegation has taken place. Impersonation is a case of simple *delegation*.

initialization parameter

A parameter that initializes the context associated with a servlet.

install task
Ant task useful for development and debugging where you need to restart an application. Requires that the WAR file (or directory) be on the same server on which Tomcat is running. Restarts of Tomcat cause the installation to be forgotten.

instance document
An XML document written against a specific schema.

ISO 3166
The international standard for country codes maintained by the International Organization for Standardization (ISO).

ISV
Independent Software Vendor.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

J2EE™
See *Java 2 Platform, Enterprise Edition*.

J2ME™
See *Java 2 Platform, Micro Edition*.

J2SE™
See *Java 2 Platform, Standard Edition*.

JAR
Java ARchive. A platform-independent file format that permits many files to be aggregated into one file.

Java™ 2 Platform, Enterprise Edition (J2EE)
An environment for developing and deploying enterprise applications. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing multitiered, Web-based applications.

Java 2 Platform, Micro Edition (J2ME)
A highly optimized Java runtime environment targeting a wide range of consumer products, including pagers, cellular phones, screenphones, digital set-top boxes and car navigation systems.

Java 2 Platform, Standard Edition (J2SE)
The core Java technology platform.

Java API for XML Binding (JAXB)
A Java technology that enables you to generate Java classes from XML schemas. As part of this process, JAXB also provides methods for unmar-

shalling XML instance documents into Java content trees, and then marshalling Java content trees back into XML instance documents. Put another way, JAXB provides a fast and convenient way to bind XML schemas to Java representations, making it easy for Java developers to incorporate XML data and processing functions in Java applications.

Java API for XML Messaging (JAXM)

An API that provides a standard way to send XML documents over the Internet from the Java platform. It is based on the SOAP 1.1 and SOAP with Attachments specifications, which define a basic framework for exchanging XML messages. JAXM can be extended to work with higher level messaging protocols, such as the one defined in the ebXML (electronic business XML) Message Service Specification, by adding the protocol's functionality on top of SOAP.

Java API for XML Processing (JAXP)

An API for processing XML documents. JAXP leverages the parser standards SAX and DOM so that you can choose to parse your data as a stream of events or to build a tree-structured representation of it. The latest versions of JAXP also support the XSLT (XML Stylesheet Language Transformations) standard, giving you control over the presentation of the data and enabling you to convert the data to other XML documents or to other formats, such as HTML. JAXP also provides namespace support, allowing you to work with schemas that might otherwise have naming conflicts.

Java API for XML Registries (JAXR)

An API for accessing different kinds of XML registries.

Java API for XML-based RPC (JAX-RPC)

An API for building Web services and clients that use remote procedure calls (RPC) and XML.

Java Naming and Directory Interface™ (JNDI)

An API that provides naming and directory functionality.

Java™ Secure Socket Extension (JSSE)

A set of packages that enable secure Internet communications.

Java™ Transaction API (JTA)

An API that allows applications to access transactions.

Java™ Web Services Developer Pack (Java WSDP)

An environment containing key technologies to simplify building of Web services using the Java 2 Platform.

JavaBeans™ component

A Java class that can be manipulated in a visual builder tool and composed into applications. A JavaBeans component must adhere to certain property and event interface conventions.

JavaMail™

An API for sending and receiving email.

JavaServer Pages™ (JSP™)

An extensible Web technology that uses template data, custom elements, scripting languages, and server-side Java objects to return dynamic content to a client. Typically the template data is HTML or XML elements, and in many cases the client is a Web browser.

JavaServer Pages Standard Tag Library (JSTL)

A tag library that encapsulates core functionality common to many JSP applications. JSTL has support for common, structural tasks such as iteration and conditionals, tags for manipulating XML documents, internationalization and locale-specific formatting tags, and SQL tags. It also introduces a new expression language to simplify page development, and provides an API for developers to simplify the configuration of JSTL tags and the development of custom tags that conform to JSTL conventions.

