The Java™ Web Services Tutorial

Eric Armstrong
Jennifer Ball
Stephanie Bodoff
Debbie Bode Carson
Ian Evans
Maydene Fisher
Scott Fordin
Dale Green
Kim Haase
Eric Jendrock

December 17, 2003

Contents

About This Tutorial

THIS tutorial is a beginner's guide to developing enterprise 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. Here we cover all the things you need to know to make the best use of this tutorial.

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 six 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 the technologies for developing applications that process XML documents and provide Web services:

  - The Java API for XML Processing (JAXP)

  - The Java API for XML-based RPC (JAX-RPC)
  - SOAP with Attachments API for Java (SAAJ)
  - The Java API for XML Registries (JAXR)
  - The Java Architecture for XML Binding (JAXB)
- Web technology

  These chapters cover the component technologies used in developing the presentation layer of a Web application.

  - Java Servlets
  - JavaServer Pages
  - JavaServer Pages Standard Tag Library
  - JavaServer Faces
- Platform Services

  This chapter covers the services used by all the component technologies.

  - Security
- Case Study
  - The Coffee Break Application chapter describes an application that ties together the Web application and Web services APIs.
- Appendixes
  - Tomcat Server Administration Tool
  - Tomcat Web Application Manager
  - Registry Browser
  - Java encoding schemes
  - HTTP overview

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 Java™ Tutorial* listed in Table 1–1 are particularly relevant:

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

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

Building and Running the Examples

This section tells you everything you need to know to obtain, build, 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, the example source code is in the `<INSTALL>/jwstutorial13/examples/` directory, with subdirectories for each of the technologies included in the pack except for the JavaServer Faces and JAXB technologies. The examples for JavaServer Faces technology are included in the JavaServer Faces 1.0 Beta release in the `<JSF_HOME>/samples` directory. The examples for JAXB are included in the Java WSDP in the `<JWSDP_HOME>/jaxb/samples` directories.

This tutorial documents the Java WSDP 1.3 as well as the JavaServer Faces 1.0 Beta release. 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) 1.4.2_01 or higher. To build, deploy, and run the JavaServer Faces examples, you also need the JavaServer Faces technology Beta release in addition to the Java WSDP and the J2SE SDK. You download the Java WSDP from:

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

and the J2SE 1.4 SDK from

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

You download the JavaServer Faces technology release from:

```
http://java.sun.com/j2ee/javaserverfaces/download.html
```

Building the Examples

Most of the examples are distributed with a build file for Ant, a portable build tool contained in the Java WSDP. For information about Ant, visit `http://ant.apache.org/`. Directions for building the examples are provided in each chapter. In order to run the Ant scripts, you must configure your environment and properties files as follows:

- Add the `bin` directory of your J2SE SDK installation to the front of your path.
- Add `<`*`JWSDP_HOME`*`>/bin` to the front of your path so that Java WSDP 1.3 scripts override other installations.
- Add `<`*`JWSDP_HOME`*`>/jwsdp-shared/bin` to the front of your path so the Java WSDP 1.3 scripts that are shared by multiple components override other installations.
- Add `<`*`JWSDP_HOME`*`>/apache-ant/bin` to the front of your path so that the Java WSDP 1.3 Ant script overrides other installations. The version of Ant shipped with the Java WSDP sets the `jwsdp.home` property, which is required by the example build files. If you do not use this version of Ant, you will need to set the `jwsdp.home` property in the `build.properties` file described in the next bullet.
- Set the following properties in the file `<`*`INSTALL`*`>/jwstutorial13/examples/common/build.properties`:

- Set `tutorial.home` to the location of your Java Web Services Tutorial installation.
- Set the `username` and `password` properties to the values you specified when you installed the Java WSDP. The build scripts use these values when you invoke an administration task such as installing an application.
- Set the `host` and `port` properties to the appropriate values if you are running Tomcat on a host other than localhost or on a port other than 8080 (the default values).

Tutorial Example Directory Structure

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

- `build.xml`—Ant build file
- `src`—Java source of servlets and JavaBeans components, and tag libraries
- `web`—JSP pages and HTML pages, tag files, images

The Ant build files (`build.xml`) distributed with the examples contain targets to create a `build` subdirectory and copy and compile files into that directory and perform administrative functions on the application server. Build properties and targets common to a particular technology are specified in the files `<`*`INSTALL`*`>/jwstutorial13/examples/`*`technology`*`/common/build.properties` and `<`*`INSTALL`*`>/jwstutorial13/examples/`*`technology`*`/common/targets.xml`.

Managing the Examples

Many of the Java WSDP examples run on the Tomcat Java servlet and JSP container. You use the `manager` tool to list, install, reload, remove, deploy, and undeploy 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 1–2 lists the typographical conventions used in this tutorial.

Table 1–2 Typographical Conventions

Font Style	**Uses**
italic	Emphasis, titles, first occurrence of terms
`monospace`	URLs, code examples, file names, path names, command names, programming language keywords, properties
`italic monospace`	Variable names
<*italic monospace*>	Variable path names, environment variables in paths

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

WEB services, in the general meaning of the term, are services offered by one application to other applications via the World Wide Web. Clients of these services can aggregate them to form an end-user application, enable business transactions, or create new Web services.

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), 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, which implements version 1.3.1 of the J2EE platform, are available in the Java WSDP documentation at `<`*`JWSDP_HOME`*`>/docs/jwsdpons1as7.html`.

Most of the APIs in the Java WSDP are part of the J2EE platform, version 1.4. For more information, go to `http://java.sun.com/j2ee/`.

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 5 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 added 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
  - SOAP with Attachments API for Java (SAAJ) — sends SOAP messages over the Internet in a standard way
- 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 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. 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 157) (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 1.2.4, which includes support for XML Schema, is in the Java WSDP.

The SAX API

The Simple API for XML (page 169) (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 227) (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 231).

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 301) (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 schema 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 10 for a description of the JAXB architecture, functions, and core concepts and then see Chapter 11, 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.

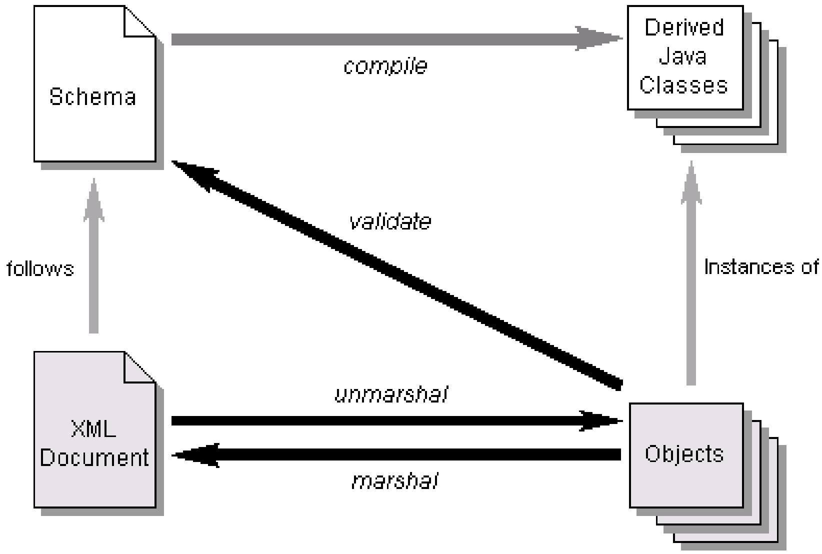


Figure 1–1 Data Binding Process

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="USAddress"/> <xsd:element name="billTo" type="USAddress"/> <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 12 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 SAAJ 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 those that find communication using the RPC model a good fit. The lower-level alternative for SOAP messaging, the SOAP with Attachments API for Java (SAAJ), 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.

SAAJ

The SOAP with Attachments API for Java (SAAJ) 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.

See Chapter 13 to see how to use the SAAJ API and run the SAAJ examples that are included with this tutorial.

A SAAJ client is a *standalone* client. That is, it sends 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.

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. Common types of attachment include sound, picture, and movie data: .mp3, .jpg, and .mpg files.

Getting a Connection

The first thing a SAAJ client needs to do is get a connection in the form of a SOAPConnection object. A SOAPConnection object is a point-to-point connection that goes directly from the sender to the recipient. The connection is created by 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 connection = factory.createConnection();
```

Creating a Message

Messages, like connections, are created by a factory. To obtain a MessageFactory object, you get an instance of the default implementation for the MessageFactory class. This instance can then be used to create a SOAPMessage object.

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

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

The new SOAPMessage object message 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 soapPart = message.getSOAPPart();
SOAPEnvelope envelope = soapPart.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 dbFactory=
    DocumentBuilderFactory.newInstance();
DocumentBuilder builder = dbFactory.newDocumentBuilder();
Document document = builder.parse("file:///foo.bar/soap.xml");
DOMSource domSource = new DOMSource(document);

soapPart.setContent(domSource);
```

This code would work equally well with a `SAXSource` or a `StreamSource` object.

You use the `setContent` method when you want to send an existing SOAP message. If you have an XML document that you want to send as the content of a SOAP message, you use the `addDocument` method on the body of the message:

```
SOAPBodyElement docElement = body.addDocument(document);
```

This allows you to keep your application data in a document that is separate from the SOAP envelope unless and until it is time to send that data as a message.

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 `handler`. The `Message` object `message` 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 handler = new DataHandler(url);
AttachmentPart attachPart =
    message.createAttachmentPart(handler);
message.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 =
    message.createAttachmentPart("content-string",
        "text/plain");
message.addAttachmentPart(attachPart);
```

Sending a Message

Once you have populated a `SOAPMessage` object, you are ready to send it. A 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);
```

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 14 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 SAAJ. 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 SAAJ 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 SAAJ 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 SAAJ messages using agreed-upon XML schemas. Therefore, for this distributor, the engineer's code uses the SAAJ API to request current prices, and the distributor returns the price list in a SOAP 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 SAAJ. Each supplier, whether using JAX-

RPC or SAAJ, 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 25 demonstrates a simple implementation of this scenario.

2

Introduction to Interactive Web Application Technologies

IN the Java 2 platform, Web components are the foundation technology for providing dynamic, user-oriented Web content. The first type of Web components introduced were Java Servlets. Java Servlets provided a portable, efficient way to extend the functionality of Web servers. Soon after, JavaServer Pages (JSP) technology, which defined another type of Web component, was introduced. JavaServer Pages technology provides a more natural mechanism for mixing static and dynamic Web content.

The interaction between Web client and a Java-technology based Web application is illustrated in Figure 2–1.The client sends an HTTP request to the Web server. A Web server that implements Java Servlet and JavaServer Pages technology converts the request into an `HTTPServletRequest` object. This object is delivered to a Web component which may interact with JavaBeans components or a database to generate dynamic content. The Web component may then generate an `HTTPServletResponse` or it may pass the request to another Web component. Eventually, the response is generated, and Web server returns it to the client.

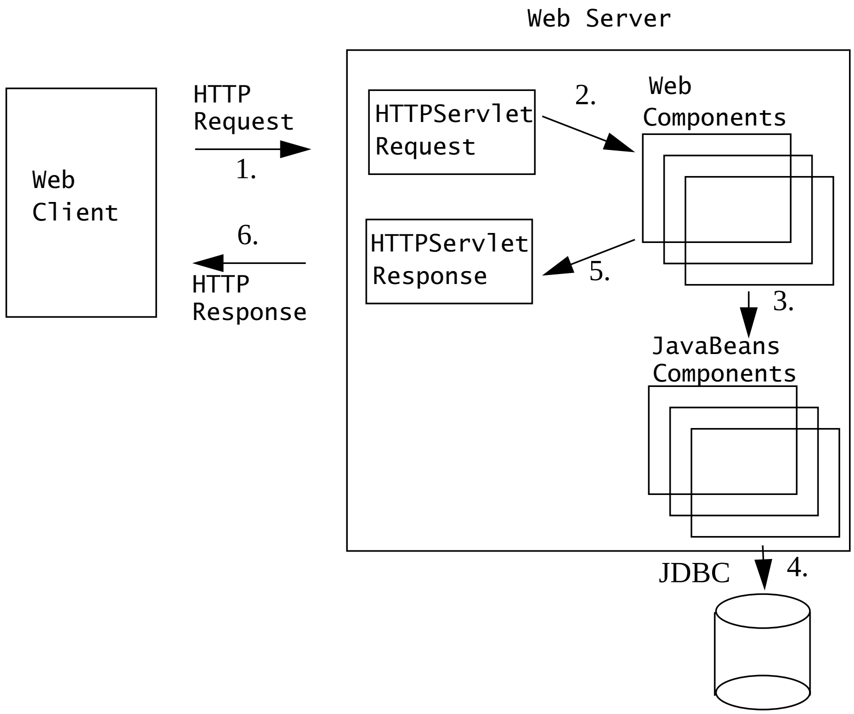


Figure 2–1 Java Web Application Request Handling

Since the introduction of Java Servlet and JSP technology, other Java technologies and frameworks for building interactive Web applications have been developed. These technologies and their relationships are illustrated in Figure 2–2.

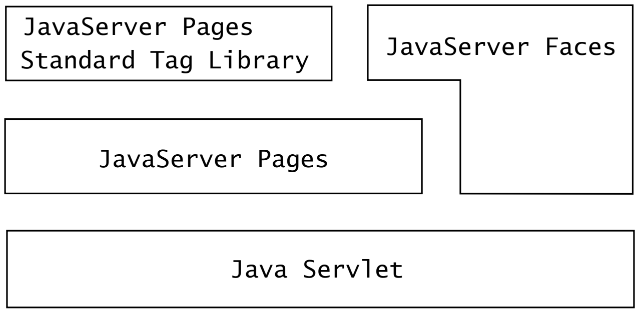


Figure 2–2 Java Web Application Technologies

Notice that Java Servlet technology is the foundation. Each technology adds a level of abstraction that makes Web application prototyping and development faster and the Web applications themselves more maintainable, scalable, and robust. This chapter starts off with an introduction to interactive Web application architectures. Then we provide an introduction to each of the technologies and summarize their roles in developing interactive Web applications. Later chapters in this tutorial describe how to use the technologies to develop interactive Web applications.

Interactive Web Application Architectures

The Model-View-Controller (MVC) architecture is a widely-used architectural approach for interactive applications. The MVC architecture separates functionality among application objects so as to minimize the degree of coupling between the objects. To achieve this, it divides applications into three layers: Model, View, and Controller. Each layer handles specific tasks and has responsibilities to the other layers:

- The Model represents business data and business logic or operations that govern access and modification of this business data. The model notifies views when it changes and provides the ability for the view to query the model about its state. It also provides the ability for the controller to access application functionality encapsulated by the model.

- The View renders the contents of a model. It gets data from the model and specifies how that data should be presented. It updates data presentation when the model changes. A view also forwards user input to a controller.
- The Controller defines application behavior. It dispatches user requests and selects views for presentation. It interprets user inputs and maps them into actions to be performed by the model. In a Web application, user inputs are HTTP GET and POST requests. A controller selects the next view to display based on the user interactions and the outcome of the model operations.

When employed in a Web application, the MVC architecture is often referred to as a Model-2 architecture. A Web application that intermixes presentation and business logic employs what is known as a Model-1 architecture. The Model-2 architecture is the recommended approach for designing Web applications.

Java Servlet Technology

As noted in the previous section, Java Servlet Technology (page 605) is the foundation technology for all interactive Web applications whose user interfaces is generated on the server.

Java Servlet technology consists of two parts:

- A Java programming language API that encapsulates requests and responses and their subobjects and a process for handling these objects.
- A declarative mechanism for specifying Web application properties outside the Web application code and which can be modified at deployment time. See Removing Web Applications (page 84) for an introduction to this aspect of Java Servlet technology.

A *servlet* is a Java programming language class that dynamically processes requests and constructs responses. 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 main methods in an HTTP servlet process the standard HTTP requests: GET and PUT. Here's an example of a servlet that allows the user to input their name into a form, and then calls another servlet to generate a greeting response:

```
public class GreetingServlet extends HttpServlet {
  public void doGet (HttpServletRequest request,
     HttpServletResponse response)
     throws ServletException, IOException
  {
     // Set response headers
     response.setContentType("text/html");
     response.setBufferSize(8192);
     PrintWriter out = response.getWriter();

     // then write the response
     out.println("<html>" +
        "<head><title>Hello</title></head>");
        out.println("<body  bgcolor=\"#ffffff\">" +
        "<img src=\"duke.waving.gif\">" +
        "<h2>Hello, my name is Duke. What's yours?</h2>" +
        "<form method=\"get\">" +
        "<input type=\"text\" name=\"username\" size=\"25\">" +
        "<p></p>" +
        "<input type=\"submit\" value=\"Submit\">" +
        "<input type=\"reset\" value=\"Reset\">" +
        "</form>");
     UserNameBean userNameBean = new UserNameBean();
     userNameBean.setName(request.getParameter("username"));
     if ( userNameBean.getName() != null
        && userNameBean.getName().length() > 0 ) {
        RequestDispatcher dispatcher =
           getServletContext().getRequestDispatcher(
           "/response");
        if (dispatcher != null)
           dispatcher.include(request, response);
     }
     out.println("</body></html>");
     out.close();
  }
}
```

As you can see from this example, the one limitation of servlets is in generating responses whose main content is static text, such as HTML markup. Since the response is wholly generated within a Java class, the text must be embedded within `println` statements that write to the response writer object. Not only is this difficult to maintain, but forces the content developer to be a Java programmer. Conversely, when the content is binary, for example an image, servlets are well suited to the task. Servlets are also well suited to performing control functions because the full capabilities of the Java programming language are available. In fact, servlets often serve as the Controller in Web applications that employ an MVC architecture.

JavaServer Pages Technology

JavaServer Pages Technology (page 641) makes all the dynamic capabilities of Java Servlet technology available to the Web application developer 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
- An expression language for accessing server-side objects
- Mechanisms for defining extensions (custom tags) to the JSP language

A *JSP page* is a 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 JSP page is translated into a servlet and compiled the first time a request is routed to it. For example, here is the JSP version of the application introduced in the previous section:

```
// greeting.jsp
<html>
<head><title>Hello</title></head>
<body bgcolor="white">
<img src="duke.waving.gif">
<h2>My name is Duke. What is yours?</h2>
<form method="get">
<input type="text" name="username" size="25">
<p></p>
<input type="submit" value="Submit">
<input type="reset" value="Reset">
</form>
<jsp:useBean id="userNameBean" class="hello.UserNameBean"
  scope="request"/>
<jsp:setProperty name="userNameBean" property="name"
  value="<%=request.getParameter('username')%>" />
<%
if ( userNameBean.getName() != null &&
```

```
      userNameBean.getName().length() > 0 ) {
%>
  <%@include file="response.jsp" %>
<%
}
%>
</body>
</html>

// response.jsp
<jsp:useBean id="userNameBean" class="hello.UserNameBean"
  scope="request"/>
<h2><font color="red">Hello, ${userNameBean.name}!</font></h2>
```

The `username` request parameter is used to set the `name` property of the JavaBeans component `UserNameBean`. Java scripting expressions are used to validate the property value and conditionally include the response if the property is valid.

Early versions of JSP technology placed an emphasis on generating dynamic content by using Java-based scripts (see Chapter 19). The latest version of JSP technology down plays this approach in favor of encapsulating such functions in custom tags (see Chapter 18). The next two sections describe two important standard tag libraries which minimize the need to use scripting in JSP pages.

In summary, the strengths of JSP technology are:

- Strong support for template data
- Powerful expression language for accessing Java objects to generate dynamic content
- Easy to extend

JSP pages typically play the role of the View in an MVC-based Web application and the Model objects are JavaBeans components. Usually, in MVC applications, the Controller creates most of the Model objects.

JavaServer Pages Standard Tag Library

The JavaServer Pages Standard Tag Library (page 681) (JSTL) encapsulates core functionality common to many JSP applications. Instead of iterating over lists using a scriptlet or different iteration tags from numerous vendors, JSTL defines a standard set of tags. This standardization allows you to learn a single set of tags and use them on multiple JSP containers. Also, a standard tag library is more likely to have an optimized implementation.

JSTL consists of several sub libraries that handle the follow functions:

- Core: flow control, URL management
- XML document manipulation: core, flow control, transformation
- Internationalization: message localization and number, currency, and date formatting
- SQL database access: query, update, transactions
- Functions (String, Array ...)

Here's the JSP page from the previous section rewritten to use the JSTL `fn:length` function and `c:if` tag to perform the checks done by the scriptlet:

```
// greeting.jsp
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
  prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions"
  prefix="fn" %>
<html>
<head><title>Hello</title></head>
<body bgcolor="white">
<img src="duke.waving.gif">
<h2>Hello, my name is Duke. What's yours?</h2>
<form method="get">
<input type="text" name="username" size="25">
<p></p>
<input type="submit" value="Submit">
<input type="reset" value="Reset">
</form>

<jsp:useBean id="userNameBean" class="hello.UserNameBean"
  scope="request"/>
<jsp:setProperty name="userNameBean" property="name"
  value="${param.username}" />

<c:if test="${fn:length(userNameBean.name) > 0}" >
  <%@include file="response.jsp" %>
</c:if>

</body>
</html>
```

JSTL contains many useful functions and it is hard to see limitations in this small example. Indeed, JSTL excels when prototyping small Web applications. Its limitations become more evident in Web applications containing several UI input components on a given page whose input must be validated or in Web applications with many pages. Thus, what's missing from JSTL are the larger granularity functions, such as user interface components and mechanisms for controlling the flow from one page to the next.

JavaServer Faces Technology

JavaServer Faces Technology (page 771) is a user interface framework for building Web applications. The main components of JavaServer Faces technology are:

- A graphical user interface (GUI) component framework
- A flexible model for rendering components in different kinds of HTML, or different markup languages and technologies. A `Renderer` generates the markup to render the component and converts the data stored in a model object to types that can be represented in a View.
- A standard `RenderKit` for generating HTML/4.01 markup.

In support of the GUI components are the following features:

- Managed model object creation
- Input validation
- Event handling
- Data conversion between model objects and components
- Page navigation configuration

All of this functionality is available via standard Java APIs and XML-based configuration files, and is thus available to applications that aren't based on JSP technology. For the JavaServer Faces applications that do employ JSP technology, the following support is included in JavaServer Faces technology:

- A standard JSP tag library for generic functions that are independent of the specific `RenderKit` in use (such as adding a validator to a component).
- A standard JSP tag library for the HTML `RenderKit`, with a tag for each combination of a component type and a method of rendering that component type. Consider the `UISelectOne` component, which represents a list of options, and allows only a single option from the list to be selected. Such a component can be rendered in three different ways (in the basic HTML `RenderKit`), each with a different `Renderer` and a corresponding custom tag:
    - `h:selectone_listbox`—Display a list of all the possible options.

- `h:selectone_menu`—Display as a combo box (the traditional HTML `select` element with `size="1"`).
- `h:selectone_radio`—Display as a set of radio buttons and corresponding labels.

You can also create more complex components like grids, tree controls, and the like. One way to accomplish this is by nesting JavaServer Faces component tags inside each other, just like you nest HTML `input` elements inside a `form` element. You can also define complex components using the JavaServer Faces API.

The GUI components and well-defined programming model significantly ease the burden of building and maintaining Web applications with server-side GUIs. With minimal effort, you can:

- Construct a GUI with reusable and extensible components
- Map GUI components on a page to server-side data
- Wire client-generated events to server-side application code
- Save and restore GUI state beyond the life of server requests

The following code examples contain the JSP page and JavaServer Faces configuration file for the JavaServer Faces version of the example discussed in previous sections:

```
//greeting.jsp

<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<html>
<head><title>Hello</title></head>
<body bgcolor="white">
<f:view>
  <h2>My name is Duke.  What is yours?</h2>
  <h:graphic_image id="wave_img" url="/duke.waving.gif" />
  <h:form id="helloForm" >
    <h:input_text id="username"
      columns="25" value="#{userNameBean.name}"
      required="true"/>
    <p></p>
    <h:command_button id="submit" label="Submit"
      action="success" />
    <h:command_button id="reset" label="Reset"
      action="success" />
```

```
  </h:form>
</f:view>
</body>
</html>
```

Notice how JavaServer Faces custom tags have replaced HTML elements. The user name set as a request parameter is handled by the `h:input` text tag in a model object `UserNameBean` which can be accessed by the response page. The `required` attribute of the `h:input_text` tag causes the JavaServer Faces implementation to check that the user name was entered. The Submit and Reset buttons are linked to commands with `h:command_button` tags.

```
// faces-config.xml
...
<navigation-rule>
  <from-tree-id>/greeting.jsp</from-tree-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-tree-id>/response.jsp</to-tree-id>
  </navigation-case>
</navigation-rule>
<managed-bean>
  <managed-bean-name>userNameBean</managed-bean-name>
  <managed-bean-class>hello.UserNameBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
...
```

The JavaServer Faces configuration file specifies when the application should navigate from the greeting to the response page and creates a bean that contains the user name and stores it in the request scope.