JAXR client

A client program that uses the JAXR API to access a business registry via a JAXR provider.

JAXR provider

An implementation of the JAXR API that provides access to a specific registry provider or to a class of registry providers that are based on a common specification.

JDBC™

An API for database-independent connectivity to a wide range of data sources.

JNDI

See *Java Naming and Directory Interface*.

JSP

See *JavaServer Pages*.

JSP action

A JSP element that can act on implicit objects and other server-side objects or can define new scripting variables. Actions follow the XML syntax for elements with a start tag, a body and an end tag; if the body is empty it can also use the empty tag syntax. The tag must use a prefix.

JSP action, custom
An action described in a portable manner by a tag library descriptor and a collection of Java classes and imported into a JSP page by a `taglib` directive. A custom action is invoked when a JSP page uses a custom tag.

JSP action, standard
An action that is defined in the JSP specification and is always available to a JSP file without being imported.

JSP application
A stand-alone Web application, written using the JavaServer Pages technology, that can contain JSP pages, servlets, HTML files, images, applets, and JavaBeans components.

JSP container
A *container* that provides the same services as a *servlet container* and an engine that interprets and processes JSP pages into a servlet.

JSP container, distributed
A JSP container that can run a Web application that is tagged as distributable and is spread across multiple Java virtual machines that might be running on different hosts.

JSP declaration
A JSP scripting element that declares methods, variables, or both in a JSP file.

JSP directive
A JSP element that gives an instruction to the JSP container and is interpreted at translation time.

JSP element
A portion of a JSP page that is recognized by a JSP translator. An element can be a *directive*, an *action*, or a *scripting element*.

JSP expression
A scripting element that contains a valid scripting language expression that is evaluated, converted to a `String`, and placed into the implicit `out` object.

JSP file
A file that contains a JSP page. In the Servlet 2.2 specification, a JSP file must have a .jsp extension.

JSP page
A text-based document using fixed template data and JSP elements that describes how to process a request to create a response.

JSP scripting element

A JSP *declaration*, *scriptlet*, or *expression*, whose tag syntax is defined by the JSP specification, and whose content is written according to the scripting language used in the JSP page. The JSP specification describes the syntax and semantics for the case where the language page attribute is "java".

JSP scriptlet

A JSP scripting element containing any code fragment that is valid in the scripting language used in the JSP page. The JSP specification describes what is a valid scriptlet for the case where the language page attribute is "java".

JSP tag

A piece of text between a left angle bracket and a right angle bracket that is used in a JSP file as part of a JSP element. The tag is distinguishable as markup, as opposed to data, because it is surrounded by angle brackets.

JSP tag library

A collection of custom tags identifying custom actions described via a tag library descriptor and Java classes.

JTA

See *Java Transaction API*.

life cycle

The framework events of a component's existence. Each type of component has defining events which mark its transition into states where it has varying availability for use. For example, a servlet is created and has its `init` method called by its container prior to invocation of its service method by clients or other servlets who require its functionality. After the call of its `init` method it has the data and readiness for its intended use. The servlet's `destroy` method is called by its container prior to the ending of its existence so that processing associated with winding up may be done, and resources may be released. The `init` and `destroy` methods in this example are *callback methods*.

localhost

For the purposes of the Java WSDP, the machine on which Tomcat is running.

local subset
That part of the DTD that is defined within the current XML file.

Logger element
A representation of a destination for logging, debugging and error messages for Tomcat.

A B C D E F G H I J K L **M** *N O P Q R S T U V W X Y Z*

marshal
The process of traversing a content tree and writing an XML document that reflects the tree's content. JAXB can marshal XML data to XML documents, SAX content handlers, and DOM nodes. See also: *unmarshal* and *validation*.

message-driven bean
An enterprise bean that is an asynchronous message consumer. A message-driven bean has no state for a specific client, but its instance variables may contain state across the handling of client messages. A client accesses a message-driven bean by sending messages to the destination for which the bean is a message listener.

mixed-content model
A DTD specification that defines an element as containing a mixture of text and one more other elements. The specification must start with `#PCDATA`, followed by alternate elements, and must end with the "zero-or-more" asterisk symbol (*).

mutual authentication
An authentication mechanism employed by two parties for the purpose of proving each other’s identity to one another.