3

Getting Started With Tomcat

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 the Examples (page xxv), then continue with this section.

Getting the Example Code

Once you've installed the tutorial, you'll find the source code for this example in `<INSTALL>/jwstutorial13/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/webservicestuto-
rial.html
```

The example application contains a JavaBeans component, a Web component, a file to build and run the application, a build properties file, 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.

Organizing Web Applications

In this example application, the source code directories are organized in a way that reflects good programming practices for Web services programming. This method of organization is described in more detail in the document at `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/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 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, `<INSTALL>/jwstutorial13/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 JSP page that contains components that enable you to enter the value to be converted, click the button to submit the value, and display 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 the `Ant` tool uses to build and deploy the Web application. This build file calls targets common to many of the example applications from a build file named *<INSTALL>*`/jwstutorial13/examples/common/targets.xml`.
- `build.properties`—The file that contains properties unique to this application. There is also an *<INSTALL>*`/jwstutorial13/examples/common/build.properties` file that contains properties unique to your installation.

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 File, page 57, the `Ant` development tool makes the creation and processing of this type of directory hierarchies relatively simple. In this example, when we run the `ant build` target, the target creates a directory structure that is separate from the source code and is organized in the directory hierarchy required by Tomcat for unpacked applications. In this example, the `build` directory contains the following files and directories:

```
build/
    index.jsp
    WEB-INF/
        web.xml
        classes/
            converterApp/
                ConverterBean.class
```

The rest of this document shows how this example application is created, built, deployed, and run.

Note: The sections Setting the PATH Variable, page 52 and Modifying the Build Properties File, page 52 discuss getting your environment setup for running this example. Whether you want to work through creating the example or just run the existing example application, you must follow the steps in these sections first.

Setting the PATH Variable

Most of the tutorial examples are distributed with a configuration file for Ant, a portable build tool included with the Java WSDP. The version of Ant shipped with the Java WSDP sets the jwsdp.home environment variable to the location of your Java WSDP installation. This variable is used by the example build files.

It is very important that you add the bin directories of the Java WSDP, J2SE SDK, and Ant installations to the front of your PATH environment variable so that the Java WSDP startup scripts for Ant and Tomcat override other installations. The path to the Ant installation that ships with the Java WSDP is *<JWSDP_HOME>*/apache-ant/bin/.

Modifying the Build Properties File

In order to invoke many of the Ant tasks, you need to edit a file named build.properties in the *<INSTALL>*/jwstutorial13/examples/common/ directory.

The build.properties file must contain a user name and password in plain text format that matches either the user name and password set up during installation or a name added subsequent to installation that is assigned the role of manager. In case you've forgotten, the user name and password that you entered during installation of the Java WSDP are stored in *<JWSDP_HOME>*/conf/tomcat-users.xml. Information on adding users is provided in Managing Roles and Users, page 954.

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

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

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 determined by the roles assigned to you in `tomcat-users.xml`) before granting you access to the server.

In addition to specifying your user name and password in the `build.properties` file, you also need to specify the directory location where the tutorial is installed. Define the variable `tutorial.home` by entering the fully-qualified path to the directory into which you installed the tutorial, for example, `/home/your_name/jwsdp-1.3/docs` or `c:/jwsdp-1.3/docs`. Note that the direction of the slash character is important in this file. The slash must be the "/" character for the build files to work properly. This is true on both the Windows and Unix platforms.

You must edit the `build.properties` file to deploy any of the example applications onto Tomcat. Open the *<INSTALL>*`/jwstutorial13/examples/common/build.properties` file and modify the following lines:

```
username=<your_username>
password=<your_password>
tutorial.home=<path_to_dir_into_which_tutorial_was_installed>
```

If you are running on a different Web server and/or port, modify the default tutorial settings for those in this file as well.

Running the Application

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

1. Download the tutorial examples and set up your environment as discussed in Getting the Example Code, page 49, Setting the PATH Variable, page 52, and Modifying the Build Properties File, page 52.
2. From a terminal window or command prompt, change to the root directory for this application, which is *<INSTALL>*`/jwstutorial13/examples/gs/` (see Creating a Simple Web Application, page 54).
3. Compile the source files by typing the following at the terminal window or command prompt (see Building the Example Application, page 60):

   ```
   ant build
   ```

Compile errors are listed in Compilation Errors, page 68.

4. Start Tomcat. If you need help doing this, see Starting Tomcat, page 63. Tomcat startup errors are discussed in Errors Starting Tomcat, page 67.
5. Install the Web application on Tomcat using `Ant` by typing the following at the terminal window or command prompt (see Installing the Web Application, page 62).

   ```
   ant install
   ```

   Installation and deployment errors are discussed in Installation and Deployment Errors, page 70.
6. Start a Web browser. Enter the following URL to run the example application (see Running the Getting Started Application, page 64):

   ```
   http://localhost:8080/gs
   ```
7. Shutdown Tomcat. See Shutting Down Tomcat, page 65 if you need assistance with this.
   - On the Unix platform, type the following at the terminal window:

     ```
     <JWSDP_HOME>/bin/shutdown.sh
     ```
   - On the Microsoft Windows platform, stop Tomcat from the Start menu by following this chain: Start→Programs→Java Web Services Developer Pack 1.3→Stop Tomcat.

Creating a Simple Web Application

The example application contains the following pieces:

- A JavaBeans component
- A Web component
- A build properties file
- A file to build and run the application
- 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.

Creating the JavaBeans Component

The ConverterBean component used in the example application is used in conjunction with a JSP page. The resulting application runs in a Web browser and enables you to convert American dollars to Yen, and convert Yen to Euros. The source code for the ConverterBean component is in *<INSTALL>*/jwstutorial13/examples/gs/src/converterApp/ConverterBean.java.

The ConverterBean class 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 ("110.97");
    euroRate = new BigDecimal (".0078");
    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);
  }
}
```

Creating a Web Client

The Web client is contained in the JSP page `<INSTALL>/jwstutorial13/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.

The JSP page 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. This example has been updated for this release to use JSP 2.0, JSTL, and expression language expressions instead of scripting expressions. For more information on JSP syntax, see Chapter 16.

Here is the source code for `index.jsp`:

```
<%-- index.jsp --%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions"
prefix="fn" %>

<%@ page import="converterApp.ConverterBean" %>
<%@ page contentType="text/html" %>

<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 JSP page</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>

<c:if test="${!empty param.amount &&
```

```
fn:length(param.amount) > 0}" >
   <p><FONT FACE="ARIAL" SIZE=10>${param.amount} dollars are
   <jsp:setProperty name="converter" property="yenAmount"
   value="${param.amount}" />
   ${converter.yenAmount} Yen.
   <p>${param.amount} Yen are
   <jsp:setProperty name="converter" property="euroAmount"
   value="${param.amount}" />
   ${converter.euroAmount} Euro. </FONT>
</c:if>
</body>
</html>
```

Creating the Build Properties File

In this release of the Java WSDP, this tutorial uses two build properties files. One of the build properties files contains properties that will be used by the `Ant` targets for many of the example applications included with this tutorial. Rather than reproduce this information in every application, a common `build.properties` file is used. This file can be found at *<INSTALL>*`/jwstutorial13/examples/common/build.properties`. As discussed in Modifying the Build Properties File, page 52, you must edit this file and provide information regarding your user name and password and the directory from which you've installed the tutorial.

The other build.properties file, *<INSTALL>*`/jwstutorial13/examples/gs/build.properties`, is in the application directory. This file contains properties specific to this application that will be passed to the `Ant` targets. This file does not require modification. The `build.properties` file for the Converter application looks like this:

```
context.path=${example}
example.path=${tutorial.install}/examples/${example}
build=${tutorial.install}/examples/${example}/build
```

The variable `example` is defined in the `build.xml` file. The variable `tutorial.install` is defined in the `common/build.properties` file.

Creating the Build File

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://apache.org`). Documentation for the `Ant` tool can be found at `http://ant.apache.org/manual/index.html`.

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, make sure you add *<JWSDP_HOME>*`/apache-ant/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.

Many of the example applications shipped with this release of the Java WSDP Tutorial use the same `Ant` targets. To simplify development, each application has its own `build.xml` file in its project source directory. The `build.xml` file in the project source directory is fairly simple. It sets some properties specific to this example and includes only one target, which is the target for building the Java source code and copying it to the correct location for deployment. It also tells `Ant` where to find the properties used in the build files and points to other files that contain common `Ant` targets.

The *<INSTALL>*`/jwstutorial13/examples/gs/build.xml` file looks like this:

```
<!DOCTYPE project [
  <!ENTITY targets SYSTEM "../common/targets.xml">
  <!ENTITY webtargets SYSTEM "../web/common/targets.xml">
]>

<project name="gs-example" default="build" basedir=".">
  <target name="init">
```

```
    <tstamp/>
  </target>

  <!-- Configure the context path for this application -->
    <property name="example" value="gs" />

  <!-- Configure properties -->
    <property file="../common/build.properties"/>
    <property file="build.properties"/>

  &targets;
  &webtargets;

  <!-- Application-Specific Targets -->

  <target name="build" depends="copy"
      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/lib">
      <fileset dir="${jwsdp.home}/jstl/lib">
      <include name="*.jar" />
      </fileset>
    </copy>
  </target>

</project>
```

To see the common build targets, you can do either of these options:

1. Look at the files in `<INSTALL>/jwstutorial13/examples/common/targets.xml` and `<INSTALL>/jwstutorial13/examples/web/common/targets.xml`.
2. In the project directory, run the command `ant -projecthelp`.

Creating the Deployment Descriptor

Certain aspects of Web application behavior can be configured when the application is installed or *deployed* to the Web container. The configuration information is maintained in a text file in XML format called a *Web application deployment descriptor*. A Web application deployment descriptor (DD) must conform to the schema described in the `Java Servlet specification`. For this simple application, the deployment descriptor, `<INSTALL>/jwstutorial13/examples/gs/web/WEB-INF/web.xml`, simply includes a description of the application. For more information on deployment descriptors, refer to Removing Web Applications (page 84) and Security in the Web-Tier, page 949.

The deployment descriptor for this application looks like this:

```
<?xml version="1.0" ?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://
java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

<!-- General description of Currency Converter Application -->
<description>
  This is the Java WSDP 1.3 release of the Getting Started
  with Tomcat application, based on JSP pages technology.
  To run this application, read the instructions in the Getting
  Started with Tomcat chapter of the Java WSDP Tutorial v.1.3,
  which explains which enviroment variables need to be set,
  which properties need to be set, which Ant commands to run,
  and how to run the application in a web browser.  When the
  application is loaded, enter an amount to be converted, then
  click the Submit button. The resulting conversion displays
  below the button.
</description>
</web-app>
```

Building the Example Application

To compile the JavaBeans component (`ConverterBean.java`), copy the generated class file to the appropriate location for deployment as an unpacked WAR, and copy the other files needed for deployment to their appropriate locations, 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 or command prompt, change to the `<INSTALL>/jwstutorial13/examples/gs/` directory.
2. Type the following command to build the Java files (this and the following steps that use `Ant` assume that you have the executable for `Ant` in your path: if not, you will need to provide the fully-qualified path to the `Ant` executable):

   ```
   ant build
   ```

If successful, the following message will echo to your screen:

```
Buildfile: build.xml
init:
prepare:
   [mkdir] Created dir:
         <INSTALL>/jwstutorial13/examples/gs/build
   [mkdir] Created dir:
   <INSTALL>/jwstutorial13/examples/gs/build/WEB-INF
   [mkdir] Created dir:
      <INSTALL>/jwstutorial13/examples/gs/build
      /WEB-INF/classes
   [mkdir] Created dir:
      <INSTALL>/jwstutorial13/examples/gs/build/WEB-INF/lib
   [mkdir] Created dir:
      <INSTALL>/jwstutorial13/examples/gs/build/WEB-INF/tags

copy:
   [copy] Copying 1 file to
      <INSTALL>/jwstutorial13/examples/gs/build
   [copy] Copying 1 file to
      <INSTALL>/jwstutorial13/examples/gs/build/WEB-INF

build:
   [javac] Compiling 1 source file to
      <INSTALL>/jwstutorial13/examples/gs/build/WEB-INF/
   classes
   [copy] Copying 2 files to <INSTALL>/jwstutorial13/
      examples/gs/build/WEB-INF/lib


BUILD SUCCESSFUL
```

As you can see from the message above, when we run the `Ant build` target, the following steps are performed:

- The `<INSTALL>/jwstutorial13/examples/gs/build/` directory and all directories required for deployment in an unpacked form are created. Tomcat allows you to deploy an application in an unpacked directory like this.
- The Java source file (`<INSTALL>/jwstutorial13/examples/gs/src/converterApp/ConverterBean.java`) will be compiled and copied to the appropriate location for deployment (`<INSTALL>/jwstutorial13/exam-`

ples/gs/build/WEB-INF/classes/converterApp/Converter-Bean.class).

- The JSP page (*<INSTALL>*/jwstutorial13/examples/gs/web/index.jsp) is copied to *<INSTALL>*/jwstutorial13/examples/gs/build/index.jsp.
- The deployment descriptor, *<INSTALL>*/jwstutorial13/examples/gs/web/WEB-INF/web.xml, is copied to *<INSTALL>*/jwstutorial13/examples/gs/build/WEB-INF/web.xml.

Installing the Web Application

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 targets discussed in Creating the Build File, page 57.

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

You install an application at a specified context and notify Tomcat of this new context with the Ant install task. Read Installing Web Applications, page 78 for more information on this procedure. Using the Ant install task, you can specify an unpacked directory or a WAR file. Alternatively, you can use the Ant deploy task to deploy WAR files as discussed in Deploying Web Applications, page 80.

In this example, we are installing the application into an unpacked directory. The Ant install task used by this application can be found in *<INSTALL>*/jwstutorial13/examples/common/targets.xml, and looks like this:

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

Starting Tomcat

You must start Tomcat before you can install this application using the `Ant` tool. To start Tomcat, select the method that is appropriate for your platform:

- On the Unix platform, type the following at the terminal window:

  `<JWSDP_HOME>/bin/startup.sh`

- On the Microsoft Windows platform, start Tomcat from the Start menu by selecting: Start→Programs→Java Web Services Developer Pack 1.3→ Start Tomcat.

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 Java WSDP index page displays, you may continue. If the index page does not load immediately, wait up to several minutes and then retry. If, after several minutes, the index page does not display, refer to the troubleshooting tips in "Unable to Locate the Server localhost:8080" Error, page 67.

Documentation for Tomcat can be found at `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/index.html`. Information on errors commonly occurring during Tomcat startup, see Errors Starting Tomcat, page 67.

Installing the Application

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 in this example is an unpacked directory. To install this application, follow these steps:

1. In a terminal window or command prompt, change to the `<INSTALL>/jwstutorial13/examples/gs` directory.
2. Type the following command to install the application on Tomcat (this step assumes that you have the executable for `Ant` in your path: if not, you will need to provide the fully-qualified path to the `Ant` executable):

   ```
   ant install
   ```

   If the installation is successful, the following message will echo to your screen:

   ```
   [install] OK - Deployed application at context path /gs
   ```

Running the Getting Started Application

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

```
http://localhost:8080/gs
```

The examples in this tutorial assume that you are running the default implementation of Tomcat and thus the server host and port are `localhost:8080`. The host `localhost` refers to the machine on which Tomcat is running. The `localhost` in this example assumes you are running the example on your local machine as part of the development process. The `8080` in this example is the port where Tomcat was installed during installation. If you are using a different server or port, modify this value accordingly.

The application should display in your Web browser. If there are any errors, refer to Common Problems and Their Solutions, page 67. To test the application,

1. Enter 1000 in the “Enter an amount to convert” field.
2. Click Submit.

Figure 3–1 shows the running application.

Figure 3–1 ConverterBean Web Client

Shutting Down Tomcat

When you are finished testing and developing your application, you should shut down Tomcat. Shut down Tomcat by one of the following methods, depending on which platform you are running:

- On the Unix platform, type the following at the terminal window:

```
<JWSDP_HOME>/bin/shutdown.sh
```

- On the Microsoft Windows platform, stop Tomcat from the Start menu by following this chain: Start→Programs→Java Web Services Developer Pack 1.3→Stop Tomcat.

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 or reload the application. The tasks we defined in the `build.xml` file make it simple to deploy changes to both the JavaBeans component and the JSP page.

In the `targets.xml` file, we have included targets for the following `Ant` tasks that support iterative development. To view a listing of all targets for this application, enter `ant -projecthelp`.

- `ant list`—use this task to view a list of all applications currently available on Tomcat.

3. `ant install`—use this task to install a Web application at a specified context and notify Tomcat that the new application is available. The install task can reference either an unpacked directory or a WAR file.

- `ant reload`—use this task to update an application in the server when the application was initially installed using the `Ant install` task, or to reload a changed Web component installed from an unpacked directory and not one installed from a WAR file.
- `ant remove`—use this task to take an installed Web application out of service.
- `ant deploy`—use this task to permanently deploy a context to Tomcat while Tomcat is running. The `deploy` task requires a WAR.
- `ant undeploy`— use this task to take a deployed application out of service.

These targets use the JWSDP Web Application Manager, which is the `manager` Web application. You can use the `Ant` tasks to access the JWSDP Web Application Manager functionality, or you can access the tool directly. For example, to view all of the applications currently installed on Tomcat, to start, stop, remove, or reload any of these applications, to install a WAR file located on the server, or to upload a WAR file to install, use the `html` version of the Application Manager. You can access the HTML version of the Manager by entering the following URL into a Web browser:

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

You will be prompted for a user name and password. This can be 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, or it can be a user name and password combination that you've set up subsequent to installation as long as it has been assigned the role of `manager`. 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. For more information on using the JWSDP Web Application Manager, read Appendix B.

Modifying a JavaBeans Component

If you want to make changes to the JavaBeans component, you change the source code, recompile it, and reload the application onto Tomcat. When using Tomcat, its `manager` Web application enables you to update an application in the server without the need to stop and restart Tomcat. 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. Reinstall the `ConverterBean` component by typing `ant reload`.
4. Reload the JSP page in the Web browser.

Modifying the Web Client

If you want to make changes to a JSP page, you change the source code and reload the application using the `reload` task. When using Tomcat, its `manager`

Web application enables you to reinstall the changed Web client in the server without the need to stop and restart Tomcat. 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 Java WSDP index page 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, 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 *<INSTALL>*/jwstutorial13/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. When you specify the build file in this way, you must specify the fully-qualified path to the file, not just a relative path. For example, you would type this command on a single line:

```
ant -buildfile <INSTALL>/jwstutorial13/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. To verify which version you are using, enter ant -version at the terminal window or command prompt. Java WSDP 1.3 uses Ant version 1.5.4. If this is not the version that displays at the version request, 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>*/apache-ant/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 window or command prompt, you get the following message:

```
file:<INSTALL>/jwstutorial13/examples/common/targets.xml:28:
java.net.ConnectException: Connection refused: connect
```

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 67.

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

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

Solution: Start Tomcat. For information on doing this, see Starting Tomcat, page 63.

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 Java WSDP index page displays, you may continue. If the Java WSDP index page 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, 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`.

Installation and Deployment Errors

Server Returned HTTP Response Code: 401 for URL

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

```
BUILD FAILED
<INSTALL>/jwstutorial13/examples/common/targets.xml:42:
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 `<INSTALL>/jwstutorial13/examples/common/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 Modifying the Build Properties File, page 52.

Failure to Run Client Application

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

Solution: If you have restarted Tomcat since the last time you installed the application, you need to install the application again now that Tomcat is restarted. For information on installing the application, see Installing the Web Application, page 62.

Solution: Start Tomcat. If you need more information on starting Tomcat, see Starting Tomcat, page 63.

Solution: If you've already started Tomcat, note that 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 67.

Solution: Make sure that you have the slashes in the right direction for the `tutorial.home` property in the `<INSTALL>/jwstutorial13/examples/common/build.properties` file. The slashes should go to the right, `/`, otherwise, the build file does not recognize this parameter and copies files to the wrong location for deployment.

The localhost Machine Is Not Found

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

Solution: This may happen 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, then check the Bypass Proxy Server For Local Addresses checkbox.

The Application Has Not Been Deployed

Symptom: The browser reports that the page cannot be found (HTTP 404) or the requested resource is not available.

Solution: Install the application. For more detail, see Deploying Web Applications, page 80 for general information on installing Web applications or Installing the Web Application, page 62 for information on starting the Getting Started example. If the application says that it was installed but you are still getting this message, it is likely that the build file is not copying the files to the correct location as expected by Tomcat. Check that the files are being copied to the proper directories by verifying that the build properties are correct as discussed in Modifying the Build Properties File, page 52 or Failure to Run Client Application, page 70.