A B C D E F G H I J K L M **N** *O P Q R S T U V W X Y Z*

namespace
A standard that lets you specify a unique label to the set of element names defined by a DTD. A document using that DTD can be included in any other document without having a conflict between element names. The elements defined in your DTD are then uniquely identified so that, for example, the parser can tell when an element called <name> should be interpreted according to your DTD, rather than using the definition for an element called `name` in a different DTD.

naming context
A set of associations between unique, atomic, people-friendly identifiers and objects.

naming environment

A mechanism that allows a component to be customized without the need to access or change the component's source code. A container implements the component's naming environment, and provides it to the component as a *JNDI naming context*. Each component names and accesses its environment entries using the `java:comp/env` JNDI context. The environment entries are declaratively specified in the component's deployment descriptor.

normalization

The process of removing redundancy by modularizing, as with subroutines, and of removing superfluous differences by reducing them to a common denominator. For example, line endings from different systems are normalized by reducing them to a single NL, and multiple whitespace characters are normalized to one space.

North American Industry Classification System (NAICS)

A system for classifying business establishments based on the processes they use to produce goods or services.

notation

A mechanism for defining a data format for a non-XML document referenced as an unparsed entity. This is a holdover from SGML that creaks a bit. The newer standard is to use MIME datatypes and namespaces to prevent naming conflicts.

A B C D E F G H I J K L M N **O** *P Q R S T U V W X Y Z*

OASIS

Organization for the Advancement of Structured Information Standards. Their home site is http://www.oasis-open.org/. The DTD repository they sponsor is at http://www.XML.org.

one-way messaging

A method of transmitting messages without having to block until a response is received.

OS principal

A principal native to the operating system on which the Web services platform is executing.

A B C D E F G H I J K L M N O **P** *Q R S T U V W X Y Z*

parameter entity

An entity that consists of DTD specifications, as distinct from a general entity. A parameter entity defined in the DTD can then be referenced at other points, in order to prevent having to recode the definition at each location it is used.

parsed entity

A general entity that contains XML, and which is therefore parsed when inserted into the XML document, as opposed to an unparsed entity.

parser

A module that reads in XML data from an input source and breaks it up into chunks so that your program knows when it is working with a tag, an attribute, or element data. A nonvalidating parser ensures that the XML data is well formed, but does not verify that it is valid. See also: *validating parser.*

principal

The identity assigned to a user as a result of authentication.

privilege

A security attribute that does not have the property of uniqueness and that may be shared by many principals.

processing instruction

Information contained in an XML structure that is intended to be interpreted by a specific application.

programmatic security

Security decisions that are made by security-aware applications. Programmatic security is useful when declarative security alone is not sufficient to express the security model of a application.

prolog

The part of an XML document that precedes the XML data. The prolog includes the declaration and an optional DTD.

public key certificate

Used in client-certificate authentication to enable the server, and optionally the client, to authenticate each other. The public key certificate is a digital equivalent of a passport. It is issued by a trusted organization, called a certificate authority (CA), and provides identification for the bearer.

A B C D E F G H I J K L M N O P Q **R** *S T U V W X Y Z*

RDF

Resource Description Framework. A standard for defining the kind of data that an XML file contains. Such information could help ensure semantic integrity, for example by helping to make sure that a date is treated as a date, rather than simply as text.

RDF schema

A standard for specifying consistency rules that apply to the specifications contained in an RDF.

reference

See entity reference

realm

See *security policy domain*. Also, a string, passed as part of an HTTP request during *basic authentication*, that defines a protection space. The protected resources on a server can be partitioned into a set of protection spaces, each with its own authentication scheme and/or authorization database.