"Build Failed: Application Already Exists at Path" Error

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

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

BUILD FAILED

<INSTALL>/jwstutorial13/examples/common/targets.xml:28: FAIL -
Application already exists at path /gs
```

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 build`, `ant install`, 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.

Solution: If the application says that it was installed but you are still getting this message, it is likely that the build file is not copying the files to the correct location as expected by Tomcat. Check that the files are being copied to the directories as discussed in Modifying the Build Properties File, page 52 or Failure to Run Client Application, page 70.

No Error Messages, Page Loads as Blank Page

Symptom: After you install the application, you enter the correct URL in your Internet Explorer Web browser. Although no error messages display, the application does not load in the browser.

Solution: On some Web browsers, you have to set the encoding to `UTF-8` in order for the application to display. Here's how to do that:

- Internet Explorer: Select View→ Encoding→Unicode.
- Netscape Navigator: Select View→Character Set→Western.

Load Cyrillic Language Pack

Symptom: After you install the application, you enter the correct URL in your Internet Explorer Web browser, and are prompted to load the Cyrillic Language Pack.

Solution: On some version of Microsoft Internet Explorer Web browsers, you have to set the encoding to `UTF-8` or `Western European` in order for the application to display properly. To do this, follow this sequence: View→Encoding→UTF-8. Click the Refresh button and the application should display in the browser window.

4

Getting Started with Web Applications

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

In the Java 2 platform, *Web components* provide the dynamic extension capabilities for a Web server. Web components are either Java Servlets, JSP pages, or Web service endpoints. 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*. 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.

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

This chapter describes the organization, configuration, and installation and deployment procedures for Web applications. Chapters 8 and 9 cover how to develop Web components for service-oriented Web applications. The technologies for developing service-oriented Web applications were introduced in Chapter 1. Chapters 11 through 18 cover how to develop the Web components for presentation-oriented Web applications. The technologies for developing presentation-oriented Web applications were introduced in Chapter 2.

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 Appendix E.

Web Application Life Cycle

A Web application consists of Web components, static resource files such as images, and helper classes and libraries. The Web container 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.The process for creating, deploying, and executing a Web application can be summarized as follows:

1. Develop the Web component code.
2. Develop the Web application deployment descriptor.
3. Compile the Web application components and helper classes referenced by the components.

4. Optionally package the application into a deployable unit.
5. Deploy the application into a Web container.
6. 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):

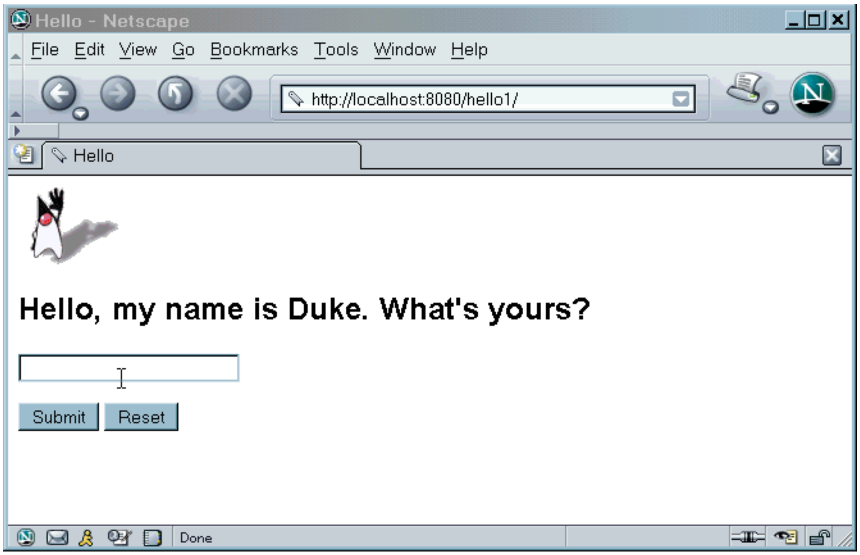


Figure 4–1 Greeting Form

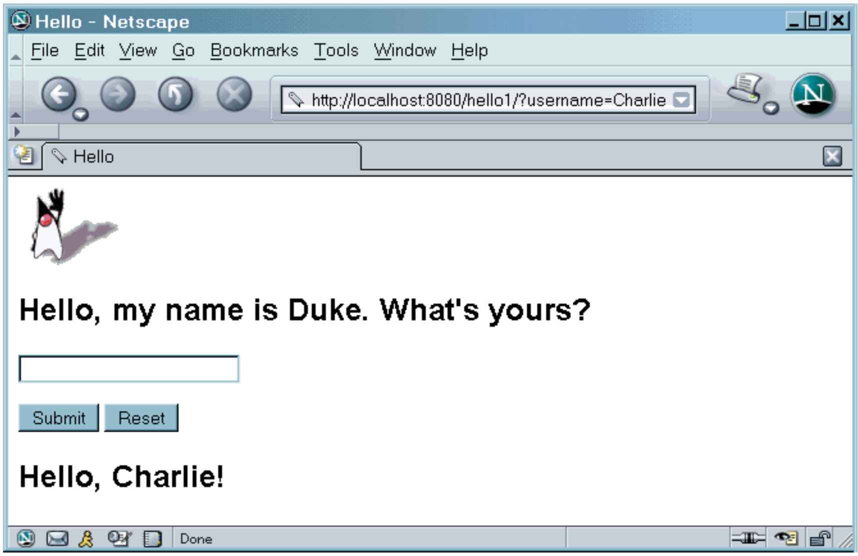


Figure 4–2 Response

The Hello application contains two Web components that generate the greeting and the response. This chapter discusses two versions of the application: a JSP version called `hello1`, in which the components are implemented by two JSP pages, `index.jsp` and `response.jsp`, and a servlet version called `hello2`, in which the components are implemented by two servlet classes, `GreetingServlet.java` and `ResponseServlet.java`. The two versions are used to illustrate tasks involved in packaging, deploying, configuring, 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 About the Examples (page xxv).

Web Applications

In the J2EE architecture, Web components and static Web content files such as images are called *Web resources*. A *Web application* is the smallest deployable and usable unit of Web resources.

In addition to Web components and Web resources, a Web application can contain other files including:

- Server-side utility classes (database beans, shopping carts, and so on). Often these classes conform to the JavaBeans component architecture.
- Client-side classes (applets and utility classes).

A Web application has a specific structure. The top-level directory of a Web application is the *document root* of the application. The document root is where JSP pages, *client-side* classes and archives, and static Web resources, such as images, are stored.

The document root contains a subdirectory named `/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 728)
- `classes`—A directory that contains *server-side classes*: servlets, utility classes, and JavaBeans components
- `tags`—A directory that contains tag files, which are implementations of tag libraries (see Tag File Location, page 715)
- `lib`—A directory that contains JAR archives of 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.

A Web applicationcan be deployed as an unpacked file structure or packaged in a JAR file known as a Web archive (WAR) file. Since the contents and use of WAR files differ from JAR files, WAR file names use a `.war` extension.

The Web module just described is portable; you can deploy it into any Web container that conforms to the Java Servlet Specification.

Packaging Web Applications

A Web application must be packaged into a WAR in certain deployment scenarios and whenever you want to distribute the Web application. You package a Web application into a WAR by executing the `jar` command in a directory laid out in the format of a Web module or by using the `Ant war` task. This tutorial uses the second approach. To build and package the `hello1` application into a WAR named `hello1.war`:

1. In a terminal window, go to *<INSTALL>*`/jwstutorial13/examples/web/hello1/`.
2. Run `ant build`. This target will spawn any necessary compilations and copy files to the *<INSTALL>*`/jwstutorial13/examples/web/hello1/build/` directory.
3. Run `ant package`. This target creates a WAR in the directory *<INSTALL>*`/jwstutorial13/examples/web/hello1/dist/`.

A sample `hello1.war` is provided in *<INSTALL>*`/jwstutorial13/examples/web/provided-wars/`.

If you have deployed the `hello1` application, before proceeding with this section, undeploy the application by following one of the procedures described in Undeploying Web Applications (page 85).

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 into Tomcat, you notify Tomcat that a new context is available. Installing an application is the recommended operation when you are iteratively developing an application because you do not have to package the Web application and because you can quickly reload a modified application. If you need to install an application on a remote machine, you need to use the deploy operation (see Deploying Web Applications, page 88).

You install an application into Tomcat with the manager application `install` command invoked via the `Ant install` task. The `Ant install` task tells the manager application running at the location specified by the `url` attribute to install an application at the context specified by the `path` attribute. In addition, you must indicate the location of the Web application files. There are two ways to do this:

- Directory path to a unpacked or packaged Web application. Note that reload only applies to an application installed from an unpacked directory and not one installed from a WAR file.
- Configuration file

To specify the directory path, you use 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 Appendix B.

You can specify the location of the Web application files via a configuration file with the `config` attribute:

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

The configuration file contains a context entry of the form:

```
<Context path="/bookstore1"
  docBase="../../jwstutorial13/examples/web/bookstore1/build"
  debug="0">
```

The format of a context entry is described in the *Server Configuration Reference* at `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/config/context.html`. The context entry specifies the location of the Web application files through its `docBase` attribute.

Note: The setting of the `docBase` attribute in the tutorial examples contains a path to the example build directory that's defined relative to the `<JWSDP_HOME>/webapps/` directory:

```
../../jwstutorial13/examples/web/bookstore1/build/
```

This setting assumes that you have installed the tutorial in the same directory as the Java WSDP. If you install the tutorial in another directory, you will need to adjust the `docBase` attribute so that it reflects the path between the `webapps` directory and the example build directory. For example, if you install the tutorial in the Java WSDP install directory, the `docBase` attribute should be changed to `../jwstutorial13/examples/web/bookstore1/build/`.

The tutorial example build files contain `Ant install` and `install-config` *targets* that invoke the `Ant install` *task*:

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

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

To install the `hello1` application described in Web Application Life Cycle (page 74):

1. In a terminal window, go to `<INSTALL>/jwstutorial13/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 also notify Tomcat of a new context with the manager application `deploy` command invoked via the `Ant deploy` task. You use the `Ant deploy` task to deploy a context to a remote instance of Tomcat.

```
<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 `deploy` task:

- Uploads the WAR to Tomcat, which puts it in the `<JWSDP_HOME>/webapps/` directory
- Starts the application

The following additional 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>/work/Catalina/localhost/`.

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

To deploy the servlet version of the hello application:

1. In a terminal window, go to `<INSTALL>/jwstutorial13/examples/web/hello2/`.
2. Run `ant build`. The `build` target will spawn any necessary compilations and copy files to the `<INSTALL>/jwstutorial13/examples/web/hello2/build/` directory.
3. Run `ant package`. This task packages the application into a WAR file named `hello2.war`.
4. Run `ant deploy`. The `deploy` target uploads the WAR file to `<JWSDP_HOME>/webapps/` and starts the application.

Listing Installed and Deployed Web Applications

You can list the installed and deployed applications by running the Manager Application in a browser:

```
http://localhost:8080/manager/html
```

You can also use the manager application `list` command invoked via `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*.

Updating Web Applications

A typical iterative development cycle involves installing or deploying a Web application and then making changes to the application components. In order to update an installed or deployed Web application, you must:

1. Recompile any modified classes.
2. If you have installed or deployed a packaged Web application, update any modified components in the WAR.
3. Reload or deploy the module.
4. Reload the URL in the client.

The procedure for updating an 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 reload` *target* that invokes the `Ant reload` *task*.

When you update a JSP page, you do not need to reload the application in the server, because Tomcat does this automatically. 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 84) and install it again.

Redeploying Web Applications

If you have deployed the application using the `Ant deploy` task you repackage the WAR, undeploy it from the server (see Undeploying Web Applications, page 85) , and then deploy it again.

To try this, 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>
```

First modify the file:

1. Edit `GreetingServlet.java` in the source directory `<INSTALL>/jwstutorial13/examples/web/hello1/src/`.
2. Run `ant build`. This task recompiles the servlet into the `build` directory.
3. Execute `ant package`.

Then, to update the `Hello2` application in the server:

1. Execute `ant undeploy`.
2. Execute `ant deploy`.

To view the updated application, reload the `Hello2` URL in the client. You should see the screen in Figure 4–3 in the browser:

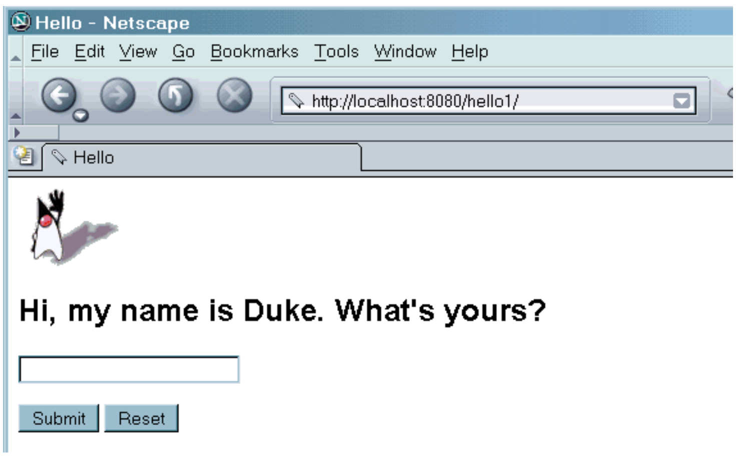


Figure 4–3 New Greeting

Debugging Web Applications

Using the Server Log

One way to debug Web applications is to look at the server log in `<JWSDP_HOME>/logs/jwsdp_log.date.txt`. The log contains output from the Tomcat server and your Web applications. You can log messages from any Java class your Web modules with `System.out.println` and the Java Logging APIs (documented at `http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/index.html`) and from Web components with the `ServletContext.log` method.

Using a Debugger

Tomcat supports the Java Platform Debugger Architecture (JPDA). Using JPDA, you can configure the server to communicate debugging information via a socket. In order to debug a Web application using a debugger:

1. Start Tomcat in a terminal window or command prompt by executing `catalina jpda run`. The debugger will listen on port 8000.
2. Compile your Java source with the `-g` flag.
3. Installyour Web application.
4. Start a debugger and connect to the debugger socket at port 8000.

Removing Web Applications

If you want to take an installed Web application out of service, you use manager application `remove` command invoked 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 manager application `undeploy` command via 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*.

Configuring Web Applications

Web applications are configured via elements contained in the Web application deployment descriptor. 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 Specifying Security Constraints (page 955).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 references to other examples that illustrate how to specify the deployment descriptor element and describes generic procedures for specifying the feature.

when you install (see Installing Web Applications, page 78) or deploy (Deploying Web Applications, page 80) a Web application to Tomcat.

Prolog

Since the Web application 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="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://
java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
```

Mapping URLs to Web Components

When a request is received by the Web container 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 application and a Web component. A URL path contains the context root and an alias:

```
http://host:port/context_root/alias
```

Setting the Component Alias

The *alias* identifies the Web component that should handle a request. The alias path must start with a forward slash `/` and end with a string or a wildcard expression with an extension (`*.jsp`, for example). Since Web containers automatically map an alias that ends with `*.jsp`, you do not have to specify an alias for a JSP page unless you wish to refer to the page by a name other than its file name. To set up the mappings for the servlet version of the `hello` application in the Web deployment descriptor, you add the following `servlet` and `servlet-mapping` elements to the Web application deployment descriptor. To define an alias for a JSP page, you replace the `servlet-class` subelement with a `jsp-file` subelement in the `servlet` element.

```
<servlet>
  <display-name>greeting</display-name>
  <servlet-name>GreetingServlet</servlet-name>
  <servlet-class>servlets.GreetingServlet</servlet-class>
</servlet>
<servlet>
  <display-name>ResponseServlet</display-name>
  <servlet-name>ResponseServlet</servlet-name>
  <servlet-class>servlets.ResponseServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>GreetingServlet</servlet-name>
  <url-pattern>/greeting</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>ResponseServlet</servlet-name>
  <url-pattern>/response</url-pattern>
</servlet-mapping>
```

Declaring Welcome Files

The *welcome files* mechanism allows you to specify a list of files that the Web container will use for appending to a request for a URL (called a *valid partial request)* that is not mapped to a Web component.

For example, suppose you define a welcome file `welcome.html`. When a client requests a URL such as *`host:port/webapp/directory`*, where *`directory`* is not mapped to a servlet or JSP page, the file *`host:port/webapp/directory/`*`welcome.html` is returned to the client.

If a Web container receives a valid partial request, the Web container examines the welcome file list and appends each welcome file in the order specified to the partial request and checks whether a static resource or servlet in the WAR is mapped to that request URL. The Web container then sends the request to the first resource in the WAR that matches.

If no welcome file is specified, Tomcat will use a file named `index.`*`XXX`*, where *`XXX`* can be `html` or `jsp`, as the default welcome file. If there is no welcome file and no file named `index.`*`XXX`*, Tomcat returns a directory listing.

The example discussed in Encapsulating Reusable Content using Tag Files (page 714) has a welcome file.

To specify welcome files in the Web application deployment descriptor, you add `welcome-file` elements in the `welcome-file-list` element:

```
<welcome-file-list>
  <welcome-file>greeting.jsp</welcome-file>
</welcome-file-list>
```

Setting Initialization Parameters

The Web components in a Web module share an object that represents their application context (see Accessing the Web Context, page 632). You can pass initialization parameters to the context or to a Web component. To set initialization parameters, 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 :

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

For an example context parameter, see the example discussed in The Example JSP Pages (page 645).

Specifying 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 611). To set up the mapping, you add an `<error-page>` element to the Web application deployment descriptor. Here is the element used to map `OrderException` to the page `errorpage.html` used in Chapter 15:

```
<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. See Handling Errors (page 652).

For an example error page mapping, see the example discussed in The Example Servlets (page 606).

Declaring 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. For an example resource reference, see Configuring the Web Application to Reference a Resource (page 93).

Duke's Bookstore Examples

In chapters 11 through 15, a common example—Duke's Bookstore—is used to illustrate the elements of Java Servlet technology, JavaServer Pages technology, and the JSP Standard Tag Library. The example emulates a simple online shopping application. It provides a book catalog from which users can select books and add them to a shopping cart. Users can view and modify the shopping cart. Once users are finished shopping, they can purchase the books in the cart.

The Duke's Bookstore examples share common classes and a database schema. These files are located in the directory `<INSTALL>/jwstutorial13/examples/web/bookstore/`. The common classes are packaged into a JAR. To create the `bookstore` library JAR:

1. In a terminal window, go to `<INSTALL>/jwstutorial13/examples/web/bookstore/`.
2. Run `ant build` to compile the bookstore files.
3. Run `ant package-bookstore` to create a library named `bookstore.jar` in `<INSTALL>/jwstutorial13/examples/bookstore/dist/`.

The next section describes how to create the bookstore database table and server resources required to run the examples.

Accessing Databases from Web Applications

Data that is shared between Web components and persistent between invocations of a Web application is usually maintained in 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
```

In the JDBC API, databases are accessed via `DataSource` objects. A `DataSource` has a set of properties that identify and describe the real world data source that it represents. These properties include information like the location of the database server, the name of the database, the network protocol to use to communicate with the server, and so on.

Applications access a data source using a connection, and a `DataSource` object can be thought of as a factory for connections to the particular data source that the `DataSource` instance represents. In a basic `DataSource` implementation, a call to the method `DataSource.getConnection` returns a connection object that is a physical connection to the data source.

If a `DataSource` object is registered with a JNDI naming service, an application can use the JNDI API to access that `DataSource` object, which can then be used to connect to the data source it represents.

`DataSource` objects that implement connection pooling also produce a connection to the particular data source that the `DataSource` class represents. The connection object that the method `DataSource.getConnection` returns is a handle to a `PooledConnection` object rather than being a physical connection. An application uses the connection object like any other connection. Connection pooling has no effect on application code except that a pooled connection, like all connections, should always be explicitly closed. When an application closes a connection that is pooled, the connection is returned to a pool of reusable connections. The next time `DataSource.getConnection` is called, a handle to one of these pooled connections will be returned if one is available. Because connection pooling avoids creating a new physical connection every time one is requested, it can help applications run significantly faster.

The Duke's Bookstore examples described in chapters 11 through 15 use the PointBase evaluation database to maintain the catalog of books. See the PointBase support site at

```
http://www.pointbase.com/node.shtml?navHier=Support/
Product+Docs&CF=support/docs/overview.html
```

for detailed information about the PointBase database.

This section describes how to:

- Install and start the PointBase database server
- Populate the database
- Define a data source in the application server
- Configure a Web application to reference the data source with a JNDI name
- Map the JNDI name to the data source defined in the application server

Installing and Starting the PointBase Database Server

You can download an evaluation copy of the PointBase 4.7 database from:

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

After you register, select the PointBase Embedded - Server Option version of the software and 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 xxvii) 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/pbclient47.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.

To start the PointBase database server:

1. In a terminal window, go to `<PB_HOME>/tools/serveroption`.
2. Execute the `startserver` script.

Populating the Example Database

To populate the database for the Duke's Bookstore examples:

1. In a terminal window, go to `<INSTALL>/jwstutorial13/examples/web/bookstore/`.
2. Run `ant create-pointbase-db`. This task runs a PointBase commander tool command to read the file `books.sql` and execute the SQL commands contained in the file. The table named `books` is created for the user `pbpublic` in the `sample` PointBase database.
3. At the end of the processing, you should see the following output:

```
...
[java] SQL> INSERT INTO books VALUES('207', 'Thrilled', 'Ben',
[java]  'The Green Project: Programming for Consumer Devices',
[java]  30.00, false, 1998, 'What a cool book', 20);
[java] 1 row(s) affected

[java] SQL> INSERT INTO books VALUES('208', 'Tru', 'Itzal',
[java]  'Duke: A Biography of the Java Evangelist',
[java]  45.00, true, 2001, 'What a cool book.', 20);
[java] 1 row(s) affected
```

You can check that the table exists with the PointBase console tool as follows:

1. In a terminal window, go to `<PB_HOME>/tools/serveroption/`.
2. Execute `startconsole`.
3. In the Connect to Database dialog
   a. Enter `jdbc:pointbase:server://localhost/sample` in the URL field.
   b. Enter `PBPUBLIC` in the password field.
4. Click OK.
5. Expand the SCHEMAS→PBPUBLIC→TABLES nodes. Notice that there is a table named BOOKS.

6. To see the contents of the books table:
   a. In the Enter SQL commands text area, enter `select * from books;`.
   b. Click the Execute button.

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 1077) 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 pbpublic in the User Name and Password fields.
9. Click the Save button.
10. Click the Commit button.

Configuring the Web Application to Reference a Resource

In order to access a database from a Web application, you must declare resource reference in the application's Web application deployment descriptor (see Declaring References to Environment Entries, Resource Environment Entries, or Resources, page 89). The resource reference declares a JNDI name, the type of the data resource, and the kind of authentication used when the resource is accessed. The JNDI name is used to create a data source object in the database helper class `database.BookDB`:

```
public BookDB () throws Exception {
  try  {
     Context initCtx = new InitialContext();
     Context envCtx = (Context)
        initCtx.lookup("java:comp/env");
     DataSource ds = (DataSource) envCtx.lookup("jdbc/BookDB");
     con =  ds.getConnection();
     System.out.println("Created connection to database.");
  } catch (Exception ex) {
     System.out.println("Couldn't create connection." +
     ex.getMessage());
     throw new
        Exception("Couldn't open connection to database: "
     +ex.getMessage());
  }
```

To specify a resource reference to the bookstore data source, the bookstore deployment descriptors include the following element:

```
15<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.

Mapping the Web Application Resource Reference to a Data Source

The Web application resource reference and the data source defined in the J2EE 1.4 Application server both have JNDI names. You connect the resource reference name to the 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="../../jwstutorial13/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 78) and Deploying Web Applications (page 80), 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-config` task.

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 15, 16, , and 18 show how to install applications using the `Ant` `install-config` task mechanism.

Further Information

For more information about Web applications, refer to the following:

- Java Servlet 2.4 Specification
  `http://java.sun.com/products/servlet/download.html#specs`
- The Java Servlets Web site
  `http://java.sun.com/products/servlet`

5

Understanding XML

THIS chapter describes the Extensible Markup Language (XML) and its related specifications. It also gives you practice in writing XML data, so you become comfortably familiar with XML syntax.

Note: The XML files mentioned in this chapter can be found in `<INSTALL>/jwstutorial13/examples/xml/samples`.

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 150), 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 109), 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 105), 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 Java-platform programs 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 114) initiative, which defines "meta tags", and the XSL (page 109) specification, which lets you translate XML tags into other XML tags.

Document-Driven Programming

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 Java Beans components for interfaces and Enterprise Java Beans 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 107) and DOM (page 107). The specifications for ensuring the validity of XML documents are DTD (page 108) (the original mechanism, defined as part of the XML specification) and various Schema Standards (page 110) 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 109) 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 109), is completed and covered in this tutorial. Another effort nearing completion is the XML Link Language specification (XML Linking, page 112), 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 113), and the meta-standard for describing the information an XML document contains, RDF (page 114). 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 115), Mathematical Markup Language (MathML, page 115), Scalable Vector Graphics (SVG, page 115), and DrawML (page 115), 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 106)
- Schema Standards (page 110)
- Linking and Presentation Standards (page 112)
- Knowledge Standards (page 113)
- Standards That Build on XML (page 115)

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).

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.

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 (JCP) 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 110) 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.

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.)

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 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, you only get to specify the structure of the <heading> element one time. There is no context-sensitivity, because a DTD specification is not hierarchical.

For example, for a mailing address that contains several "parsed character data" (PCDATA) elements, 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. So if you need another "name" element in the DTD, you need a different identifier for it. 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 might be intended to apply to the whole structure, or it might be intended only for the first item. 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 is 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 virtually any relationship you can conceive. The bad news is that it is very difficult to implement, and it's hard to learn. Most of the alternatives provide for simpler structure definitions, while incorporating XML Schema's datatyping mechanisms.

For more information on 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 (Next Generation)

Simpler than XML Structure Schema, RELAX NG is an emerging standard under the auspices of OASIS (Organization for the Advancement of Structured Information Systems). It may become an ISO standard in the near future, as well.

RELAX NG uses regular expression patterns to express constraints on structure relationships, and it is uses XML Schema datatyping mechanisms to express content constraints. This standard also uses XML syntax, and it includes a DTD to RELAX converter. (It's "next generation" because 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 Language

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.

Generating XML Data

This section also takes you step by step through the process of constructing an XML document. Along the way, you'll gain experience with the XML components you'll typically use to create your data structures.

Writing a Simple XML File

You'll start by writing 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 1105).)

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 99).

Adding a Comment

Comments are ignored by XML parsers. A program will never see them in fact, unless you activate special settings in the parser. 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 `slideshow` 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>
```

Note 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), along with possible solutions, later on in this tutorial.

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>
```

Save a copy of this file as `slideSample01.xml`, so you can use it as the initial data structure when experimenting with XML programming operations.

Writing 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.

Note: The file you'll create in this section is `slideSample02.xml`. (The browsable version is `slideSample02-xml.html`.)

As you saw in Processing Instructions (page 100), 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.

Save a copy of this file as `slideSample02.xml`, so you can use it when experimenting with processing instructions.

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 browsable version is `slideSampleBad1-xml.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 `slideSample02.xml` to `slideSampleBad1.xml`.
2. Edit `slideSampleBad1.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>
...
```

Now you have a file that you can use to generate an error in any parser, any time. (XML parsers are required to generate a fatal error for this file, because the lack of an end-tag for the <item> element means that the XML structure is no longer *well-formed*.)

Substituting and Inserting Text

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 `&` 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 programming language. Table 5–1 shows the predefined entities for special characters.

Table 5–1 Predefined Entities

Character	Reference
&	&
<	<
>	>
"	"
'	'

Character References

A character reference like “ 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 < in the file, instead of “<”.

Note: The results of the modifications below are contained in `slideSample03.xml`.

Add the text highlighted below to your `slideSample.xml` file, and save a copy of it for future use as `slideSample03.xml`:

```
  <!-- 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 use an XML parser to echo this data, you will see the desired output:

```
Market Size < predicted
```

You see an angle bracket ("<") where you coded "<", because the XML parser converts the reference into the entity it represents, and passes that 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`.

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, and save a copy of the file as `slideSample04.xml`:

```
 ...
<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 echo this file with an XML parser, you'll see the following output:

```
Diagram:
frobmorten <--------------fuznaten
     |       <3>    ^
     | <1>          |   <1> = fozzle
    V               |   <2> = framboze
  staten----------------------+   <3> = frenzle
             <2>
```

The point here is that the text in the `CDATA` section will have arrived as it was written. Since the parser doesn't treat the angle brackets as XML, they don'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.)

Creating a Document Type Definition

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

To begin learning about DTD definitions, let's start by telling the parser where text is expected and where any text (other than whitespace) would be an error. (Whitespace in such locations 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 5–2.

Table 5–2 DTD Element Qualifiers

Qualifier	Name	Meaning
?	Question Mark	Optional (zero or one)

Table 5–2 DTD Element Qualifiers (Continued)

Qualifier	Name	Meaning
*	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.

Save a copy of this DTD as slideSample1a.dtd, for use when experimenting with basic DTD processing.

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 ( ...`, 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, and save a copy of the file as `slideSample05.xml`:

```
<!--  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 in a moment to define some entities that can be used in the document.

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.

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 5–3 presents the valid choices for the attribute type.

Table 5–3 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 5–4 shows the possible choices.

Table 5–4 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.

Finally, save a copy of the DTD as `slideshow1b.dtd`, for use when experimenting with attribute definitions.

Defining Entities in the DTD

So far, you've seen predefined entities like `&` 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 you'll create here is contained in `slideSample06.xml`. (The browsable version is `slideSample06-xml.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.

When you echo this version of the file with an XML parser, here is the kind of thing you'll see:

```
Wake up to WonderWidgets!
```

Note that the product name has been substituted for the entity reference.

To finish, save a copy of the file as `slideSample06.xml`.

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 browsable versions are `slideSample07-xml.html` and `copyright-xml.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, and save a copy of the file as slideSample07.html:

```
<!-- 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;.
```

When you echo the latest version of the slide presentation with an XML parser, here is what you'll see:

```
...
<slide type="all">
  <item>
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...
  </item>
</slide>
...
```

You'll notice 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 `<item>` element, instead of on the same line—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 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`. It shows how binary references can be made, assuming 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 `slideshow1b.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 `http://www.iana.org/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.

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` and `xhtml.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.

Save the DTD as slideshow2.dtd, for use when experimenting with parameter entities.

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.

Resolving A Naming Conflict

The XML structures you have created thus far have actually encountered a small naming conflict. 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 use XML namespaces to resolve the conflict. You'll take a look at that approach in the next section. Alternatively, you could use one of the more hierarchical schema proposals described in Schema Standards (page 110). The simplest way to solve the problem for now, though, is simply to 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 browsable versions are `slideshow3-dtd.html`, `slideSample09-xml.html`, `copyright-xml.html`, and `xhtml-dtd.html`.)

To keep the two title elements separate, you'll create 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;)*>
```

Save this DTD as `slideshow3.dtd`.

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>...
```

Save a copy of this file as `slideSample09.xml`.

Using Namespaces

As you saw earlier, 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.
- 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.

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 web site created by the Organization for the Advancement of Structured Information Standards (OASIS). You can find a list of technical committees at `http://www.oasis-open.org/`, or check their repository of XML standards at `http://www.XML.org`.

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

In Saving Yourself Some Work (page 151), you saw that it is a good idea to define 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. 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. 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.

Summary

Congratulations! You have now created a number of XML files that you can use for testing purposes. Here’s a table that describes the files you have constructed.

Table 5–5 Listing of Sample XML Files

File	Contents
`slideSample01.xml`	A basic file containing a few elements and attributes, as well as comments.
`slideSample02.xml`	Includes a processing instruction.
`SlideSampleBad1.xml`	A file that is *not* well-formed.
`slideSample03.xml`	Includes a simple entity reference (<).
`slideSample04.xml`	Contains a CDATA section.

Table 5–5 Listing of Sample XML Files

File	Contents
`slideSample05.xml`	References either a simple external DTD for elements (`slideshow1a.dtd`), for use with a nonvalidating parser, or else a DTD that defines attributes (`slideshow1b.dtd`) for use with a validating parser.
`slideSample06.xml`	Defines two entities locally (product and products), and references `slideshow1b.dtd`.
`slideSample07.xml`	References an external entity defined locally (`copyright.xml`), and references `slideshow1b.dtd`.
`slideSample08.xml`	References `xhtml.dtd` using a parameter entity in `slideshow2.dtd`, producing a naming conflict, since `title` is declared in both.
`slideSample09.xml`	Changes the `title` element to `slide-title`, so it can reference `xhtml.dtd` using a parameter entity in `slideshow3.dtd` without conflict.

6

Java API for XML Processing

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 Sun's 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.

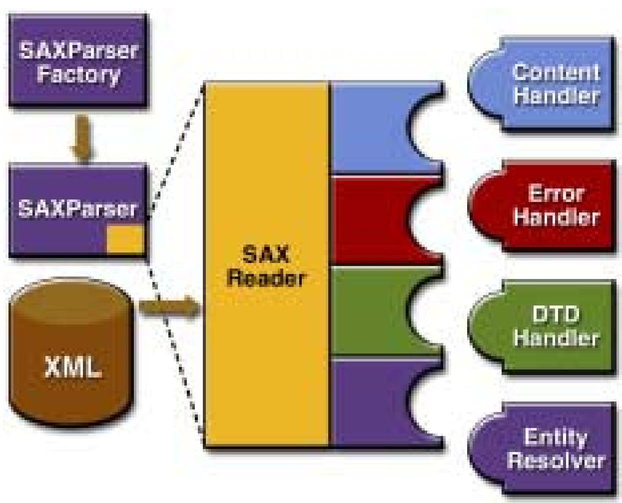


Figure 6–1 SAX APIs

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 6–1.

Table 6–1 SAX Packages

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 6–2 shows the JAXP APIs in action:

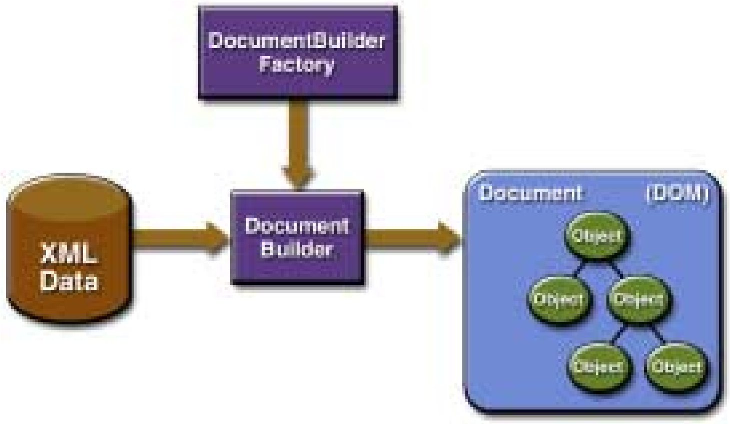


Figure 6–2 DOM APIs

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 6–2.:

Table 6–2 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 6–3 shows the XSLT APIs in action.

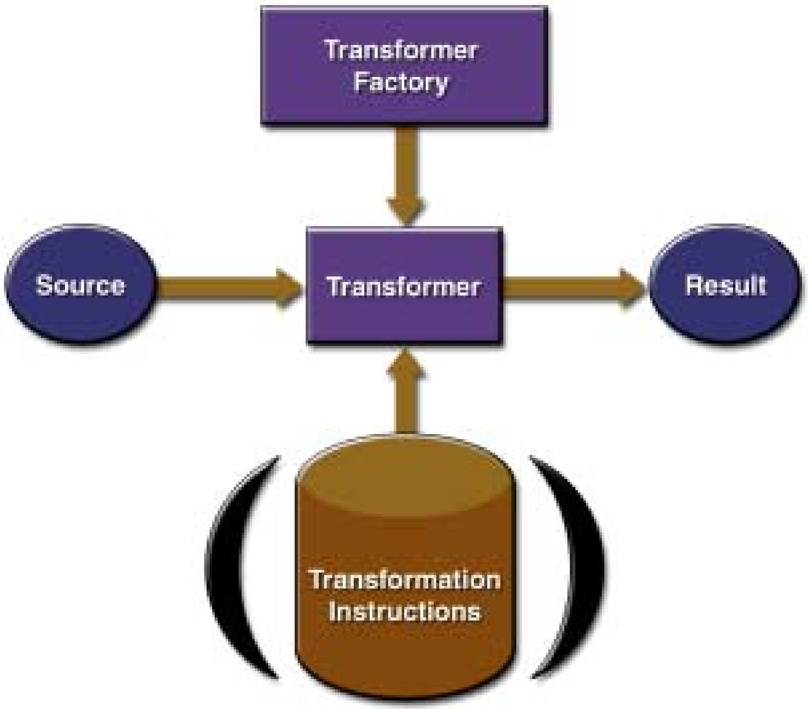


Figure 6–3 XSLT APIs

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 6–3 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 J2EE 1.4 Application Server, the JAXP libraries are distributed in the directory `<J2EE_HOME>/lib/endorsed`. To run the sample programs, you'll need to used the Java 2 platform's "endorsed standards" mechanism to access those libraries. For details, see Compiling and Running the Program (page 181).

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:

Simple API for XML (page 169)

If the data structures have already been determined, and you are writing a server application or an XML filter that needs to do fast processing.

Document Object Model (page 227)

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 301)

If you need to transform XML tags into some other form, if you want to generate XML output, or (in combination with the SAX API) if you want to convert legacy data structures to XML.

7

Simple API for XML

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 227).

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 XML files used in this chapter can be found in `<INSTALL>/jwstutorial13/examples/xml/samples`.
The programs and output listings can be found in `<INSTALL>/jwstutorial13/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 320), 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 228).)

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 107).

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`, as described in Writing a Simple XML File (page 117). (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 1105).

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 193).

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 “&” or “<”, as appropriate. You’ll find out more about that kind of processing when we discuss entity references in Displaying Special Characters and CDATA (page 201).

Compiling and Running the Program

In the Java WSDP release, the JAXP libraries are in the directory <*JWSDP_HOME*>/jaxp/lib/endorsed. These are newer versions of the standard JAXP libraries that are part of the Java 2 platform.

Tomcat automatically uses the newer libraries when a program runs. So you won’t have to be concerned with where they reside when you deploy an application.

And since the JAXP APIs are identical in both versions, you won’t need to be concerned at compile time either. So compiling the program you created is as simple as issuing the command:

```
javac Echo.java
```

But to run the program outside of the server container, you need to make sure that the java runtime finds the newer versions of the JAXP libraries. That situation can occur, for example, when unit-testing parts of your application outside of the sever, as well as here, when running the XML tutorial examples.

There are two ways to make sure that the program uses the latest version of the JAXP libraries:

- Copy the contents of the <*JWSDP_HOME*>/jaxp/lib/endorsed directory to jdk/jre/lib/endorsed. You can then run the program with this command:

  ```
  <J2SE_HOME>/bin/java Echo slideSample.xml
  ```

  The libraries will then be found in the endorsed standards directory, <J2SE_HOME>/jre/lib/endorsed.

- Use the endorsed directories system property to specify the location of the libraries, by specifying this option on the java command line: -D"java.endorsed.dirs=<*JWSDP_HOME*>/jaxp/lib/endorsed"

Note: Since the JAXP *APIs* are already built into the Java 2 platform, they don't need to be specified at compile time. (In fact, the -D option is not even allowed at compile time, because endorsed standards are *required* to maintain consistent APIs.) However, when the JAXP factories instantiate an *implementation*, the endorsed directories mechanism is employed to make sure that the desired implementation is instantiated.

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 modify the `Echo` program to display a processing instruction contained in `slideSample02.xml`.

Note: The code discussed in this section is in `Echo05.java`. The file it operates on is `slideSample02.xml`, as described in Writing Processing Instructions (page 122). The output is in `Echo05-02.txt`. (The browsable versions are `slideSample02-xml.html` and `Echo05-02.html`.)

As you saw in Writing Processing Instructions (page 122), 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. The sample file `slideSample02.xml` contains a processing instruction for a mythical slide presentation program that queries 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 -->
```

To display that processing instruction, 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

The parser can generate one of three kinds of errors: fatal error, error, and warning. In this exercise, you'll how the parser handles a fatal error.

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.

Note: The XML file used in this exercise is `slideSampleBad1.xml`, as described in Introducing an Error (page 124). The output is in `Echo05-Bad1.txt`. (The browsable versions are `slideSampleBad1-xml.html` and `Echo05-Bad1.html`.)

When you created `slideSampleBad1.xml`, you deliberately created an XML file that was not well-formed. Run the Echo program on that file now. 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 this version of the program on `slideSampleBad1.xml` 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!

Displaying Special Characters and CDATA

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. In this section, you'll learn how the parser handles:

- Special Characters ("<", “&”, and so on)
- 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;
```

Earlier, you put an entity reference into your XML document by coding:

```
Market Size &lt; predicted
```

Note: The file containing this XML is slideSample03.xml, as described in Using an Entity Reference in an XML Document (page 126). The results of processing it are shown in Echo07-03.txt. (The browsable versions are slideSample03-xml.html and Echo07-03.html.)

When you run the Echo program on slideSample03.xml, 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, you use a CDATA section.

Note: The XML file used in this example is slideSample04.xml, as described in Handling Text with XML-Style Syntax (page 202). 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]]>. The file `slideSample04.xml` contains this 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. (Later in this tutorial, you will see how to use a `LexicalHandler` to find out whether or not you are processing a CDATA section.)

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.)

Parsing with a DTD

After the XML declaration, the document prolog can include a DTD, or reference an external DTD, or both. In this section, you'll see the effect of the DTD on the data that the parser delivers to your application.

DTD's Effect on the Nonvalidating Parser

In this section, you'll use the Echo program to see how the data appears to the SAX parser when the data file references a DTD.

Note: The XML file used in this section is `slideSample05.xml`, which references `slideshow1a.dtd`, as described in Parsing with a DTD (page 204). The output is shown in `Echo07-05.txt`. (The browsable versions are `slideshow1a-dtd.html`, `slideSample05-xml.html`, and `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`.

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 says that whitespace as ignorable.

Echoing Entity References

When you wrote `slideSample06.xml`, you defined entities for the product name. Now it's time to see how they're echoed when you process them with the SAX parser.

Note: The XML used here is contained in `slideSample06.xml`, which references `slideshow1b.dtd`, as described in Defining Attributes and Entities in the DTD (page 134). The output is shown in `Echo09-06.txt`. (The browsable versions are `slideSample06-xml.html`, `slideshow1b-dtd.html` and `Echo09-06.html`.)

When you run the Echo program on `slideSample06.xml`, 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.

Echoing the External Entity

In `slideSample07.xml`, you defined an external entity to reference a copyright file.

Note: The XML used 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`.)

When you run the Echo program on that 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.

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)`. Sun's 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 Validating with XML Schema (page 292). 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 Validating with XML Schema (page 292). 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 295).

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 is produced using `slideSample07.xml`, as described in Referencing Binary Entities (page 140). The output 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.

Parsing a Parameterized DTD

This section uses the Echo program to see what happens when you reference `xhtml.dtd` in `slideshow2.dtd`. It also covers the kinds of warnings that are generated by the SAX parser when a DTD is present.

Note: The XML file used here is `slideSample08.xml`, which references `slideshow2.dtd`. The output is contained in `Echo10-08.txt`. (The browsable versions are `slideSample08-xml.html`, `slideshow2-dtd.html`, and `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.

The problem is 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.

The `slideSample09.xml` version solves the problem by changing the name of the slide title. Run the Echo program on that 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 change in that version of the file had the effect of putting the DTD's `title` element into a slideshow "namespace" that you artificially constructed by hyphenating the name, so the `title` element in the "slideshow namespace" (`slide-title`, really) was no longer in conflict with the `title` element in `xhtml.dtd`.

Note: As mentioned in Using Namespaces (page 147), namespaces let you accomplish the same goal without having to rename any elements.

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 227).

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 `<` and `&`. 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 reference 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 `slideSample09.xml`. The results of processing are in `Echo12-09.txt` (The browsable versions are `slideSample09-xml.html` and `Echo12-09.html`.)

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=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 Choosing your Parser Implementation (page 210) 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 mechanism: `http://www.ascc.net/xml/resource/schematron/schematron.html`

8

Document Object Model

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 `<INSTALL>/jwstutorial13/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 107) 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 Versus 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 (page 103) 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 103), 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 290) and Obtaining Node Content (page 291). 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 301), 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 284).

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 193) 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, Validating with XML Schema (page 293). 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 193). For a more complete background on validation issues, refer to Using the Validating Parser (page 210).

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 8–1 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 8–1 shows the tree you see when you run the DomEcho program on the first XML file you created in the DOM tutorial.

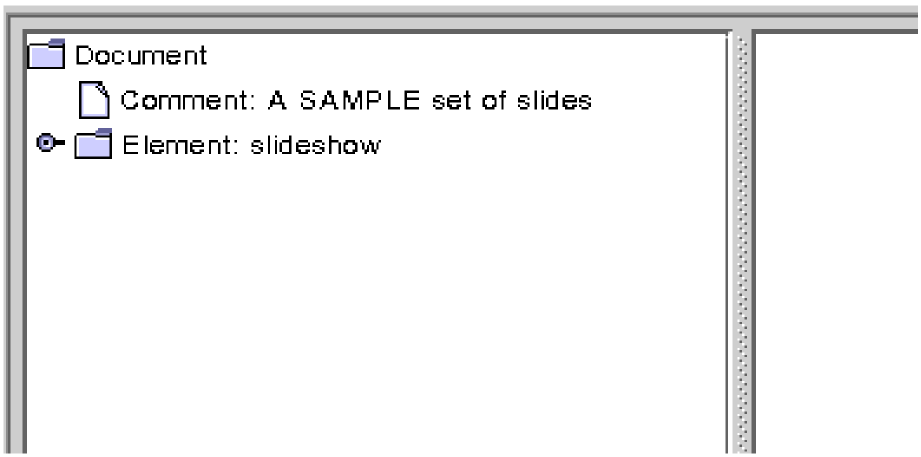


Figure 8–1 Document, Comment, and Element Nodes Displayed

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 one node of `Document` type for the whole tree, the root node. The `Comment` node displays the `value` attribute, while the `Element` node displays the element `name`, "slideshow".

Compare Figure 8–1 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 8–2.

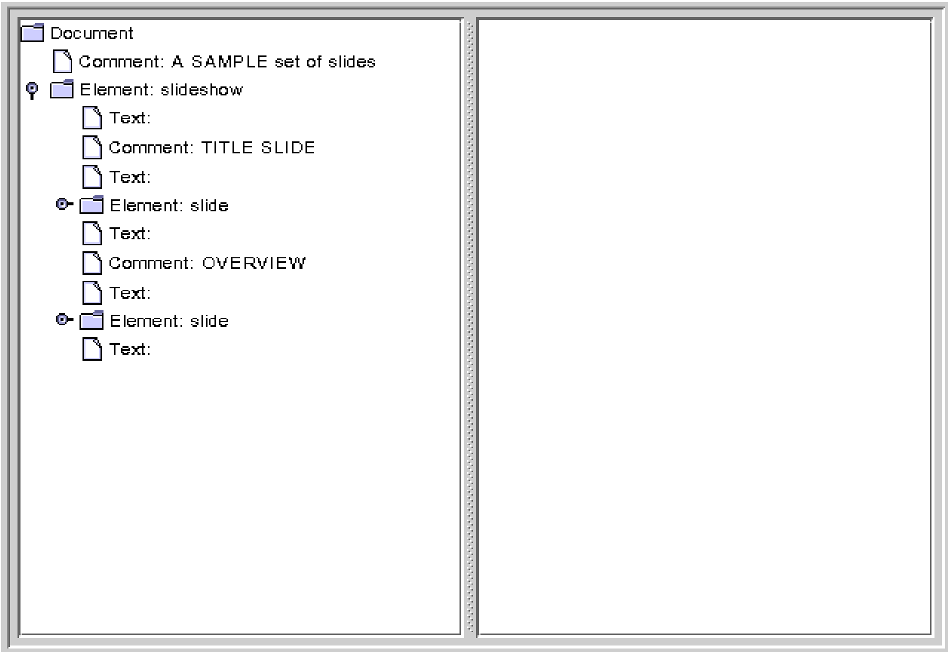


Figure 8–2 Element Node Expanded, No Attribute Nodes Showing

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 8–3 shows the result of running the `DomEcho` application on `slideSample10.xml`, which includes a `DOCTYPE` entry that identifies the document's DTD.

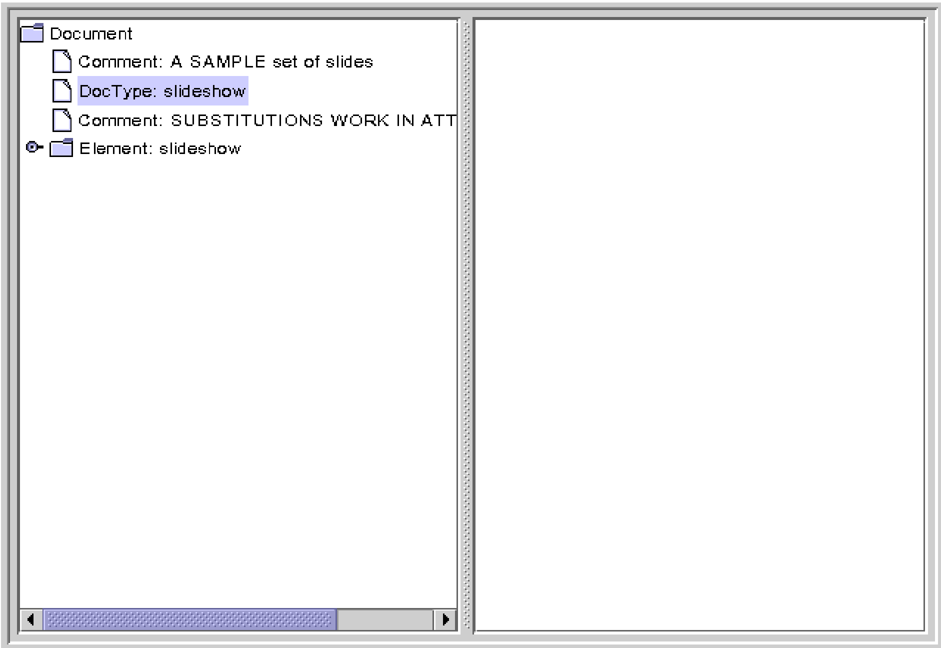


Figure 8–3 DocType Node Displayed

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 8–4.

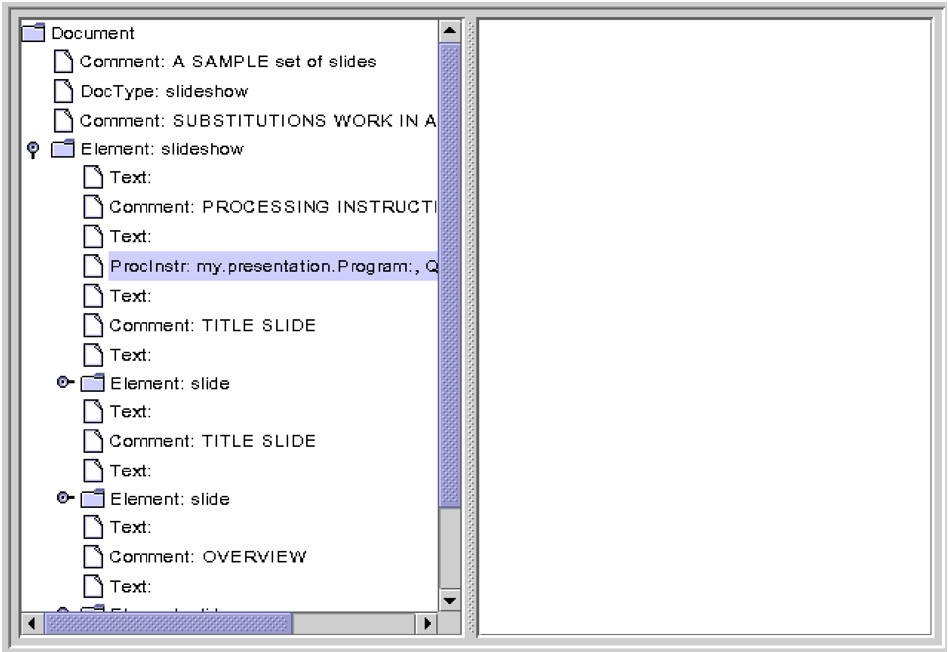


Figure 8–4 Processing Instruction Node Displayed

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 8–5.

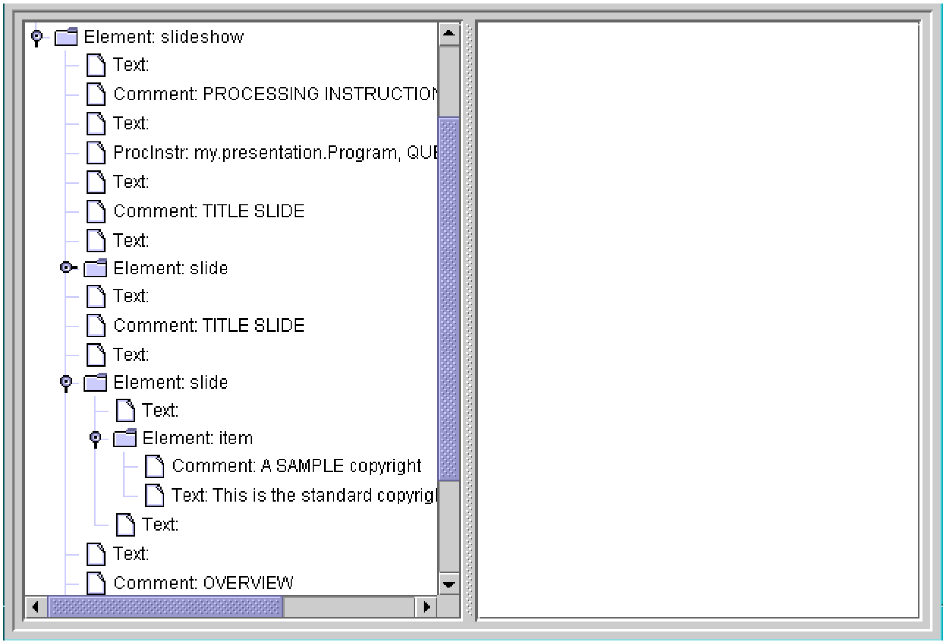


Figure 8–5 JAXP 1.2 DOM — Item Text Returned from an Entity Reference

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 8–6.