In the Tomcat server authentication service, a realm is a complete database of roles, users, and groups that identify valid users of a Web application or a set of Web applications.

Realm element

A representation of a database of user names, passwords and roles assigned to those users.

registry

An infrastructure that enables the building, deployment and discovery of Web services. It is a neutral third party that facilitates dynamic and loosely coupled business-to-business (B2B) interactions.

registry provider

An implementation of a business registry that conforms to a specification for XML registries.

reload task

Used with the Tomcat `manager` Web application to redeploy a changed Web application onto a running Tomcat server.

request-response messaging

A method of messaging that includes blocking until a response is received.

resource manager

Provides access to a set of shared resources. A resource manager participates in transactions that are externally controlled and coordinated by a transaction manager. A resource manager is typically in different address space or on a different machine from the clients that access it. Note: An *enterprise information system* is referred to as resource manager when it is mentioned in the context of resource and transaction management.

resource manager connection

An object that represents a session with a resource manager.

resource manager connection factory

An object used for creating a resource manager connection.

role (security)

An abstract logical grouping of users that is defined by the Application Assembler. When an application is deployed, the roles are mapped to security identities, such as *principals* or *groups*, in the operational environment.

In the Tomcat server authentication service, a role is an abstract name for permission to access a particular set of resources. A role can be compared to a key that can open a lock. Many people might have a copy of the key, and the lock doesn't care who you are, only that you have the right key.

role mapping

The process of associating the groups and/or principals recognized by the container to security roles specified in the *deployment descriptor*. Security roles have to be mapped before a component is installed in the server.

rollback

The point in a transaction when all updates to any resources involved in the transaction are reversed.

root

The outermost element in an XML document. The element that contains all other elements.

SAX

Simple API for *XML*. An event-driven interface in which the parser invokes one of several methods supplied by the caller when a parsing event occurs. Events include recognizing an XML tag, finding an error, encountering a reference to an external entity, or processing a DTD specification.

schema

A database-inspired method for specifying constraints on XML documents using an XML-based language. Schemas address deficiencies in DTDs, such as the inability to put constraints on the kinds of data that can occur in a particular field. Since schemas are founded on XML, they are hierarchical, so it is easier to create an unambiguous specification, and possible to determine the scope over which a comment is meant to apply.

Secure Socket Layer (SSL)

A technology that allows Web browsers and Web servers to communicate over a secured connection.

security attributes

A set of properties associated with a principal. Security attributes can be associated with a principal by an authentication protocol or by a Java WSDP Product Provider.

security constraint

Determines who is authorized to access a Web resource collection.

security context

An object that encapsulates the shared state information regarding security between two entities.

security permission

A mechanism, defined by J2SE, to express the programming restrictions imposed on component developers.

security policy domain

A scope over which security policies are defined and enforced by a security administrator. A security policy domain has a collection of users (or principals), uses a well defined authentication protocol(s) for authenticating users (or principals), and may have groups to simplify setting of security policies.

security role

See *role (security)*.

security technology domain

A scope over which the same security mechanism is used to enforce a security policy. Multiple security policy domains can exist within a single technology domain.

server certificate

Used with HTTPS protocol to authenticate Web applications.The certificate can be self-signed or approved by a Certificate Authority (CA). The HTTPS service of the Tomcat server will not run unless a server certificate has been installed.

server principal
The OS principal that the server is executing as.

service element
A representation of the combination of one or more Connector components that share a single engine component for processing incoming requests.

servlet
A Java program that extends the functionality of a Web server, generating dynamic content and interacting with Web applications using a request-response paradigm.

servlet container
A *container* that provides the network services over which requests and responses are sent, decodes requests, and formats responses. All servlet containers must support HTTP as a protocol for requests and responses, but may also support additional request-response protocols such as HTTPS.

servlet container, distributed
A servlet container that can run a Web application that is tagged as distributable and that executes across multiple Java virtual machines running on the same host or on different hosts.