```
public static void main(String argv[])
{
  ...
  DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
  factory.setExpandEntityReferences(false);
  ...
```

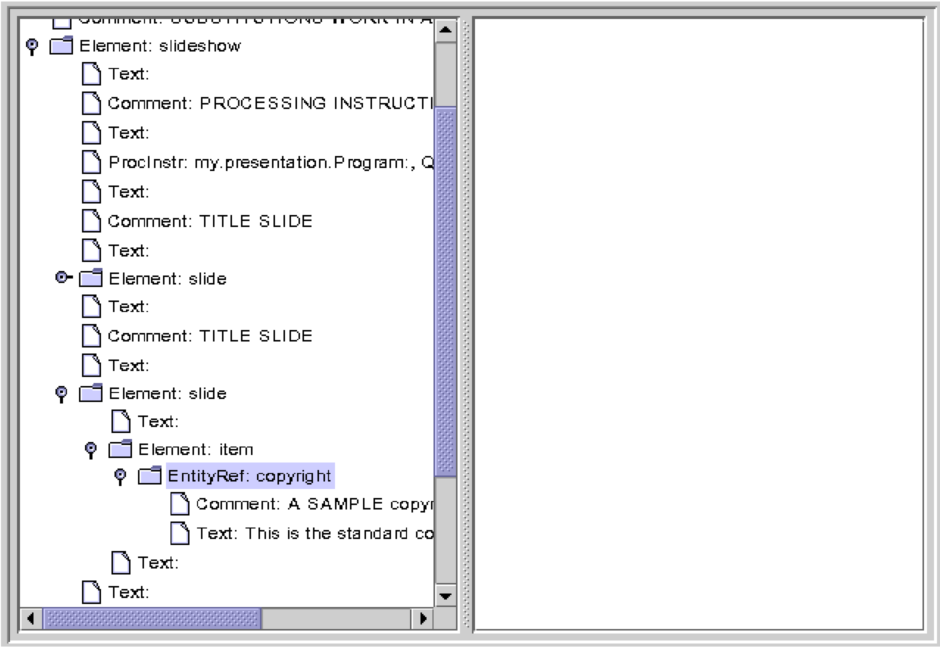


Figure 8–6 JAXP 1.1 in 1.4 Platform— Entity Reference Node Displayed

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 8–7.

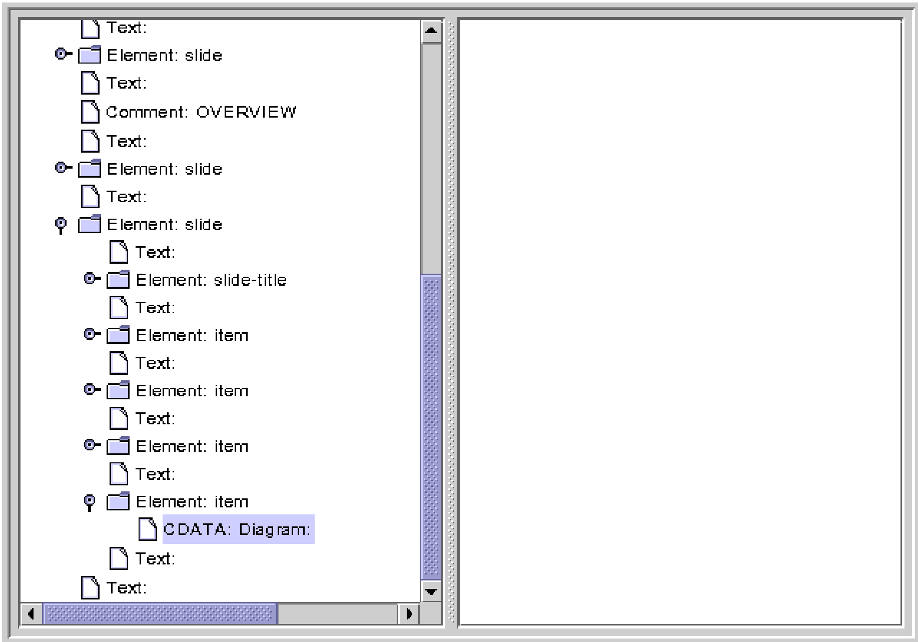


Figure 8–7 CDATA Node Displayed

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 whitespace that is not a significant part of element content.

The default values for all of these properties is `false`, which preserves all the lexical information necessary to reconstruct the incoming document in its original form. Setting them to true lets you 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 8–2 summarizes the effect of the settings.

Table 8–2 Configuring `DocumentBuilderFactory`

API	Preserve Lexical Info	Focus on Content
`setCoalescing()`	false	true
`setExpandEntityReferences()`	false	true
`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 284), 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 8–8. The only nodes remaining in the tree are the high-level "structure" nodes.

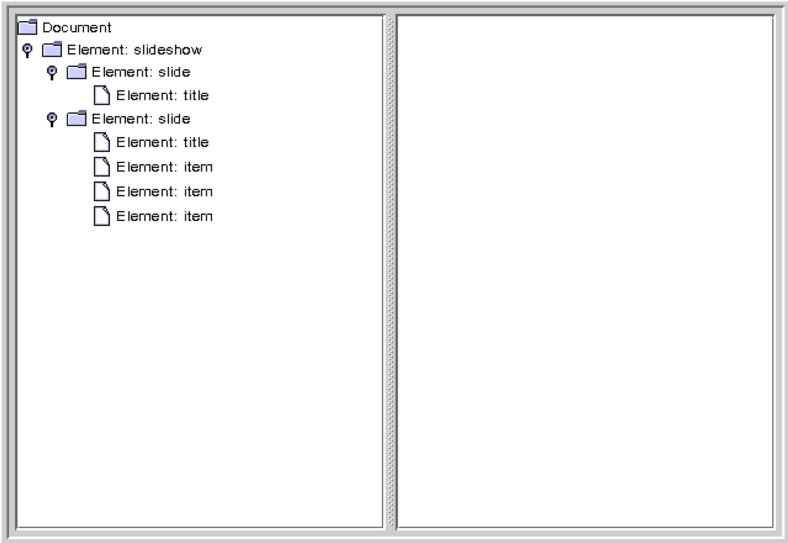


Figure 8–8 Tree View with a Collapsed Hierarchy

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 Choosing your Parser Implementation and Using the DTDHandler and EntityResolver (page 226), 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 first line of 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("\n");
            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 8–9. Expanding the hierarchy shows that the `JTree` now includes identifying text for a node whenever possible.

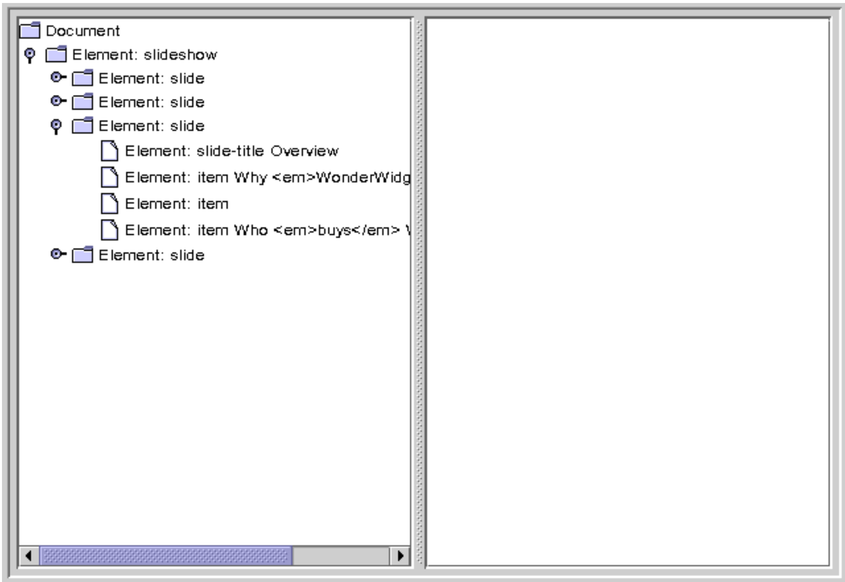


Figure 8–9 Collapsed Hierarchy Showing Text in Nodes

Selecting an item that includes XHTML subelements produces a display like that shown in Figure 8–10:

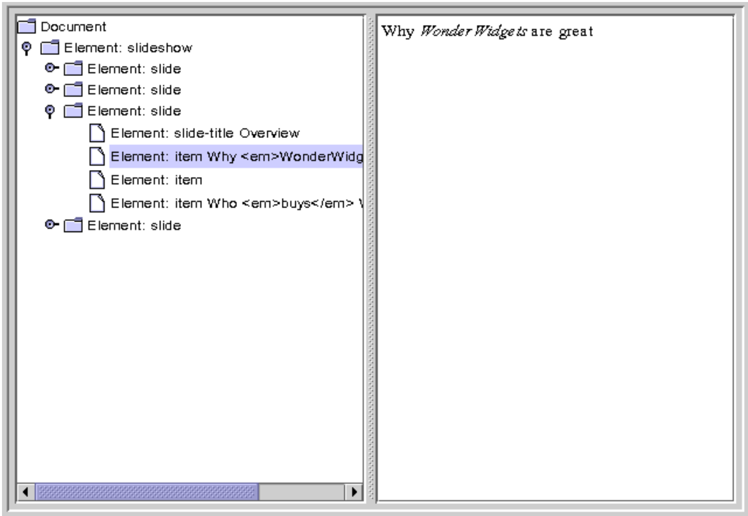


Figure 8–10 Node with `<em>` Tag Selected

Selecting a node that contains an entity reference causes the entity text to be included, as shown in Figure 8–11:

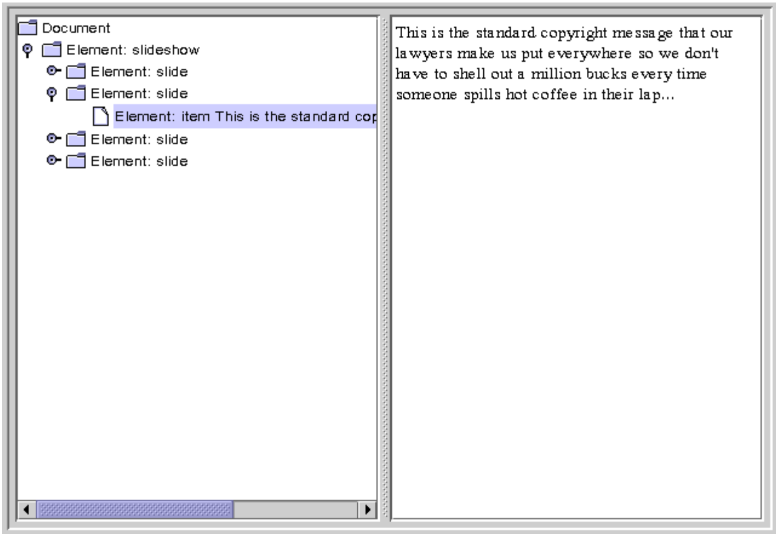


Figure 8–11 Node with Entity Reference Selected

Finally, selecting a node that includes a `CDATA` section produces results like those shown in Figure 8–12:

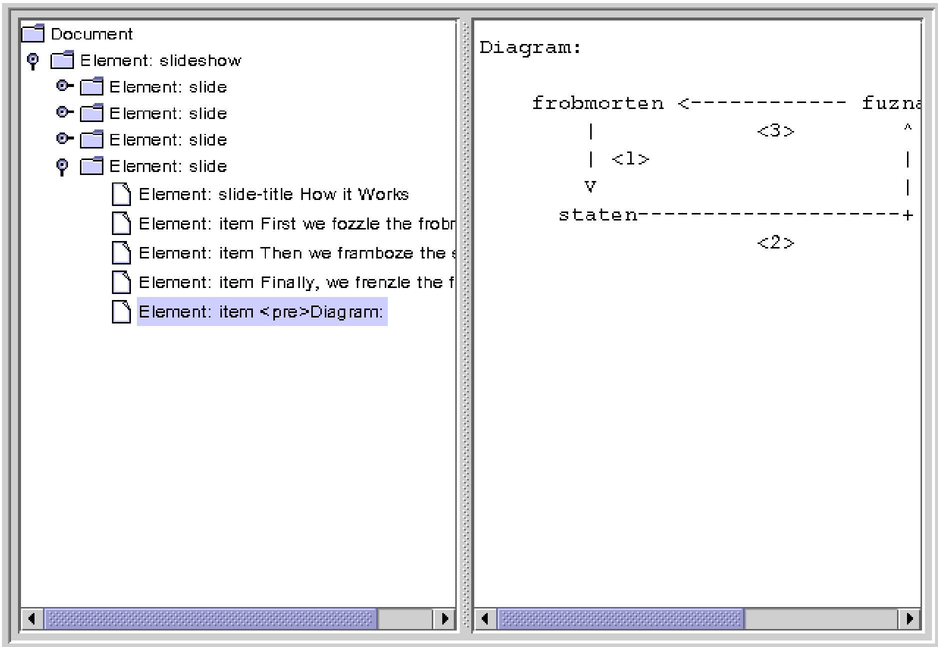


Figure 8–12 Node with `CDATA` Component Selected

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 8–13:

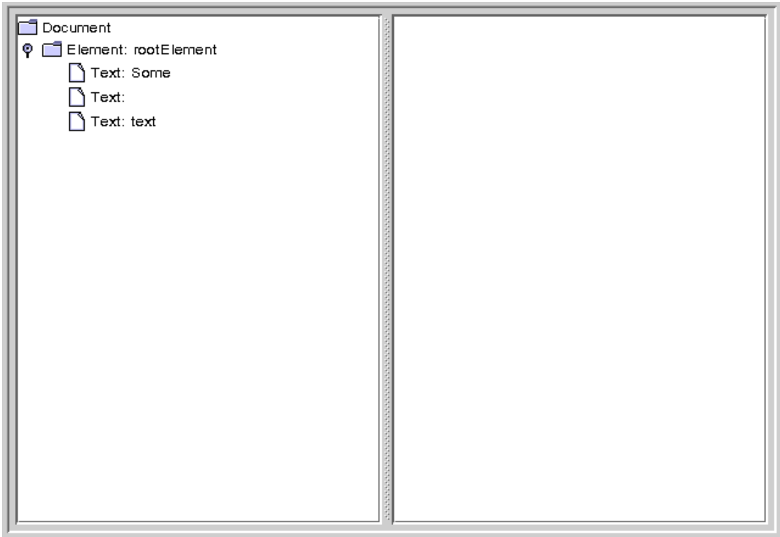


Figure 8–13 Element Node and Text Nodes Created

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 8–14:

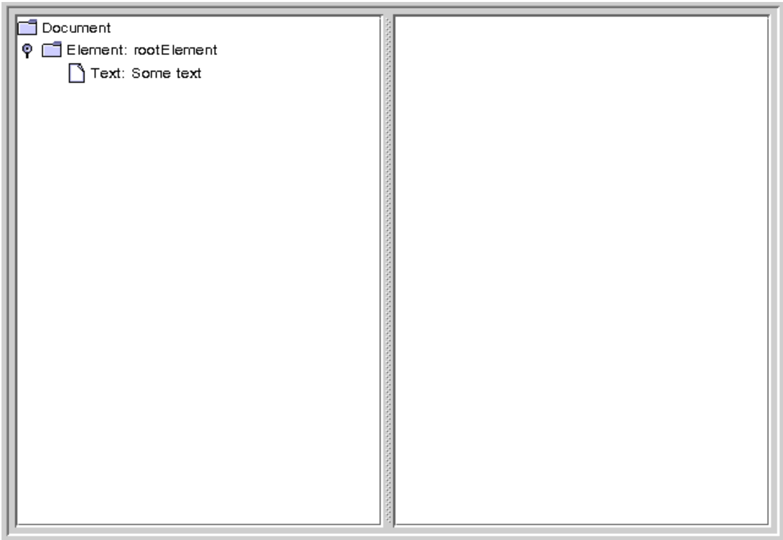


Figure 8–14 Text Nodes Merged After Normalization

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 231) in When to Use DOM.

Note, too, that you can use APIs described in Summary of Lexical Controls (page 266) 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 231) in When to Use DOM.

Again, you can simplify this code by using the APIs described in Summary of Lexical Controls (page 266) 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 the `setAttributeNode` method to add it.

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!

Validating with XML Schema

You're now 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 semblance 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.)

You 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 mechanism: `http://www.ascc.net/xml/resource/schematron/schematron.html`

9

XML Stylesheet Language for Transformations

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 an interpreting implementation of XSLT, called Xalan.

In this chapter, you'll learn how to use Xalan. 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 `<INSTALL>/jwstutorial13/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 321), 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.

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.

`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.

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 335). 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 314). (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 `<` or a CDATA section was used to include it. (Similarly, the text will contain an `&` character, regardless of whether it was delivered using `&` 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 9–1 lists the wildcards that can be used in XPath expressions to broaden the scope of the pattern matching.

Table 9–1 XPath Wildcards

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 names 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 9–2 lists the operators that can be used in an Xpath expression

Table 9–2 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 < 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 < or enclosed in a CDATA section. Similarly, occurrences of “&” will need to be converted to &.

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.

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.

Note: You can specify a variety of output properties for transformer objects, as defined in the W3C specification at `http://www.w3.org/TR/xslt#output`. For example, to get indented output, you can invoke:

```
transformer.setOutputProperty("indent", "yes");
```

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 181) 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 239).

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 9–1 shows the address book entry that was created.

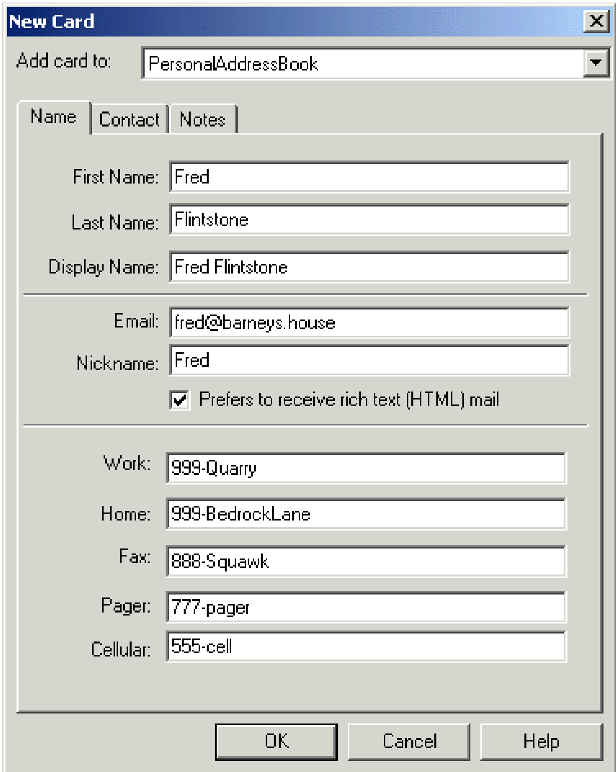


Figure 9–1 Address Book Entry

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 112)) 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 303), 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 303).

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 304).)

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 TransformationAppStylizer
{
  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 `StreamSource` 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 304).)

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 `<` entity reference, it will appear as the `<` character in the generated text. When XML is generated, on the other hand, the `<` entity reference in the stylesheet would be unchanged, so it would appear as `<` in the generated text.

Note: If you actually want `<` to be generated as part of the HTML output, you'll need to encode it as `<`—that sequence becomes `<` on output, because only the `&` 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 What Else Can XSLT Do? (page 357) 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 with Xalan

To run a transform from the command line, you initiate a Xalan `Process` using the following command:

```
java org.apache.xalan.xslt.Process
  -IN article3.xml -XSL article3.xsl
```

Note: Remember to use the endorsed directories mechanism to access the Xalan libraries, as described in Compiling and Running the Program (page 181).

With this command, the output goes to `System.out`. The -OUT option can also be used to output to a file.

The `Process` command allows for a variety of other options, as well. For details, see `http://xml.apache.org/xalan-j/commandline.html`.

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.
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 9–2 should help you understand what's happening when it executes.

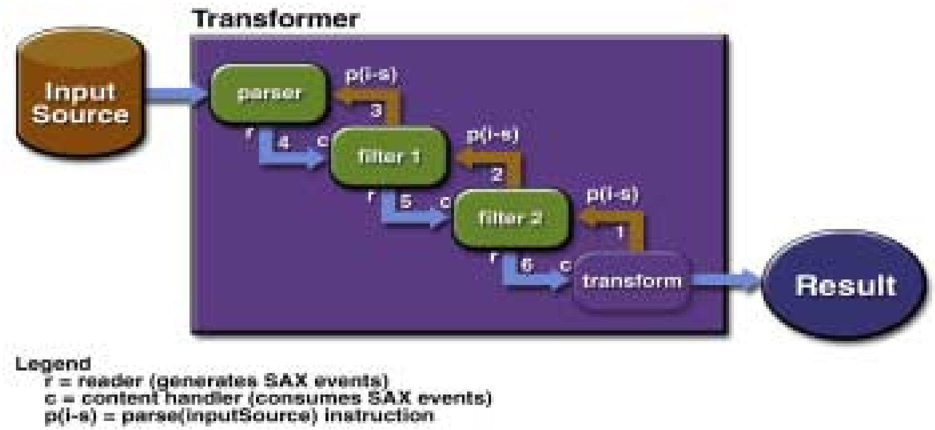


Figure 9–2 Operation of Chained Filters

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 DocBook 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 can 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/`
- Output properties that can be programmatically specified on transformer objects: `http://www.w3.org/TR/xslt#output`.
- Using Xalan from the command line: `http://xml.apache.org/xalan-j/commandline.html`

10

Binding XML Schema to Java Classes with JAXB

THE Java™ Architecture for XML Binding (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. As part of this process, JAXB provides methods for unmarshalling XML instance documents into Java content trees, and then marshalling Java content trees back into XML instance documents.

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 11, 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 10–1 shows the components that make up a JAXB implementation.

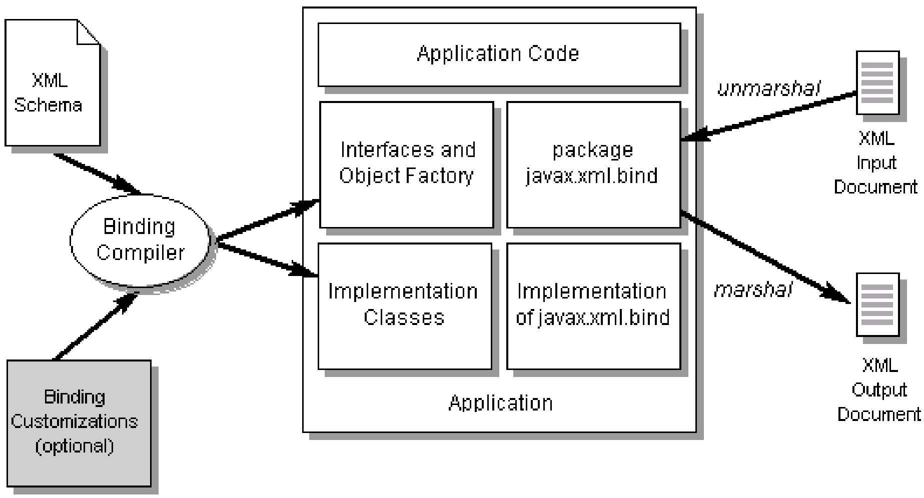


Figure 10–1 JAXB Architectural Overview

As shown in Figure 10–1, a JAXB implementation comprises the following eight core components.

Table 10–1 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 Customizations	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 customizations* 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 customizations 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.
Implementation of `javax.xml.bind`	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 10–1 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 10–2 shows what occurs during the JAXB binding process.

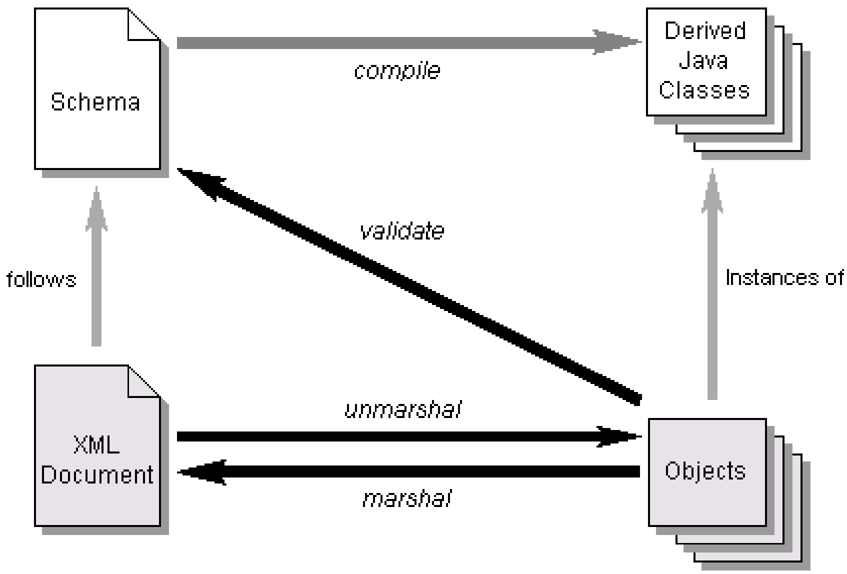


Figure 10–2 Steps in the JAXB Binding Process

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 Create Marshal Example (page 415) 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 10–2 shows the properties that the `Marshaller` class supports.

Table 10–2 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.
`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 11 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 10–3 lists the mapping of XML data types to Java data types in JAXB.

Table 10–3 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`

Table 10–3 JAXB Mapping of XML Schema Built-in Data Types (Continued)

XML Schema Type	Java Data Type
`xsd:time`	`java.util.Calendar`
`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 11.

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 10–4 lists the four scopes for custom bindings.

Table 10–4 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` subdirectory of the Java WSDP. This directory contains sample applications, a JAXB binding compiler (`xjc`), and implementations of the runtime binding framework APIs contained in the `javax.xml.bind` package. For instructions on using the JAXB, see Chapter 11.

11

Using JAXB

THIS chapter provides instructions for using several of the sample Java applications that were included in the Java WSDP. These examples demonstrate and build 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 examples 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 424) for instructions on using five additional examples 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 examples 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.

Description

This chapter describes ten examples; the basic examples (Unmarshal Read, Modify Marshal, Create Marshal, Unmarshal Validate, Validate-On-Demand) demonstrate basic JAXB concepts like ummarshalling, marshalling, and validating XML content, while the customize examples (Customize Inline, Datatype Converter, External Customize, Fix Collides, Bind Choice) demonstrate various ways of customizing the binding of XML schemas to Java objects. Each of the examples in this chapter is based on a *Purchase Order* scenario. With the exception of the Bind Choice and the Fix Collides examples, each uses an XML document, `po.xml`, written against an XML schema, `po.xsd`.

Table 11–1 Sample JAXB Application Descriptions

Example Name	Description
Unmarshal Read Example	Demonstrates how to unmarshal an XML document into a Java content tree and access the data contained within it.
Modify Marshal Example	Demonstrates how to modify a Java content tree.
Create Marshal Example	Demonstrates how to use the *`ObjectFactory`* class to create a Java content tree from scratch and then marshal it to XML data.
Unmarshal Validate Example	Demonstrates how to enable validation during unmarshalling.
Validate-On-Demand Example	Demonstrates how to validate a Java content tree at runtime.
Customize Inline Example	Demonstrates how to customize the default JAXB bindings by means of inline annotations in an XML schema.

Table 11–1 Sample JAXB Application Descriptions

Example Name	Description
Datatype Converter Example	Similar to the Customize Inline example, this example illustrates alternate, more terse bindings of XML `simpleType` definitions to Java datatypes.
External Customize Example	Illustrates how to use an external binding declarations file to pass binding customizations for a read-only schema to the JAXB binding compiler.
Fix Collides Example	Illustrates how to use customizations to resolve name conflicts reported by the JAXB binding compiler. 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.
Bind Choice Example	Illustrates how to bind a `choice` model group to a Java interface.

Note: These examples are all located in the `$JWSDP_HOME/jaxb/samples` directory.

Each example 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 the Customize Inline and Datatype Converter examples, this file contains inline binding customizations. Note that the Bind Choice and Fix Collides examples use `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 example, with minor content differences to highlight different JAXB concepts. Note that the Bind Choice and Fix Collides examples use `example.xml` rather than `po.xml`.
- `Main.java` is the main Java class for each example.
- `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 examples.

- `MyDatatypeConverter.java` in the `inline-customize` example is a Java class used to provide custom datatype conversions.
- `binding.xjb` in the External Customize, Bind Choice, and Fix Collides examples is an external binding declarations file that is passed to the JAXB binding compiler to customize the default JAXB bindings.
- `example.xsd` in the Fix Collides example is a short schema file that contains deliberate naming conflicts, to show how to resolve such conflicts with custom JAXB bindings.

Using the Examples

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 examples, you have a choice of performing these steps by hand, or by using `Ant` with the `build.xml` project file included in each example directory.

Note: It is recommended that you familiarize yourself with the manual process for at least the Unmarshal Read example. The manual process is similar for each of the examples.

Configuring and Running the Examples Manually

This section describes how to configure and run the Unmarshal Read example. The instructions for the other examples are essentially the same; just change the `<INSTALL>/jwstutorial13/examples/jaxb/unmarshal-read` directory to the directory for the example you want to use.

Solaris/Linux

1. Set the following environment variables:

   ```
   export JAVA_HOME=<your J2SE installation directory>
   export JWSDP_HOME=<your JWSDP 1.3 installation directory>
   ```

2. Change to the desired example directory.

   For example, to run the Unmarshal Read example:

   ```
   cd <INSTALL>/jwstutorial13/examples/jaxb/unmarshal-read
   ```

   (*<INSTALL>* is the directory where you installed the tutorial bundle.)

3. Use the `xjc.sh` command to generate JAXB Java classes from the source XML schema.

   ```
   $JWSDP_HOME/jaxb/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 398) for a complete list of JAXB binding compiler options.

4. 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"
```

5. Compile the generated JAXB Java classes.

```
$JAVA_HOME/bin/javac Main.java primer/po/*.java primer/
po/impl/*.java
```

6. 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 the following environment variable:

   ```
   set JAVA_HOME=<your J2SE installation directory>
   set JWSDP_HOME=<your JWSDP 1.3 installation directory>
   ```

2. Change to the desired example directory.

   For example, to run the Unmarshal Read example:

   ```
   cd <INSTALL>\jwstutorial13\examples\jaxb\unmarshal-read
   ```

   (<*INSTALL*> is the directory where you installed the tutorial bundle.)

3. Use the `xjc.bat` command to generate JAXB Java classes from the source XML schema.

   ```
   %JWSDP_HOME%\jaxb\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 398) for a complete list of JAXB binding compiler options.

4. 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"
   ```

5. Compile the schema-derived JAXB Java classes.

   ```
   %JAVA_HOME%\bin\javac Main.java primer\po\*.java
    primer\po\impl\*.java
   ```

6. 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 400). The methods used for building and processing the Java content tree in each of the basic examples are analyzed in Basic Examples (page 411).

Configuring and Running the Samples With Ant

The `build.xml` file included in each example directory is an Ant project file that, when run, automatically performs all the steps listed in Configuring and Running the Examples Manually (page 394). 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 example.

As mentioned previously, it is recommended that you familiarize yourself with the manual steps for performing these tasks for at least the first example.

Solaris/Linux

1. Set the following environment variables:
   ```
   export JAVA_HOME=<your J2SE installation directory>
   export JWSDP_HOME=<your JWSDP installation directory>
   ```
2. Change to the desired example directory.

   For example, to run the Unmarshal Read example:
   ```
   cd <INSTALL>/jwstutorial13/examples/jaxb/unmarshal-read
   ```
   (*<INSTALL>* is the directory where you installed the tutorial bundle.)
3. Run Ant:
   ```
   $JWSDP_HOME/apache-ant/bin/ant -emacs
   ```
4. Repeat these steps for each example.

Windows NT/2000/XP

1. Set the following environment variables:
   ```
   set JAVA_HOME=<your J2SE installation directory>
   set JWSDP_HOME=<your JWSDP installation directory>
   ```
2. Change to the desired example directory.

   For example, to run the Unmarshal Read example:
   ```
   cd <INSTALL>\jwstutorial13\examples\jaxb\unmarshal-read
   ```

(<*INSTALL*> is the directory where you installed the tutorial bundle.)

3. Run Ant:

   `%JWSDP_HOME%\apache-ant\bin\ant -emacs`

4. Repeat these steps for each example.

The schema-derived JAXB classes and how they are bound to the source schema is described in About the Schema-to-Java Bindings (page 400). The methods used for building and processing the Java content tree are described in Basic Examples (page 411).

JAXB Compiler Options

The JAXB schema binding compiler is located in the `<JWSDP_HOME>/jaxb/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 11–2.

Table 11–2 `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.

Table 11–2 `xjc` Command-Line Options (Continued)

Option or Argument	Description
`-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.

`-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.
`-catalog <file>`	Specify catalog files to resolve external entity references.Supports TR9401, XCatalog, and OASIS XML Catalog format.

Table 11–2 `xjc` Command-Line Options (Continued)

Option or Argument	Description
`-readOnly`	Generated source files will be marked read-only. By default, `xjc` does not write-protect the schema-derived source files it generates.
`-use-runtime` <*pkg*>	Suppress the generation of the `impl.runtime` package and refer to another existing runtime in the specified package. This option is useful when you are compiling multiple independent schemas. Because the generated impl.runtime packages are identical, except for their package declarations, you can reduce the size of your generated codebase by telling the compiler to reuse an existing `impl.runtime` package.
`-xmlschema`	Treat input schemas as W3C XML Schema (default). If you do not specify this switch, your input schemas will be treated as W3C XML Schema.
`-relaxng`	Treat input schemas as RELAX NG (experimental, unsupported). Support for RELAX NG schemas is provided as a JAXB Vendor Extension.
`-dtd`	Treat input schemas as XML DTD (experimental, unsupported). Support for RELAX NG schemas is provided as a JAXB Vendor Extension.
`-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 basic examples (Unmarshal Read, Modify Marshal, Create Marshal, Unmarshal Validate, Validate-On-Demand), the JAXB binding compiler generates a Java package named `primer.po` containing eleven classes, making a total of twelve classes in each of the basic examples:

Table 11–3 Schema-Derived JAXB Classes in the Basic Examples

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.DefaultJAXBContextImpl`; 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/PurchaseOrderTypeImpl.java`	Implementation of `PurchaseOrderType.java`
`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 the basic examples are described below.

Table 11–4 Schema-to-Java Bindings for the Basic Examples

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>`
	`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:complexType name="Items">`
`  <xsd:sequence>`
`    <xsd:element name="item" minOccurs="1" maxOc-`
`curs="unbounded">`
	`Items.java`

Table 11–4 Schema-to-Java Bindings for the Basic Examples (Continued)

XML Schema	JAXB Binding
<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.ItemType`
</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>	

Schema-Derived JAXB Classes

The code for the individual classes generated by the JAXB binding compiler for the basic examples 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 Examples

This section describes five basic examples (Unmarshal Read, Modify Marshal, Create Marshal, Unmarshal Validate, Validate-On-Demand) 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

Unmarshal Read Example

The purpose of the Unmarshal Read example is to demonstrate how to unmarshal an XML document into a Java content tree and access the data contained within it.

1. The `<INSTALL>/jwstutorial13/examples/jaxb/unmarshal-read/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 example 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"
```

Modify Marshal Example

The purpose of the Modify Marshal example is to demonstrate how to modify a Java content tree.

1. The `<INSTALL>/jwstutorial13/examples/jaxb/modify-marshal/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 example 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>
```

Create Marshal Example

The Create Marshal example demonstrates how to use the `ObjectFactory` class to create a Java content tree from scratch and then marshal it to XML data.

1. The `<INSTALL>/jwstutorial13/examples/jaxb/create-marshal/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 example 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>
```

Unmarshal Validate Example

The Unmarshal Validate example demonstrates 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 377).

1. The `<INSTALL>/jwstutorial13/examples/jaxb/unmarshal-validate/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 example produces the following output:

```
DefaultValidationEventHandler: [ERROR]: "-1" does not satisfy
the "positiveInteger" type
Caught UnmarshalException
```

Validate-On-Demand Example

The Validate-On-Demand example demonstrates 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 377).

1. The `<INSTALL>/jwstutorial13/examples/jaxb/ondemand-validate/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 example 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 examples that build on the concepts demonstrated in the basic examples.

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 Examples (page 411), which focus on the Java code in the respective `Main.java` class files, the examples 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 398).

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 11–4.

Figure 11–1 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 11–1 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 436) 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 434) 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 `propertyCollectionType`, which is the collection type for the property. `propertyCollectionType` 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 `generateIsSetMethod`. The value can be either `true`, `false`, `1`, or `0`.
- `enableFailFastCheck` defines the customization value `enableFailFastCheck`. 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 443) 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 431).

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 `<typesafeEnumMember>` 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 the Customize Inline example, 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 429).

Customize Inline Example

The Customize Inline example illustrates some basic customizations made by means of inline annotations to an XML schema named po.xsd. In addition, this example 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 the Customize Inline example, 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

The customized schema used in the Customize Inline example is in the file `<INSTALL>/jwstutorial13/examples/jaxb/inline-customize/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 element 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 <b>Purchase Order</b> consists of addresses and items."` The `<` 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 `<`. 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 `<INSTALL>/jwstutorial13/examples/jaxb/inline-customize/MyDatatypeConverter` class, shown below, provides 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 the Customize Inline example, 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.

Datatype Converter Example

The Datatype Converter example is very similar to the Customize Inline example. As with the Customize Inline example, the customizations in the Datatype Converter example are made by using inline binding declarations in the XML schema for the application, `po.xsd`.

The global, schema, and package, and most of the class customizations for the Customize Inline and Datatype Converter examples are identical. Where the Datatype Converter example differs from the Customize Inline example 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, the Datatype Converter example 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>
```

External Customize Example

The External Customize example is identical to the Datatype Converter example, except that the binding declarations in the External Customize example are made by means of an external binding declarations file rather than inline in the source XML schema.

The binding customization file used in the External Customize example is `<INSTALL>/jwstutorial13/examples/jaxb/external-customize/binding.xjb`.

This section compares the customization declarations in `bindings.xjb` with the analogous declarations used in the XML schema, `po.xsd`, in the Datatype Converter example. 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 445), 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 445) 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 the Datatype Converter example. 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 the Datatype Converter example 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 the Datatype Converter example 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.

Fix Collides Example

The Fix Collides example 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 example, it is recommended that you run the `ant fail` command in the `<INSTALL>/jwstutorial13/examples/jaxb/fix-collides` directory to display the error output generated by the `xjc` compiler. The XML schema for the Fix Collides, `example.xsd`, contains deliberate name conflicts.

Like the External Customize example, the Fix Collides example 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

The example.xsd Schema

The XML schema, `<INSTALL>/jwstutorial13/examples/jaxb/fix-collides/example.xsd`, used in the Fix Collides example 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:element name="zip" type="xs:integer"/>
    </xs:sequence>
    <xs:attribute name="zip" type="xs:string"/>
  </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 `<INSTALL>/jwstutorial13/examples/jaxb/fix-collides` directory:

```
[echo] Compiling the schema w/o external binding file (name
collision errors expected)...
[xjc] Compiling file:/C:/jwstutorial13/examples/jaxb/fix-
collides/ example.xsd
[xjc] [ERROR] Attempt to create a property having the same name
as the reserved word "Class".
[xjc]   line 14 of example.xsd
[xjc] [ERROR] A property with the same name "Zip" is generated
from more than one schema component.
[xjc]   line 17 of example.xsd
[xjc] [ERROR] (Relevant to above error) another one is generated
from this schema component.
[xjc]   line 15 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 18 of example.xsd
```

The binding.xjb Declarations File

The `<INSTALL>/jwstutorial13/examples/jaxb/fix-collides/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 `<INSTALL>/jwstutorial13/examples/jaxb/fix-collides` 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.

Bind Choice Example

The Bind Choice example shows how to bind a `choice` model group to a Java interface. Like the External Customize and Fix Collides examples, the Bind Choice example uses an external binding declarations file, `binding.xjb`, to define the JAXB binding customization.

The schema declarations in `<INSTALL>/jwstutorial13/examples/jaxb/bind-choice/example.xsd` that will be globally changed 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="FooBar">
  <xs:complexType>
    <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:group ref="ModelGroupChoice"/>
    </xs:sequence>
    <xs:attribute name="zip" type="xs:string"/>
  </xs:complexType>
</xs:element>

  <xs:group name="ModelGroupChoice">
    <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>
  </xs:group>
</xs:schema>
```

Customizing a choice Model Group

The `<INSTALL>/jwstutorial13/examples/jaxb/bind-choice/binding.xjb` binding declarations file demonstrates one way to override the default derived names for `choice` model groups in `example.xsd` by means of a `<jxb:globalBindings>` declaration:

```
<jxb:bindings schemaLocation="example.xsd" node="/xs:schema">
  <jxb:globalBindings bindingStyle="modelGroupBinding"/>
    <jxb:schemaBindings/>
      <jxb:package name="example"/>
    </jxb:schemaBindings>
  </jxb:bindings
</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:group name="ModelGroupChoice">
  <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>
</xs:group>
```

the globalBindings customization shown above causes JAXB to generate the following Java class:

```
/**
 * Java content class for model group.
 */
 public interface ModelGroupChoice {
       int getValue();
       void setValue(int value);
       boolean isSetValue();

       java.lang.String getComment();
       void setComment(java.lang.String value);
       boolean isSetComment();

       boolean isBool();
       void setBool(boolean value);
       boolean isSetBool();

       Object getContent();
       boolean isSetContent();
       void unSetContent();
   }
```

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.

Additionally, the generated Java interface FooBarType, representing the anonymous type definition for element FooBar, contains a nested interface for the choice model group containing phoneNumber and speedDial.

12

Building Web Services With JAX-RPC

JAX-RPC stands for Java API for XML-based RPC. It's an API for building Web services and clients that use 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 the envelope structure, encoding rules, and convention for representing remote procedure calls and responses. These calls and responses are transmitted as SOAP messages (XML files) over HTTP.

Although SOAP messages are complex, the JAX-RPC 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, the developer does not generate or parse SOAP messages. It is the JAX-RPC runtime system that converts the API calls and responses to and from SOAP messages.

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.

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) 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.net.URI

java.util.Calendar
java.util.Date
```

This release of JAX-RPC also supports several implementation classes of the `java.util.Collection` interface. See Table 12–1.

Table 12–1 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.

Value Types

A *value type* is a class whose state may be passed between a client and remote service as a method parameter or return value. For example, in an application for a university library, a client might call a remote procedure with a value type parameter named `Book`, a class that contains the fields `Title`, `Author`, and `Publisher`.

To be supported by JAX-RPC, a value type 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 value type may contain public, private, or protected fields. The field of a value type 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 Service Implementation (page 1004).

Setting the Port

Several files in the JAX-RPC examples depend on the port that you specified when you installed the Web Services Developer Pack. The tutorial examples assume that the server runs on the default port, 8080. If you have changed the port, you must update the port number in the following files before building and running the examples:

- `<INSTALL>/jwstutorial13/examples/jaxrpc/advanced/dii/conf/config-client.xml`
- `<INSTALL>/jwstutorial13/examples/jaxrpc/advanced/dii/src/client/hello/DIIClient.properties`
- `<INSTALL>/jwstutorial13/examples/jaxrpc/advanced/dynamic/conf/config-client.xml`
- `<INSTALL>/jwstutorial13/examples/jaxrpc/advanced/dynamic/src/client/hello/ProxyClient.properties`
- `<INSTALL>/jwstutorial13/examples/jaxrpc/advanced/handler/conf/config.xml`
- `<INSTALL>/jwstutorial13/examples/jaxrpc/advanced/stubs/conf/config-client.xml`
- `<INSTALL>/jwstutorial13/examples/jaxrpc/advanced/stubs/conf/config-client-oneway.xml`
- `<INSTALL>/jwstutorial13/examples/jaxrpc/basicauthclient/SecureHello.wsdl`
- `<INSTALL>/jwstutorial13/examples/jaxrpc/dynamicproxy/config-wsdl.xml`
- `<INSTALL>/jwstutorial13/examples/jaxrpc/mutualauthclient/SecureHello.wsdl`
- `<INSTALL>/jwstutorial13/examples/jaxrpc/staticstub/config-wsdl.xml`
- `<INSTALL>/jwstutorial13/examples/jaxrpc/webclient/config-wsdl.xml`
- `<INSTALL>/jwstutorial13/examples/jaxrpc/webclient/web/response.jsp`

Creating a Web Service with JAX-RPC

This section shows how to build and deploy a simple Web service called `MyHelloService`. A later section, Creating Web Service Clients with JAX-RPC (page 465), provides examples of JAX-RPC clients that access this service. The source code required by `MyHelloService` is in `<INSTALL>/jwstutorial13/examples/jaxrpc/helloservice/`.

These are the basic steps for creating the service:

1. Code the service endpoint interface and implementation class.
2. Build, generate, and package the files required by the service.
3. Deploy the WAR filethat contains the service.

The sections that follow cover these steps in greater detail.

Note: Before proceeding, you should try out the introductory examples in Chapter 3. Make sure that you've followed the instructions in Setting Up (page 49).

Coding the Service Endpoint Interface and Implementation Class

A service endpoint interface declares the methods that a remote client may invoke on the service. In this example, the interface declares a single method named `sayHello`.

A service endpoint 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 456).

In this example, the service endpoint interface is `HelloIF.java`:

```
package helloservice;

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 the class that implements the interface. In this example, the implementation class is called `HelloImpl`:

```
package helloservice;

public class HelloImpl implements HelloIF {

    public String message =“Hello“;

    public String sayHello(String s) {
        return message + s;
    }
}
```

Building the Service

To build `MyHelloService`, in a terminal window go to the *<INSTALL>*`/jwstutorial13/examples/jaxrpc/helloservice/` directory and type the following:

```
ant build
```

The preceding command executes these`ant` tasks:

- `compile-service`
- `generate-sei-service`
- `package-service`
- `process-war`

compile-service

This`ant` task compiles `HelloIF.java` and `HelloImpl.java`, writing the class files to the `build` subdirectory.

a. Click Next.

generate-sei-service

The `generate-sei-service` task runs the `wscompile` tool, which defines the service by creating the `model.gz` file in the `build` directory. The `model.gz` file contains the internal data structures that describe the service. The `generate-sei-service` task runs `wscompile` as follows:

```
wscompile -define -d build -nd build
-classpath build config-interface.xml -model build/model.gz
```

The `-define` flag instructs the tool to read the service endpoint interface and to create a WSDL file. The `-d` and `-nd` flags tell the tool to write output to the `build` subdirectory. The tool reads the following `config-interface.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <service
      name="MyHelloService"
      targetNamespace="urn:Foo"
      typeNamespace="urn:Foo"
      packageName="helloservice">
      <interface name="helloservice.HelloIF"/>
  </service>
</configuration>
```

The `config.xml-interface` file tells `wscompile` to create a model file with the following information:

- The service name is `MyHelloService`.
- The WSDL namespace is `urn:Foo`. (To understand this namespace, you need to be familiar with WSDL technology. See Further Information, page 516)
- The classes for the `MyHelloService` are in the `helloservice` package.
- The service endpoint interface is `helloservice.HelloIF`.

(If you are familiar with SOAP and WSDL, note that by default the service will be rpc/encoded. For doc/literal, see Advanced JAX-RPC Examples, page 477.)

package-service

The `package-service` 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/model.gz
WEB-INF/web.xml
```

The class files were created by the `compile-service` target discussed in a 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=”urn:Foo”
    typeNamespaceBase=”urn:Foo”
    urlPatternBase=”/ws”>

    <endpoint
        name=”MyHello”
        displayName=”HelloWorld Service”
        description=”A simple web service”
        interface=”helloservice.HelloIF”
        model=”/WEB-INF/model.gz”
        implementation=”hello.HelloImpl”/>

    <endpointMapping
        endpointName=”MyHello”
        urlPattern=”/hello”/>

</webServices>
```

Several of the `webServices` attributes, such as `targetNamespaceBase`, are used in the WSDL file, which is created by the `process-war` task described in the next section. (WSDL files can be complex and are not discussed in this tutorial. See Types Supported By JAX-RPC, page 456). Note that the `urlPattern` value (`/hello`) is part of the service’s URL, which is described in the section Verifying the Deployment (page 464)).

process-war

This `ant` task runs the `wsdeploy` tool as follows:

```
wsdeploy -o dist/hello-jaxrpc.war
dist/hello-jaxrpc-portable.war
```

The `wsdeploy` tool performs these tasks:

- Reads the `dist/hello-jaxrpc-portable.war` file as input
- Gets information from the `jaxrpc-ri.xml` file that's inside the `hello-jaxrpc-portable.war` file
- Generates the tie classes for the service
- Generates a WSDL file named `MyHelloService.wsdl`
- Packages the tie classes, the `MyHelloService.wsdl` file, and the contents of `hello-jaxrpc-portable.war` file into a deployable WAR file named `dist/hello-jaxrpc.war`

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`.

Deploying the Service

Type the following command:

```
ant deploy
```

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 lists the URL of 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 is the alias of the servlet. This string 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 the section, package-service (page 462).

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
```

Creating Web Service Clients with JAX-RPC

This section shows how to create and run these types of clients:

- Static stub
- Dynamic proxy
- Dynamic invocation interface (DII)

When you run these client examples, they will access the `MyHelloService` that you deployed in the preceding section.