servlet context
An object that contains a servlet's view of the Web application within which the servlet is running. Using the context, a servlet can log events, obtain URL references to resources, and set and store attributes that other servlets in the context can use.

servlet mapping
Defines an association between a URL pattern and a servlet. The mapping is used to map requests to servlets.

session
An object used by a servlet to track a user's interaction with a Web application across multiple HTTP requests.

session bean
An enterprise bean that is created by a client and that usually exists only for the duration of a single client-server session. A session bean performs operations, such as calculations or accessing a database, for the client. Although a session bean may be transactional, it is not recoverable should a system crash occur. Session bean objects can be either stateless or can maintain conversational state across methods and transactions. If a session bean maintains state, then the EJB container manages this state if the object must be removed from memory. However, the session bean object itself must manage its own persistent data.

SGML

Standard Generalized Markup Language. The parent of both HTML and XML. However, while HTML shares SGML's propensity for embedding presentation information in the markup, XML is a standard that allows information content to be totally separated from the mechanisms for rendering that content.

SOAP

Simple Object Access Protocol

SOAP with Attachments API for Java (SAAJ)

The basic package for SOAP messaging which contains the API for creating and populating a SOAP message.

SSL

Secure Socket Layer. A security protocol that provides privacy over the Internet. The protocol allows client-server applications to communicate in a way that cannot be eavesdropped or tampered with. Servers are always authenticated and clients are optionally authenticated.

SQL

Structured Query Language. The standardized relational database language for defining database objects and manipulating data.

SQL/J

A set of standards that includes specifications for embedding SQL statements in methods in the Java programming language and specifications for calling Java static methods as SQL stored procedures and user-defined functions. An SQL checker can detects errors in static SQL statements at program development time, rather than at execution time as with a JDBC driver.

standalone client

A client that does not use a messaging provider and does not run in a container.

A B C D E F G H I J K L M N O P Q R S **T** ***U V W X Y Z***

tag

A piece of text that describes a unit of data, or element, in XML. The tag is distinguishable as markup, as opposed to data, because it is surrounded by angle brackets (< and >). To treat such markup syntax as data, you use an entity reference or a CDATA section.

template

A set of formatting instructions that apply to the nodes selected by an XPATH expression.

Tomcat
The Java Servlet and JSP Web server and container developed by the Apache Software Foundation and included with the Java WSDP. Many applications in this tutorial are run on Tomcat.

transaction
An atomic unit of work that modifies data. A transaction encloses one or more program statements, all of which either complete or roll back. Transactions enable multiple users to access the same data concurrently.

transaction isolation level
The degree to which the intermediate state of the data being modified by a transaction is visible to other concurrent transactions and data being modified by other transactions is visible to it.

transaction manager
Provides the services and management functions required to support transaction demarcation, transactional resource management, synchronization, and transaction context propagation.

translet
Pre-compiled version of a tranformation.

A B C D E F G H I J K L M N O P Q R S T U *V W X Y Z*

Unicode
A standard defined by the Unicode Consortium that uses a 16-bit code page which maps digits to characters in languages around the world. Because 16 bits covers 32,768 codes, Unicode is large enough to include all the world's languages, with the exception of ideographic languages that have a different character for every concept, like Chinese. For more info, see http://www.unicode.org/.

Universal Description, Discovery, and Integration (UDDI) project
An industry initiative to create a platform-independent, open framework for describing services, discovering businesses, and integrating business services using the Internet, as well as a registry. It is being developed by a vendor consortium.

Universal Standard Products and Services Classification (UNSPSC)
A schema that classifies and identifies commodities. It is used in sell side and buy side catalogs and as a standardized account code in analyzing expenditure.

unmarshal
The process of reading an XML document and constructing a tree of content objects. Each content object corresponds directly to an instance in the input document of the corresponding schema component, and the content tree rep-

resents the document's content and structure as a whole. See also: *marshal* and *validation*.

unparsed entity

A general entity that contains something other than XML. By its nature, an unparsed entity contains binary data.

URI

Uniform Resource Identifier. A globally unique identifier for an abstract or physical resource. A *URL* is a kind of URI that specifies the retrieval protocol (http or https for Web applications) and physical location of a resource (host name and host-relative path). A *URN* is another type of URI.

URL

Uniform Resource Locator. A standard for writing a textual reference to an arbitrary piece of data in the World Wide Web. A URL looks like `protocol://host/localinfo` where `protocol` specifies a protocol for fetching the object (such as HTTP or FTP), `host` specifies the Internet name of the targeted host, and `localinfo` is a string (often a file name) passed to the protocol handler on the remote host.

URL path

The part of a URL passed by an HTTP request to invoke a servlet. A URL path consists of the Context Path + Servlet Path + Path Info, where

- Context Path is the path prefix associated with a servlet context that this servlet is a part of. If this context is the default context rooted at the base of the Web server’s URL namespace, the path prefix will be an empty string. Otherwise, the path prefix starts with a / character but does not end with a / character.
- Servlet Path is the path section that directly corresponds to the mapping which activated this request. This path starts with a / character.
- Path Info is the part of the request path that is not part of the Context Path or the Servlet Path.

URN

Uniform Resource Name. A unique identifier that identifies an entity, but doesn’t tell where it is located. A system can use a URN to look up an entity locally before trying to find it on the Web. It also allows the Web location to change, while still allowing the entity to be found.

user (security)

An individual (or application program) identity that has been authenticated. A user can have a set of roles associated with that identity, which entitles them to access all resources protected by those roles.

user data constraint

Indicates how data between a client and a Web container should be protected. The protection can be the prevention of tampering with the data or prevention of eavesdropping on the data.

A B C D E F G H I J K L M N O P Q R S T U **V** *W X Y Z*

valid

A valid XML document, in addition to being well formed, conforms to all the constraints imposed by a DTD. It does not contain any tags that are not permitted by the DTD, and the order of the tags conforms to the DTD's specifications.

validation

The process of verifying that the constraints expressed in a source schema are satisfied in a given content tree. In JAXB, a content tree is valid only if marshalling the tree would generate a document that is valid with respect to the source schema. An XML document is said to be valid if it satisfies the constraints defined in the DTD and or schema(s) against which the document is written.

validating parser

A parser that ensures that an XML document is valid, as well as well-formed. See also: *parser.*

Valve element

A representation of a component that will be inserted into the request processing pipeline for Tomcat.

virtual host

Multiple “hosts + domain names” mapped to a single IP address.

A B C D E F G H I J K L M N O P Q R S T U V **W** *X Y Z*

W3C

World Wide Web Consortium. The international body that governs Internet standards.

WAR file

Web application archive. A JAR archive that contains a Web module.

warning

A SAX parser warning is generated when the document's DTD contains duplicate definitions, and similar situations that are not necessarily an error, but which the document author might like to know about, since they could be. See also: *fatal error*, *error*.

Web application
An application written for the Internet, including those built with Java technologies such as JavaServer Pages and servlets, as well as those built with non-Java technologies such as CGI and Perl.

Web Application Archive (WAR)
A hierarchy of directories and files in a standard Web application format, contained in a packed file with an extension .war.

Web application, distributable
A Web application that uses Java WSDP technology written so that it can be deployed in a Web container distributed across multiple Java virtual machines running on the same host or different hosts. The deployment descriptor for such an application uses the distributable element.

Web component
A component that provides services in response to requests; either a *servlet* or a *JSP page*.

Web container
A *container* that implements the Web component contract of the J2EE architecture. This contract specifies a runtime environment for Web components that includes security, concurrency, life cycle management, transaction, deployment, and other services. A Web container provides the same services as a *JSP container* and a federated view of the J2EE platform APIs. A Web container is provided by a *Web server*.

Web container, distributed
A Web container that can run a Web application that is tagged as distributable and that executes across multiple Java virtual machines running on the same host or on different hosts.

Web module
A unit that consists of one or more Web components, other resources, and a Web deployment descriptor.