Static Stub Client Example

This example resides in the `<INSTALL>/jwstutorial13/examples/jaxrpc/staticstub/` directory.

`HelloClient` is a stand-alone program that calls the `sayHello` method of the `MyHelloService`. It makes this call through a stub, a local object which acts as a proxy for the remote service. Because the stub is created before runtime (by `wscompile`), it is usually called a *static stub.*

Coding the Static Stub Client

Before it can invoke the remote methods on the stub the client performs these steps:

1. Creates a `Stub` object:

   ```
   (Stub)(new MyHelloService_Impl().getHelloIFPort())
   ```

   The code in this method is implementation-specific because it relies on a `MyHelloService_Impl` object, which is not defined in the specifications. The `MyHelloService_Impl` class will be generated by `wscompile` in the following section.

2. Sets the endpoint address that the stub uses to access the service:

   ```
   stub._setProperty
   (javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY, args[0]);
   ```

   At runtime, the endpoint address is passed to `HelloClient` in `args[0]` as a command-line parameter, which `ant` gets from the `endpoint.address` property in the `build.properties` file.

3. Casts `stub` to the service endpoint interface, `HelloIF`:

   ```
   HelloIF hello = (HelloIF)stub;
   ```

Here is the full source code listing for the `HelloClient.java` file, which is located in the directory `<INSTALL>/jwstutorial13/examples/jaxrpc/staticstub/src/`:

```
package staticstub;

import javax.xml.rpc.Stub;

public class HelloClient {

    private String endpointAddress;

    public static void main(String[] args) {

        System.out.println("Endpoint address = " + args[0]);
        try {
            Stub stub = createProxy();
            stub._setProperty
              (javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,
               args[0]);
            HelloIF hello = (HelloIF)stub;
            System.out.println(hello.sayHello("Duke!"));
        } catch (Exception ex) {
```

```
                ex.printStackTrace();
            }
        }

        private static Stub createProxy() {
           // Note: MyHelloService_Impl is implementation-specific.
             return
             (Stub) (new MyHelloService_Impl().getHelloIFPort());
        }
    }
```

Building the Static Stub Client

Before performing the steps in this section, you must first create and deploy `MyHelloService` as described in Creating a Web Service with JAX-RPC (page 459).

To build and package the client, go to the *<JWSDP_HOME>*`/docs/tutorial/examples/jaxrpc/staticstub/` directory and type the following:

```
ant build
```

The preceding command invokes these `ant` tasks:

- `generate-stubs`
- `compile-client`
- `package-client`

The `generate-stubs` task runs the `wscompile` tool as follows:

```
wscompile -gen:client -d build -classpath build config-wsdl.xml
```

This `wscompile` command reads the WSDL file that was installed on Tomcat when the service was deployed. The `wscompile` command generates files based on the information in the WSDL file and on the command-line flags. The `-gen:client` flag instructs `wscompile` to generate the stubs, other runtime files such as serializers, and value types. The `-d` flag tells the tool to write the output to the `build` subdirectory. The tool reads the following `config-wsdl.xml` file, which specifies the location of the WSDL file:

```
<?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="staticstub"/>
</configuration>
```

The `compile-client` task compiles `src/HelloClient.java` and writes the class file to the `build` subdirectory.

The `package-client` task packages the files created by the `generate-stubs` and `compile-client` tasks into the `dist/client.jar` file. Except for the `HelloClient.class`, all of the files in `client.jar` were created by `wscompile`. Note that `wscompile` generated the `HelloIF.class` based on the information it read from the WSDL file.

Running the Static Stub 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 <endpoint-address>
```

The classpath includes the `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.

Dynamic Proxy Client Example

This example resides in the `<INSTALL>/jwstutorial13/examples/jaxrpc/dynamicproxy/` directory.

The client in the preceding section 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. Although the source code for the static stub client relied on an implementation-specific class, the code for the dynamic proxy client does not have this limitation.

Coding the Dynamic Proxy Client

The `DynamicProxyHello` program constructs the dynamic proxy as follows:

1. Creates a `Service` object named `helloService`:

   ```
   Service helloService =
       serviceFactory.createService(helloWsdlUrl,
       new QName(nameSpaceUri, serviceName));
   ```

   A `Service` object is a factory for proxies. To create the `Service` object (`helloService`), the program calls the `createService` method on another type of factory, a `ServiceFactory` object.

   The `createService` method has two parameters, the URL of the WSDL file and a `QName` object. At runtime, the client gets information about the service by looking up its WSDL. In this example, the URL of the WSDL file points to the WSDL that was deployed with `MyHelloService`:

   ```
   http://localhost:8080/hello-jaxrpc/hello?WSDL
   ```

   A `QName` object is a tuple that represents an XML qualified name. The tuple is composed of a namespace URI and the local part of the qualified name. In the `QName` parameter of the `createService` invocation, the local part is the service name, `MyHelloService`.

2. The program creates a proxy (`myProxy`) with a type of the service endpoint interface (`HelloIF`):

   ```
   dynamicproxy.HelloIF myProxy =
      (dynamicproxy.HelloIF)helloService.getPort(
      new QName(nameSpaceUri, portName),
      dynamicproxy.HelloIF.class);
   ```

   The `helloService` object is a factory for dynamic proxies. To create `myProxy`, the program calls the `getPort` method of `helloService`. This method has two parameters: a `QName` object that specifies the port name and a `java.lang.Class` object for the service endpoint interface (`HelloIF`). The `HelloIF` class is generated by `wscompile`. The port name (`HelloIFPort`) is specified by the WSDL file.

Here is the listing for the `HelloClient.java` file, located in the *`<INSTALL>`*`/jwstutorial13/examples/jaxrpc/dynamicproxy/src/` directory:

```
package dynamicproxy;

import java.net.URL;
import javax.xml.rpc.Service;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceFactory;
import dynamicproxy.HelloIF;

public class HelloClient {

    public static void main(String[] args) {
        try {

            String UrlString = args[0] + "?WSDL";
            String nameSpaceUri = "urn:Foo";
            String serviceName = "MyHelloService";
            String portName = "HelloIFPort";

            System.out.println("UrlString = " + UrlString);
            URL helloWsdlUrl = new URL(UrlString);

            ServiceFactory serviceFactory =
                ServiceFactory.newInstance();

            Service helloService =
                serviceFactory.createService(helloWsdlUrl,
                new QName(nameSpaceUri, serviceName));

            dynamicproxy.HelloIF myProxy =
                (dynamicproxy.HelloIF)
                helloService.getPort(
                new QName(nameSpaceUri, portName),
                dynamicproxy.HelloIF.class);

            System.out.println(myProxy.sayHello("Buzz"));

        } catch (Exception ex) {
```

```
                ex.printStackTrace();
            }
        }
    }
```

Building and Running the Dynamic Proxy Client

Before performing the steps in this section, you must first create and deploy `MyHelloService` as described in Creating a Web Service with JAX-RPC (page 459).

To build and package the client, go to the `<INSTALL>/jwstutorial13/examples/jaxrpc/dynamicproxy/` directory and type the following:

```
ant build
```

The preceding command runs these tasks:

- `generate-interface`
- `compile-client`
- `package-dynamic`

The `generate-interface` task runs `wscompile with the -import` option. The `wscompile` command reads the `MyHelloService.wsdl` file and generates the service endpoint interface class (`HelloIF.class`). Although this `wscompile` invocation also creates stubs, the dynamic proxy client does not use these stubs, which are required only by static stub clients.

The `compile-client` task compiles the `src/HelloClient.java` file.

The `package-dynamic` task creates the `dist/client.jar` file, which contains `HelloIF.class` and `HelloClient.class`.

To run the client, type the following:

```
ant run
```

The client should display the following line:

```
Hello Buzz
```

Dynamic Invocation Interface (DII) Client Example

This example resides in the `<INSTALL>/jwstutorial13/examples/jaxrpc/dii/` directory.

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. In contrast to a static stub or dynamic proxy client, a DII client does not require runtime classes generated by wscompile. However, as you'll see in the following section, the source code for a DII client is more complicated than the code of the other two types of clients.

This example is for advanced users who are familiar with WSDL documents. (See Further Information, page 516.)

Coding the DII Client

The `DIIHello` program performs these steps:

1. Creates a `Service` object.

   ```
   Service service =
       factory.createService(new QName(qnameService));
   ```

   To get a `Service` object, the program invokes the `createService` method of a `ServiceFactory` object. The parameter of the `createService` method is a `QName` object that represents the name of the service, `MyHelloService`. The WSDL file specifies this name as follows:

   ```
   <service name="MyHelloService">
   ```

2. From the `Service` object, creates a `Call` object:

   ```
   QName port = new QName(qnamePort);
   Call call = service.createCall(port);
   ```

   A `Call` object supports the dynamic invocation of the remote procedures of a service. To get a `Call` object, the program invokes the `Service` object's `createCall` method. The parameter of `createCall` is a `QName` object that represents the service endpoint interface, `MyHelloServiceRPC`. In the WSDL file, the name of this interface is designated by the `portType` element:

   ```
   <portType name="HelloIF">
   ```

3. Sets the service endpoint address on the `Call` object:

```
call.setTargetEndpointAddress(endpoint);
```

In the WSDL file, this address is specified by the `<soap:address>` element.

4. Sets these properties on the `Call` object:

```
SOAPACTION_USE_PROPERTY
SOAPACTION_URI_PROPERTY
ENCODING_STYLE_PROPERTY
```

To learn more about these properties, refer to the SOAP and WSDL documents listed in Further Information (page 516).

5. Specifies the method's return type, name, and parameter:

```
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);
```

To specify the return type, the program invokes the `setReturnType` method on the `Call` object. The parameter of `setReturnType` is a `QName` object that represents an XML string type.

The program designates the method name by invoking the `setOperationName` method with a `QName` object that represents `sayHello`.

To indicate the method parameter, the program invokes the `addParameter` method on the `Call` object. The `addParameter` method has three arguments: a `String` for the parameter name (`String_1`), a `QName` object for the XML type, and a `ParameterMode` object to indicate the passing mode of the parameter (`IN`).

6. Invokes the remote method on the `Call` object:

```
String[] params = { "Murphy" };
String result = (String)call.invoke(params);
```

The program assigns the parameter value (`Murphy`) to a `String` array (`params`) and then executes the `invoke` method with the `String` array as an argument.

Here is the listing for the `HelloClient.java` file, located in the *<INSTALL>*/`jwstutorial13/examples/jaxrpc/dii/src/` directory:

```
package dii;

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 qnameService = “MyHelloService”;
    private static String qnamePort = “HelloIF”;

    private static String BODY_NAMESPACE_VALUE =
        “urn:Foo”;
    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) {

        System.out.println(“Endpoint address = “ + args[0]);

        try {
            ServiceFactory factory =
                ServiceFactory.newInstance();
            Service service =
                factory.createService(
                new QName(qnameService));

            QName port = new QName(qnamePort);

            Call call = service.createCall(port);
            call.setTargetEndpointAddress(args[0]);

            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 = { "Murph!" };

            String result = (String)call.invoke(params);
            System.out.println(result);

        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Building and Running the DII Client

Before performing the steps in this section, you must first create and deploy `MyHelloService` as described in Creating a Web Service with JAX-RPC (page 459).

To build and package the client, go to the *<INSTALL>*`/jwstutorial13/examples/jaxrpc/dii/` directory and type the following:

```
ant build
```

This `build` task compiles `HelloClient` and packages it into the `dist/client.jar` file. Unlike the previous client examples, the DII client does not require files generated by `wscompile`.

To run the client, type this command:

```
ant run
```

The client should display this line:

```
Hello Murph!
```

More JAX-RPC Client Examples

Other chapters in this book also have JAX-RPC client examples:

- Chapter 25 includes a static stub client in the Web tier. See the section, Coffee Break Server (page 1026).
- Chapter 24 describes a static stub client that demonstrates authentication. See the section, Example: Basic Authentication with JAX-RPC (page 981).
- Chapter 19 shows how a JSP page can be a static stub client that accesses a remote Web service. See the section, The Example JSP Pages (page 758).

In this chapter, the section Advanced JAX-RPC Examples (page 477) describes JAX-RPC clients with features such as SOAP message handlers and WS-I compliance.

Web Services Interoperability (WS-I) and JAX-RPC

JAX-RPC 1.1 supports the WS-I Basic Profile Version 1.0, Working Group Approval Draft. The WS-I Basic Profile is a document that clarifies the SOAP 1.1 and WSDL 1.1 specifications in order to promote SOAP interoperability. (For links related to WS-I, see Further Information, page 516.)

To support WS-I, JAX-RPC has the following features:

- When run with the `-f:wsi` option, `wscompile` verifies that a WSDL is WS-I compliant and/or generates classes needed by JAX-RPC services and clients that are WS-I compliant.
- The JAX-RPC runtime supports doc/literal and rpc/literal for services, static stubs, dynamic proxies, and DII.

For coding examples of WS-I compliant clients and services, see the section Advanced JAX-RPC Examples (page 477)

Advanced JAX-RPC Examples

This section describes the code examples in the `<INSTALL>/jwstutorial13/examples/jaxrpc/advanced/` directory. To understand these examples, you should already be familiar with the following:

- Ant
- JAX-RPC fundamentals
- SOAP
- WSDL
- WS-I Basic Profile

The following sections describe the advanced examples:

Note: Before proceeding, make sure that you've followed the instructions in Setting Up (page 49) for setting the `PATH` variable and editing the `common/build.properties` file.

SOAP Message Handlers Example

JAX-RPC makes it easy to develop Web services and clients because it shields application developers from the underlying SOAP messages. Instead of writing code to build and parse SOAP messages, application developers merely implement the service methods and invoke them from remote clients. The handler processing is hidden from the JAX-RPC client and service implementation code.

However, there are times when you might want to access the SOAP messages that flow between JAX-RPC clients and services. For example, for auditing purposes, you might want to log every invocation of a service method. Or, you might want to encrypt remote calls at the SOAP message level. These logging and encrypting operations can be implemented with SOAP message handlers.

A SOAP message handler is a stateless instance that accesses SOAP messages representing RPC requests, responses, or faults. Tied to service endpoints, handlers enable you to process SOAP messages and to extend the functionality of the service. For a given service endpoint, one or more handlers may reside on the server and client.

Handler Processing

A SOAP request is processed as follows:

- The client handler is invoked before the SOAP request is sent to the server.
- The service handler is invoked before the SOAP request is dispatched to the service endpoint.

A SOAP response is processed in this order:

- The service handler is invoked before the SOAP response is sent back to the client.
- The client handler is invoked before the SOAP response is transformed into a Java method return and passed back to the client program.

A SOAP fault is processed in the same manner as a SOAP response.

Clients and servers may have multiple handlers, which are configured into handler chains. For example, in a client handler chain with three handlers, the first handler processes the SOAP request, then the second processes the request, and then the third. When the third handler finishes, the request is sent to the server.

Overview of Handler Example

The client and service implementation code are quite simple. The client invokes the service method named `helloServer`, which echoes a `String` back to the client.

The client JAR file includes the following classes:

- `HandlerSampleClient` - the client program that invokes `helloServer`
- `ClientHandler1` - a client request handler

The service WAR includes these classes:

- `HandlerTestImpl` - the service implementation class that implements the `helloServer` method
- `ServerHandler1` - a server request handler

- `ServerHandler2` - another server request handler, invoked after `ServerHandler1`

Handler Programming Model

The process for building the service with a handler requires these steps:

1. Write the code for the service endpoint interface, service implementation class, and server handler classes
2. Create the `jaxrpc-ri.xml` file.

   Read by `wsdeploy`, the `jaxrpc-ri.xml` file has information about the service handlers. For details see the section, The Service jaxrpc-ri.xml File (page 486).
3. Create the `web.xml` file.
4. Compile the code from step 1.
5. Package the `web.xml` file, `jaxrpc-ri.xml` file, and compiled classes, into a raw WAR file (`raw.war`).
6. Run `wsdeploy` to cook the raw WAR file and create a deployable WAR file (`handler.war`).
7. Deploy the WAR file.

   The deployed service has this endpoint address URL:

```
http://localhost:8080/handler/test
```

For a client with a handler, do the following:

1. Code the client program and the handler.
2. Create the `config.xml` file.

   This file is read by `wscompile` and contains information about the client handler. For details see the section, The Client config.xml File (page 483).
3. Compile the client handler.
4. Run `wscompile` to generate the service endpoint interface and client-side classes.

   The `wscompile` command accesses the deployed WSDL file specified in `config.xml`.
5. Compile the client program.

The client is now ready to be run.

Building and Running the Handler Example

To build and run this example, go to the *<INSTALL>*/jwstutorial13/examples/jaxrpc/advanced/handler directory and type the following commands:

```
ant build
ant deploy
ant build-client
ant run
```

The ant run tasks executes the client program, which should display the following:

```
ClientHandler1: name = My Client Handler
ClientHandler1: adding loggerElement
ClientHandler1: adding nameElement
HandlerSampleClient: hi there
HandlerSampleClient: all done.
```

At runtime, the example handlers and service implementation write the following lines to the *<JWSDP_HOME>*/logs/launcher.server.log file:

```
ServerHandler1: name = server1
ServerHandler2: name = server2
ServerHandler1: important message (level 10)
ServerHandler2: Request is from Duke
TestHandlerImpl: helloServer() message = hi there
```

Client Handler

The ClientHandler1 instance processes the SOAP request before it is transmitted to the service endpoint. The source code for ClientHandler1 is in the *<INSTALL>*/jwstutorial13/examples/jaxrpc/advanced/handler/client/ directory.

Like the other handlers in this example, ClientHandler1 is a subclass of javax.xml.rpc.handler.GenericHandler:

```
public class ClientHandler1 extends GenericHandler . . .
```

GenericHandler is an abstract class that implements javax.xml.rpc.handler.Handler, an interface that defines the methods required by any handler. Because it provides default implementations for these methods, GenericHandler makes it easy to write your own handlers.

Of the several methods implemented by `GenericHandler`, `ClientHandler1` overrides only these methods:

- `init`
- `getHeaders`
- `handleRequest`

The init Method of ClientHandler1

The `init` life-cycle method enables the `Handler` instance to initialize itself. Typically, you'll implement the `init` method to connect to resources such as databases. You can set instance variables in the `init` method, but be aware that a handler is stateless-- the next time a handler is invoked, it might have a different state. Prior to termination, the handler's `destroy` method is invoked. If you'd connected to resources in the `init` method, you might need to disconnect from them in the `destroy` method.

The `init` method of `ClientHandler1` follows. It fetches the value of the property name, which was set in the `config.xml` file. (See The Client config.xml File, page 483.)

```
public void init(HandlerInfo info) {
    handlerInfo = info;
    name = (String) info.getHandlerConfig().get("name");
    System.out.println("ClientHandler1: name = " + name);
}
```

The getHeaders Method of ClientHandler1

The `getHeaders` method retrieves the header blocks of the message processed by this handler. Because `getHeaders` is declared as `abstract` in `GenericHandler`, it must be implemented in `ClientHandler1`:

```
public QName[] getHeaders() {
    return handlerInfo.getHeaders();
}
```

The handleRequest Method of ClientHandler1

A handler may process a SOAP request, response, or fault. Defined by the `Handler` interface, the `handleRequest`, `handleResponse`, and `handleFault` methods perform the actual processing of the SOAP messages. The `ClientHandler1` class implements the `handleRequest` method, listed at the end of this section.

The handleRequest method of ClientHandler1 gets access to the SOAP message from the SOAPMessageContext parameter. (For more information about the SOAP APIs, see Chapter 13.) The method adds two SOAPHeaderElement objects to the SOAP request: loggerElement and nameElement. The loggerElement header will be processed by ServiceHandler1 and the nameElement header by ServiceHandler2. The ServiceHandler1 instance will check the logging level that ClientHandler1 set by invoking loggerElement.setValue. The ServiceHandler2 instance will retrieve the Duke string from the header that ClientHandler1 specified when calling nameElement.addTextNode.

A handler is associated with a SOAP actor (role). If a header element is targeted at a specific actor and the header element has the MustUnderstand attribute set to 1, and if a server request handler uses that actor, then the server handler must process the element. In this case, if the server handler of the targeted actor does not process the header, then a MustUnderstand fault will be thrown. If the server handler uses a different actor than the header element, then the MustUnderstand fault will not be thrown. All of the handlers in this example use the default actor "next." ClientHandler1 adds header elements and invokes setMustUnderstand(true) on loggerElement. The ServerHandler1 program will process loggerElement accordingly.

Here is the handleRequest method of ClientHandler1:

```
public boolean handleRequest(MessageContext context) {
    try {
        SOAPMessageContext smc = (SOAPMessageContext) context;
        SOAPMessage message = smc.getMessage();
        SOAPPart soapPart = message.getSOAPPart();
        SOAPEnvelope envelope = soapPart.getEnvelope();
        SOAPHeader header = message.getSOAPHeader();
        if (header == null) {
            header = envelope.addHeader();
        }

        System.out.println
            ("ClientHandler1: adding loggerElement");
        SOAPHeaderElement loggerElement =
           header.addHeaderElement
           (envelope.createName("loginfo",
           "ns1", "http://example.com/headerprops"));
        loggerElement.setMustUnderstand(true);
        loggerElement.setValue("10");

        System.out.println
            ("ClientHandler1: adding nameElement");
```

```
        SOAPHeaderElement nameElement =
            header.addHeaderElement
            (envelope.createName("clientname",
            "ns1", "http://example.com/headerprops"));
        nameElement.addTextNode("Duke");

    } catch (Exception e) {
        throw new JAXRPCException(e);
    }
    return true;
}
```

The Client config.xml File

The `ant build-client` task runs the `wscompile` command, which reads the `config.xml` file. Within this file, you declare handlers in `<handlerChains>`, an element that represents an ordered list of handlers. Because the client has only one handler, `<handlerChains>` contains a single `<handler>` subelement. The `runAt` attribute of the `<chain>` subelement specifies that this handler will run on the client side. The `className` attribute of the `<handler>` subelement identifies `client.ClientHandler1`. The optional `<property>` element assigns a name and value to a property that is passed to the handler instance in the `HandlerInfo` parameter of the `init` method. In `ClientHandler1`, the `init` method prints out the property value.