Web resource
A static or dynamic object contained in a Web application archive that can be referenced by a URL.

Web resource collection
A list of URL patterns and HTTP methods that describe a set of resources to be protected.

Web server

Software that provides services to access the Internet, an intranet, or an extranet. A Web server hosts Web sites, provides support for HTTP and other protocols, and executes server-side programs (such as CGI scripts or servlets) that perform certain functions. In the J2EE architecture, a Web server provides services to a *Web container*. For example, a Web container typically relies on a Web server to provide HTTP message handling. The J2EE architecture assumes that a Web container is hosted by a Web server from the same vendor, so does not specify the contract between these two entities. A Web server may host one or more Web containers.

Web service

An application that exists in a distributed environment, such as the Internet. A Web service accepts a request, performs its function based on the request, and returns a response. The request and the response can be part of the same operation, or they can occur separately, in which case the consumer does not need to wait for a response. Both the request and the response usually take the form of XML, a portable data-interchange format, and are delivered over a wire protocol, such as HTTP.

well-formed

An XML document that is syntactically correct. It does not have any angle brackets that are not part of tags, all tags have an ending tag or are themselves self-ending, and all tags are fully nested. Knowing that a document is well formed makes it possible to process it. A well-formed document may not be valid however. To determine that, you need a *validating parser* and a *DTD*.

A B C D E F G H I J K L M N O P Q R S T U V W* X *Y Z

Xalan

An interpreting version of XSLT.

XHTML

An XML lookalike for *HTML* defined by one of several XHTML DTDs. To use XHTML for everything would of course defeat the purpose of XML, since the idea of XML is to identify information content, not just tell how to display it. You can reference it in a DTD, which allows you to say, for example, that the text in an element can contain <em> and <b> tags, rather than being limited to plain text.

XLink

The part of the XLL specification that is concerned with specifying links between documents.

XLL

The XML Link Language specification, consisting of *XLink* and *XPointer.*

XML

Extensible Markup Language. A markup language that allows you to define the tags (markup) needed to identify the content, data, and text, in XML documents. It differs from *HTML* the markup language most often used to present information on the internet. HTML has fixed tags that deal mainly with style or presentation. An XML document must undergo a transformation into a language with style tags under the control of a stylesheet before it can be presented by a browser or other presentation mechanism. Two types of style sheets used with XML are *CSS* and *XSL*. Typically, XML is transformed into HTML for presentation. Although tags may be defined as needed in the generation of an XML document, a *DTD* may be used to define the elements allowed in a particular type of document. A document may be compared with the rules in the DTD to determine its validity and to locate particular elements in the document. Web services application's deployment descriptors are expressed in XML with DTDs defining allowed elements. Programs for processing XML documents use *SAX* or *DOM* APIs.

XML registry

See *registry.*

XML Schema

The W3C schema specification for XML documents.

XPath

See XSL.

XPointer

The part of the XLL specification that is concerned with identifying sections of documents so that they can referenced in links or included in other documents.

XSL

Extensible Stylesheet Language. Extensible Stylesheet Language. An important standard that achieves several goals. XSL lets you:

a.Specify an addressing mechanism, so you can identify the parts of an XML file that a transformation applies to. (XPath)

b.Specify tag conversions, so you convert XML data into a different formats. (XSLT)

c.Specify display characteristics, such page sizes, margins, and font heights and widths, as well as the flow objects on each page. Information fills in one area of a page and then automatically flows to the next object when that area fills up. That allows you to wrap text around pictures, for example, or to continue a newsletter article on a different page. (XML-FO)

XSL-FO

A subcomponent of XSL used for describing font sizes, page layouts, and how information “flows” from one page to another.

XSLT

XSL Transformation. An XML file that controls the transformation of an XML document into another XML document or HTML. The target document often will have presentation related tags dictating how it will be rendered by a browser or other presentation mechanism. XSLT was formerly part of XSL, which also included a tag language of style flow objects.

XSLTC

A compiling version of XSLT.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

About the Authors

Java API for XML Processing

Eric Armstrong has been programming and writing professionally since before there were personal computers. His production experience includes artificial intelligence (AI) programs, system libraries, real-time programs, and business applications in a variety of languages. He works as a consultant at Sun's Java Software division in the Bay Area, and he is a contributor to JavaWorld. He wrote The *JBuilder2 Bible*, as well as Sun's Java XML programming tutorial. For a time, Eric was involved in efforts to design next-generation collaborative discussion/decision systems. His learn-by-ear, see-the-fingering music teaching program is currently on hold while he finishes a weight training book. His Web site is `http://www.treelight.com`.

Web Applications and Technology

Stephanie Bodoff is a staff writer at Sun Microsystems. In previous positions she worked as a software engineer on distributed computing and telecommunications systems and object-oriented software development methods. Since her conversion to technical writing, Stephanie has documented object-oriented databases, application servers, and enterprise application development methods. She is a co-author of *The J2EE Tutorial*, *Designing Enterprise Applications with the Java™ 2 Platform, Enterprise Edition*, and *Object-Oriented Software Development: The Fusion Method*.

Getting Started, Web Application Security

Debbie Carson is a staff writer with Sun Microsystems, where she documents both the J2EE and J2SE platforms. In previous positions she documented creating database applications using C++ and Java technologies and creating distributed applications using Java technology. In addition to this chapter, she currently writes about the CORBA technologies Java IDL and Java Remote Method Invocation over Internet InterORB Protocol (RMI-IIOP), Web services security, and Web services tools.

Java API for XML Messaging, Introduction to Web Services

Maydene Fisher has documented various Java APIs at Sun Microsystems for the last five years. She authored two books on the JDBC API, *JDBC™ Database Access with Java: A Tutorial and Annotated Reference* and *JDBC™ API Tutorial and Reference, Second Edition: Universal Data Access for the Java™ 2 Platform*. Before joining Sun, she helped document the object-oriented programming language ScriptX at Kaleida Labs and worked on Wall Street, where she wrote developer and user manuals for complex financial computer models written in C++. In previous lives, she has been an English teacher, a shopkeeper in Mendocino, and a financial planner.

Java API for XML Binding

Scott Fordin is a senior staff writer, illustrator, and online help specialist in the Java and XML Technology groups at Sun Microsystems. He has written numerous articles on Java, XML, and Web service technologies, and is the maintainer of Sun's XML Web site. In addition to his Web work, Scott has written many developer guides, administrator guides, user guides, specifications, whitepapers, and tutorials for a wide range of products. Some of his recent work includes writing and illustrating the JAXB User's Guide, editing the JAXB specification, co-editing and illustrating the Web Services Choreography Interface W3C Technical Note, writing and illustrating the Solaris Management Console Programming Guide, and co-developing the embedded online help model for the Solaris Management Console.

Java API for RPC-based XML

Dale Green is a staff writer with Sun Microsystems, where he documents the J2EE platform and the Java API for RPC-based XML. In previous positions he programmed business applications, designed databases, taught technical classes, and documented RDBMS products. He wrote the Internationalization and Reflection trails for the *Java Tutorial Continued*, and co-authored *The J2EE Tutorial*.

Java API for XML Registries, Java WSDP Registry Server

Kim Haase is a staff writer with Sun Microsystems. In previous positions she has documented compilers, debuggers, and floating-point programming. She currently writes about the Java Message Service, the Java API for XML Registries, the SOAP with Attachments API for Java, and the Java API for XML Messaging. She is a co-author of *Java™ Message Service API Tutorial and Reference* and *The J2EE Tutorial*.

Introduction to Web Services, Web Application Security

Eric Jendrock is a staff writer with Sun Microsystems, where he documents the J2EE platform and the Java WSDP. Previously, he documented middleware products and standards. Currently, he writes about the J2EE Compatibility Test Suite and security in the Web-tier and in the J2EE platform.

Index