The `config.xml` file 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/handler/test?WSDL"
         packageName="handlersample">
    <handlerChains>
      <chain runAt="client">
        <handler className="client.ClientHandler1">
           <property name="name" value="My Client Handler"/>
        </handler>
      </chain>
    </handlerChains>

  </wsdl>
</configuration>
```

Server Handlers

This example has two service handlers: `ServerHandler1` and `ServerHandler2`. The source code for these handlers is in the `examples/jaxrpc/advanced/handler/service/` subdirectory.

ServerHandler1

This handler processes the SOAP request message that has been transmitted to the service endpoint.

The `init` method of `ServerHandler1` fetches the value of the property `name`, which was set in the `jaxrpc-ri.xml` file. The `init` method follows:

```
public void init(HandlerInfo info) {
    handlerInfo = info;
    name = (String) info.getHandlerConfig().get("name");
    System.out.println("ServerHandler1: name = " + name);
}
```

The `handleRequest` method iterates through the header elements until it finds one with the element name `loginfo`. This element was added to the request message by the client handler. The `handleRequest` method must process the element and then detach (remove) it because of the following two reasons: First, the client handler invoked `setMustUnderstand(true)` on the element. Second, the server handler is using the actor ("next") targeted by the header element. Also, the `jaxrpc-ri.xml` file must declare that `ServerHandler1` understands this particular element. (See The Service jaxrpc-ri.xml File, page 486.) Without this declaration, a `MustUnderstand` fault will be thrown.

To process the `loginfo` element, `handleRequest` invokes `getValue` and prints out a message if the log level is greater than 5. Because `ClientHandler1` set the level to 10, `handleRequest` prints the message.

Here is the code for the `handleRequest` method of `ServerHandler1`:

```
public boolean handleRequest(MessageContext context) {
    try {

        SOAPMessageContext smc = (SOAPMessageContext) context;
        SOAPMessage message = smc.getMessage();
        SOAPHeader header = message.getSOAPHeader();

        if (header != null) {
            Iterator iter = header.examineAllHeaderElements();
```

```
            while (iter.hasNext()) {
               SOAPElement element = (SOAPElement) iter.next();
                if
(element.getElementName().getLocalName().equals("loginfo")) {

                    int logLevel =
                       Integer.parseInt(element.getValue());
                    if (logLevel > 5) {
                        System.out.println
                           ("ServerHandler1: important " +
                          "message (level " + logLevel + ")");
                    } else {
                        // message not important enough to log
                    }
                    element.detachNode();
                    break;
               }
            }
        }

    } catch (Exception e) {
        throw new JAXRPCException(e);
    }

    return true;
}
```

ServerHandler2

After `ServerHandler1` finishes processing the SOAP request, `ServerHandler2` is invoked and processes the same request. Because both handlers are for the same service endpoint, they are packaged in the same WAR file and specified in the same `jaxrpc-ri.xml` file.

To fetch the header blocks of the message, the `init` method of `ServerHandler2` invokes `info.getHeaders`, which returns a array of `QName` (qualified name) objects. Next, the `init` method gets the value of the `name` property, which was declared in the `jaxrpc-ri.xml` file. The `getHeaders` method of `ServerHandler2` returns an array of `QName` objects that represent the header blocks. Here is the code for the `init` and `getHeaders` methods:

```
public void init(HandlerInfo info) {
    qnames = info.getHeaders();
    name = (String) info.getHandlerConfig().get("name");
    System.out.println("ServerHandler2: name = " + name);
}
```

```
public QName[] getHeaders() {
    return qnames;
}
```

The handleRequest methods of ServerHandler1 and ServerHandler2 began similarly. They both get the SOAP header from the SOAPMessageContext and then iterate the header elements until matching a name that was set by ClientHandler1. In ServerHandler2, the matching header element name is clientname. From this header element, handleRequest of ServerHandler2 extracts and then prints the String that ClientHandler1 passed to the addTextNode method. Here is a partial code listing of the handleRequest method of ServerHandler2:

```
. . .
if (header != null) {
    Iterator iter = header.examineAllHeaderElements();
    while (iter.hasNext()) {
        SOAPElement element = (SOAPElement) iter.next();
        if
(element.getElementName().getLocalName().equals("clientname"))
        {
            String clientName = element.getValue();
            System.out.println
                ("ServerHandler2: Request is from " +
                clientName);
        }
 }
. . .
```

The Service jaxrpc-ri.xml File

The ant build task compiles and packages the service files, including the handlers. During this task the wsdeploy command reads the jaxrpc-ri.xml file to get information about the service.

In jaxrpc-ri.xml, he <chain> subelement of <handlerChains> specifies server for the runAt attribute. Because the service endpoint has two handlers, the <handlerChains> element encloses two <handler> elements.

The first <handler> element declares that ServerHandler1 understands the loginfo header. This declaration is required because ClientHandler1 invoked setMustUnderstand(true) on loggerElement. The attributes of the ServerHandler1 <handler> element correspond to the parameters of the envelope.createName method invoked in ClientHandler1.

The second `<handler>` element is for `ServerHandler2`. For requests, by default multiple handlers are invoked in the order in which they appear in the `<handlerChains>` element. For responses and faults, the handlers are invoked in the reverse order.

Here is the `jaxrpc-ri.xml` file:

```
<?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="/test">

    <endpoint
        name="MyHandlerApp"
        displayName="An application for testing handlers"
        description="...description of app"
        interface="service.HandlerTest"
        implementation="service.HandlerTestImpl">

    <handlerChains>
      <chain runAt="server">
        <handler className="service.ServerHandler1"
            headers="ns1:loginfo"
            xmlns:ns1="http://example.com/headerprops">
          <property name="name" value="server1"/>
        </handler>

        <handler className="service.ServerHandler2">
          <property name="name" value="server2"/>
        </handler>
      </chain>
    </handlerChains>
    </endpoint>

    <endpointMapping
        endpointName="MyHandlerApp"
        urlPattern="/test"/>
</webServices>
```

Advanced Static Stub Example

This example demonstrates the following:

- Creating a service and a static stub client that use doc/literal and are WS-I compliant.
- Generating the wrapper classes required by literal.
- Starting with a WSDL file, creating a service and a static stub client.
- Making a one-way call from a static stub client to a WS-I compliant service.

The files for this example are in the `<INSTALL>/jwstutorial13/examples/jaxrpc/advanced/stubs/` directory.

Building and Running the Advanced Static Example

To build, deploy, and run this example, go to the `<INSTALL>/jwstutorial13/examples/jaxrpc/advanced/stubs/` directory and type the following:

```
ant build
ant deploy
ant build-client
ant run
```

The `run` task executes the `HelloClient` program, which should display the following lines:

```
Running echo String from a staticstub client.
Hi there Duke!!!

Running SimpleValueType from a staticstub client.
Echoing the boolean set in ValueType by server :false
Echoing the integer set in ValueType by server :54
Echoing the string set in ValueType by server  :Server Entry :
Test Data

Running the one way operation using ValuType from a staticstub
client

Running ComplexValuType from a staticstub client
Client output for testComplexValueType : 12345
Client output for testComplexValueType : 4.0
```

```
Client output for testComplexValueType : true

Original unsigned long in
ValueTypeWObjectMemberAObjectMemberArray : 129
Modified unsigned long in
ValueTypeWObjectMemberAObjectMemberArray  : 258

Running int[] from a staticstub client
Client output for testSimpleIntArray   : true
```

Service Programming Model for the Advanced Static Stub Example

With JWSDP, to create a JAX-RPC service you begin with either a service endpoint interface or a WSDL file. In the example described in Creating a Web Service with JAX-RPC (page 459), you started with a service endpoint interface. In this example, you'll start with a WSDL file before building the service.

The steps that follow outline the programming model for creating a WS-I compliant service when starting with a WSDL file follow. Steps 2, 4, 5, and 6 are performed by subtasks of the `ant build-service` task.

1. Get the WSDL file.

   Typically, you'd create a WSDL file with a development tool. For this example, the `conf/HelloWorldService.wsdl` file has been provided for you.

2. Generate the service endpoint interface and the other server-side classes.

   ant task: `generate-server`

   This task runs `wscompile` with the `-import`, `-model`, and `-f:wsi` options. The `wscompile` tool stores the generated classes and corresponding source code in the `build/classes/server/` subdirectory.

   For more information, see the sections Generating WS-I Compliant Service Files with wscompile (page 491) and Service Endpoint Interface Generated by wscompile (page 494).

3. Code the service implementation class.

   Located in the `src/server/` subdirectory, the `HelloImpl.java` code implements the service endpoint interface (`HelloIF`) generated in the preceding step. The `HelloImpl.java` file is included with the tutorial, so in this example you don't have to code it. If you did have write `Hel-`

`loImpl.java`, you'd first examine the generated `HelloIF.java` code for the signatures of the methods that you need to implement.

This approach might seem backwards, because with the help of an IDE such as SunONE Studio, you'd probably code the service implementation first and then generate the service endpoint interface and WSDL file. However, if you are developing a client for someone else's service, you might want to start with a WSDL to create a simple service for unit testing.

4. Compile the service implementation class.

   ant task: `compile-server-classes`

5. Create the raw WAR file.

   ant task: `create-raw-war`

   This step packages the server-side files into a raw WAR file. The term raw indicates that this WAR file is not ready for deployment. In this example, the raw WAR file resides in the `build/war/` subdirectory and is named `jaxrpc-DocumentLitHelloService-raw.war`.

6. Create a deployable (cooked) WAR file.

   ant task: `build-war`

   To cook the raw WAR file, you run the `wsdeploy` command. The cooked WAR file is named `jaxrpc-DocumentLitHelloService.war`. (For more information about `wsdeploy`, see the section process-war (page 464).)

7. Deploy the WAR file.

   ant task: `deploy`

   This action deploys the `HelloWorldService` at the following URL:

```
http://localhost:8080/jaxrpc-DoclitHelloService/hello
```

Generating WS-I Compliant Service Files with wscompile

Because the client and server in this example are WS-I compliant, `wscompile` is run with the `-f:wsi` option. The `ant generate-server` task runs `wscompile` as follows:

```
wscompile -keep -import
          -model model-DocumentLitHelloService.xml.gz
          -f:wsi conf/config-server.xml
```

Table Table 12–1 describes these options.

Table 12–2 wscompile Options for WS-I Compliance

Option	Description
-keep	Keep all generated files, including the source files. These files reside in the build/classes/server/hello/ subdirectory.
-import	Import the service description from the WSDL file listed in the configuration file (conf/config-server.xml) and then generate service interfaces and value types.
-model	Write information about the internal data structures used by the service to a model file. The wsdeploy command gets information from the model file that it needs for generating runtime classes. The model file is implementation specific and is not portable.
-f:wsi	Generate files for a WS-I compliant service or client.

The `wscompile` command requires a configuration file. When you start with a WSDL file, the configuration file must have a `<wsdl>` element, which specifies the location of the WSDL file. Here is the listing for `conf/config-server.xml`, the configuration file used in this example:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
    xmlns="http://java.sun.com/xml/ns/jax-rpc/pi/config">
    <wsdl
        location="conf/HelloWorldService.wsdl"
        packageName="hello" />
</configuration>
```

Client Programming Model for the Advanced Static Stub Example

When creating a client for a Web service, you usually begin with the WSDL file that's been made available by the service developers. The steps that follow briefly describe the process for creating WS-I compliant client when starting with a WSDL file. Note that steps 2 and 6 are subtasks of the ant `build-client` task.

1. Make sure that the service has been deployed.

   When you ran the `ant deploy` command, a WSDL file was deployed with the service. In the next step, the `wscompile` command that generates client files refers to the deployed WSDL file.

2. Generate service endpoint interface and client-side classes.

   ant task: `generate-client`

   This task runs `wscompile` with the `-gen:client` and `-f:wsi` options. The `wscompile` tool stores the generated classes and corresponding source code in the `build/classes/client/` subdirectory.

   For more information, see the sections Generating the Static Stub Client Files with wscompile (page 493) and Service Endpoint Interface Generated by wscompile (page 494).

3. Identify the signatures of the remote methods.

   A remote client may call the methods defined in the service endpoint interface. To identify the method signatures, examine the source code of the interface, which was generated by `wscompile` in the previous step.

4. Identify the wrapper classes.

For literal (not encoded) operations, the return types and parameters are within wrapper classes. When writing the client program, you'll need to know how to get and set the fields contained in the wrapper classes. Therefore, you'll need to examine the wrapper source code, also generated by `wscompile`. For an example, see the section Wrapper Classes for the sayHello Method (page 495).

5. Code the client program.

   Now that you're familiar with the remote method signatures and wrapper classes, you're ready to write the client code. In this example, the provided source code for the `HelloClient` program resides in the `src/client/` subdirectory.

6. Compile the client program and classes.

   ant task: `compile-client-classes`

   This task compiles the source code created in the previous step. After compilation, the client is ready to run.

Generating the Static Stub Client Files with wscompile

The `generate-client` task runs wscompile as follows:

```
wscompile -keep -gen:client -f:wsi conf/config-client.xml
```

The `-keep` and `-f:wsi` options are described in Table 12–2. The `-gen:client` option instructs `wscompile` to generate files for a client.

In the following listing of the configuration file (`conf/config-client.xml`), note that the `<wsdl>` element specifies the WSDL file that was deployed with the service.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
    <wsdl location=
    "http://localhost:8080/jaxrpc-DocumentLitHelloService/
hello?WSDL"
        packageName="hello" />
</configuration>
```

The wscompile tool generates class files required by the static stub client. The HelloClient program creates a stub as follows:

```
stub = (HelloIF_Stub)
    (new HelloWorldService_Impl().getHelloIFPort());
```

The wscompile tool generates the HelloIF_Stub and HelloWorldService_Impl classes and places them in the build/classes/server/ subdirectory. To construct the HelloIF_Stub name, wscompile appends _Stub to the portType name defined in the WSDL file:

```
<portType name="HelloIF">
```

To create the HelloWorldService_Impl name, wscompile appends _Impl to the service name, which the WSDL file specifies as follows:

```
<service name="HelloWorldService">
```

Service Endpoint Interface Generated by wscompile

The service endpoint interface defines the methods that a remote client can invoke. In this example, both invocations of wscompile generate a service endpoint interface named HelloIF:

```
package hello;

public interface HelloIF extends java.rmi.Remote {
    public hello.ChangeComplexValueTypeResponse
      changeComplexValueType(hello.ChangeComplexValueTyp
       parameters) throws java.rmi.RemoteException;
    public hello.ChangeValueTypeResponse
        changeValueType(hello.ChangeValueType parameters)
        throws java.rmi.RemoteException;
    public hello.ReverseArrayResponse
        reverseArray(hello.ReverseArray parameters)
        throws java.rmi.RemoteException;
    public hello.SayHelloResponse
         sayHello(hello.SayHello parameters)
         throws java.rmi.RemoteException;
}
```

Note that HelloIF.java, as well as the source code for other generated files, is in the build/classes/server/ and build/classes/client subdirectories.

Wrapper Classes for the sayHello Method

Because this example uses literal (not encoded), the parameters and return types of remote methods must be enclosed in wrapper classes. Because the client in this section is WS-I compliant, the `HelloIF` interface defines `sayHello` with the `SayHelloResponse` and `SayHello` wrapper classes:

```
public hello.SayHelloResponse
    sayHello(hello.SayHello parameters)
    throws java.rmi.RemoteException;
```

The name of the wrapper class for the return type is the capitalized name of the method plus `Response`. For example, the `SayHelloResponse` class is the return type for the `sayHello` method. The `SayHelloResponse` class is a wrapper for a `String` variable, which can be accessed through a getter and a setter. The listing for `SayHelloResponse` follows.

```
package hello;


public class SayHelloResponse {
    protected java.lang.String result;

    public SayHelloResponse() {
    }

    public SayHelloResponse(java.lang.String result) {
        this.result = result;
    }

    public java.lang.String getResult() {
        return result;
    }

    public void setResult(java.lang.String result) {
        this.result = result;
    }
}
```

The parameters for the remote call are wrapped in a single class, `SayHello`. The name of the class is the capitalized name of the method. In the listing `SayHello` that follows, note that it wraps two String parameters and provides a getter and setter for each String.

```
package hello;


public class SayHello {
    protected java.lang.String string_1;
    protected java.lang.String string_2;

    public SayHello() {
    }

    public SayHello(java.lang.String string_1,
       java.lang.String string_2) {
        this.string_1 = string_1;
        this.string_2 = string_2;
    }

    public java.lang.String getString_1() {
        return string_1;
    }

    public void setString_1(java.lang.String string_1) {
        this.string_1 = string_1;
    }

    public java.lang.String getString_2() {
        return string_2;
    }

public void setString_2(java.lang.String string_2) {
        this.string_2 = string_2;
    }
}
```

When it invokes the sayHello method, the HelloClient program wraps the two String parameters in the SayHello class. To fetch the String returned by the method, the program invokes getResult on the SayHelloResponse wrapper class. The HelloClient program invokes sayHello as follows:

```
System.out.println(stub.sayHello(new SayHello
    (“Hi there “, “Duke!!!”)).getResult());
```

Wrapper Classes for the reverseArray Method

The `reverseArray` method accepts an `int` array and returns the array in reverse order. The service endpoint interface, `HelloIF`, defines `reverseArray` as follows:

```
public hello.ReverseArrayResponse
    reverseArray(hello.ReverseArray parameters)
    throws java.rmi.RemoteException;
```

The `reverseArray` method uses three wrapper classes:

- `IntArrayTest` - wrapper of an `int` array
- `ReverseArray` - method parameter and wrapper of `IntArrayTest`
- `ReverseArrayResponse` - method return type and wrapper of `IntArrayTest`

The `ReverseArray` and `ReverseArrayResponse` classes contain the `IntArrayTest` class, which in turn contains an `int` array. In effect, the `int` array is wrapped twice.

In the client, setting the method parameter requires two steps:

1. Setting the int array contained in `IntArrayTest`.
2. Setting the `IntArrayTest` in `ReverseArray`.

Similarly, getting the method return value also requires two steps:

1. Getting the `IntArrayTest` from `ReverseArrayResponse`.
2. Getting the `int` array from `IntArrayTest`.

The `HelloClient` program invokes `reverseArray` in the following code:

```
int[] intArray = {1,4,7,9,11};
IntArrayTest param = new IntArrayTest(intArray);
ReverseArrayResponse result1 =
    stub.reverseArray(new ReverseArray(param));
IntArrayTest result = result1.getResult();
int[] rintArray = result.getIntArray();
System.out.println
    ("\nRunning int[] from a staticstub client ");
System.out.println("Client output for testSimpleIntArray   : "
     + java.util.Arrays.equals(intArray, rintArray));
```

Generated by `wscompile`, the `IntArrayTest.java` file resides in the `build/classes/client/` subdirectory. In the `IntArrayTest` listing that follows, note that the `int` array field may be set either with a constructor or the `setIntArray` method.

```
package hello;


public class IntArrayTest {
    protected int[] intArray;

    public IntArrayTest() {
    }

    public IntArrayTest(int[] intArray) {
        this.intArray = intArray;
    }

    public int[] getIntArray() {
        return intArray;
    }

    public void setIntArray(int[] intArray) {
        this.intArray = intArray;
    }
}
```

The `ReverseArray` parameter class wraps `IntArrayTest` provides a getter, setter, and constructors:

```
package hello;


public class ReverseArray {
    protected hello.IntArrayTest intArrayTest_1;

    public ReverseArray() {
    }

    public ReverseArray(hello.IntArrayTest intArrayTest_1) {
        this.intArrayTest_1 = intArrayTest_1;
    }

    public hello.IntArrayTest getIntArrayTest_1() {
        return intArrayTest_1;
    }

    public void setIntArrayTest_1(hello.IntArrayTest
```

```
            intArrayTest_1) {
             this.intArrayTest_1 = intArrayTest_1;
        }
    }
```

Like the `ReverseArray` class, the `ReverseArrayResponse` class wraps `IntArrayTest`:

```
    package hello;


    public class ReverseArrayResponse {
        protected hello.IntArrayTest result;

        public ReverseArrayResponse() {
        }

        public ReverseArrayResponse(hello.IntArrayTest result) {
            this.result = result;
        }

        public hello.IntArrayTest getResult() {
            return result;
        }

        public void setResult(hello.IntArrayTest result) {
            this.result = result;
        }
    }
```

Wrapper Classes for the changeValueType Method

A value type is a user-defined class with a state that may be passed as a method parameter or return type. A value type often represents a logical entity such as a customer or a purchase order. This example demonstrates the use of a class named `ValueType`, a user-defined class with a state that consists of three properties: a `boolean`, an `Integer`, and a `String`. Each of these properties has a getter and setter method.

The `wscompile` tool generates the `ValueType` class and source code files, placing them in the `build/classes/server/` and `build/classes/client` subdirectories. Here is the source code for `ValueType`:

```
package hello;


public class ValueType {
    protected boolean boolProperty;
    protected java.lang.Integer integerProperty;
    protected java.lang.String stringProperty;

    public ValueType() {
    }

    public ValueType(boolean boolProperty, java.lang.Integer
      integerProperty, java.lang.String stringProperty) {
        this.boolProperty = boolProperty;
        this.integerProperty = integerProperty;
        this.stringProperty = stringProperty;
    }

    public boolean isBoolProperty() {
        return boolProperty;
    }

    public void setBoolProperty(boolean boolProperty) {
        this.boolProperty = boolProperty;
    }

    public java.lang.Integer getIntegerProperty() {
        return integerProperty;
    }

    public void setIntegerProperty(java.lang.Integer
       integerProperty) {
        this.integerProperty = integerProperty;
    }

    public java.lang.String getStringProperty() {
        return stringProperty;
    }

    public void setStringProperty(java.lang.String
       stringProperty) {
        this.stringProperty = stringProperty;
    }
}
```

The changeValueType method wraps ValueType in the ChangeValueType parameter and the ChangeValueTypeResponse return value. The service endpoint interface defines changeValueType as follows:

```
public hello.ChangeValueTypeResponse
    changeValueType(hello.ChangeValueType parameters)
    throws java.rmi.RemoteException;
```

The following code listing shows how the HelloImpl class implements the changeValueType method. (The HelloImpl.java file is in the src/server/ subdirectory.) The changeValue method extracts the ValueType parameter from the ChangeValueType wrapper class by invoking the getValueType_1 method. Then the method alters the ValueType state by invoking the setter method for each property. Finally, the method returns the altered ValueType wrapped in a ChangeValueTypeResponse.

```
public ChangeValueTypeResponse changeValueType(ChangeValueType
evt) {

    System.out.println("in echoValue type : " +
        evt.getValueType_1());
    ValueType vt = evt.getValueType_1();
    String str = vt.getStringProperty();
    vt.setStringProperty( "Server Entry : " + str);
    vt.setIntegerProperty(new Integer(54));
    vt.setBoolProperty(false);
    return (new ChangeValueTypeResponse( vt ));
}
```

The following code snippet shows how the HelloClient program invokes the changeValueType method.

```
ValueType vt = stub.changeValueType(new
ChangeValueType(valueType)).getResult();
System.out.println
    ("Echoing the boolean set in ValueType by server :"
    + vt.isBoolProperty());
System.out.println
    ("Echoing the integer set in ValueType by server :"
    + vt.getIntegerProperty().intValue());
System.out.println
    ("Echoing the string set in ValueType by server  :"
    + vt.getStringProperty());
```

The ChangeValueType and ChangeValueType classes and source code reside in the build/client/classes and build/server/classes subdirectories. If you examine the source code, you can see the getter and setter methods that are invoked by the implementation class (HelloImpl) and client program (HelloClient).

Wrapper Classes for the changeComplexValueType Method

This method shows how to pass a value type that contains several simple types (such as String and Calendar), an int array, and another value type. The HelloIF interface defines changeComplexValueType as follows:

```
public hello.ChangeComplexValueTypeResponse
     changeComplexValueType(hello.ChangeComplexValueType
     parameters) throws java.rmi.RemoteException;
```

ChangeComplexValueTypeResponse and ChangeComplexValueType are wrappers for ValueTypeWObjectMemberAObjectMemberArray, a complex type:

```
public class ValueTypeWObjectMemberAObjectMemberArray {
    protected java.util.Calendar calender1;
    protected java.util.Calendar calender2;
    protected long longUnsignedInt;
    protected hello.BaseFooObject myValueType;
    protected hello.IntArrayTest simpleArray;
    protected java.lang.String simpleString;
. . .
```

IntArrayTest is a wrapper for an int array:

```
public class IntArrayTest {
    protected int[] intArray;
. . .
```

BaseFooObject is another value type:

```
public class BaseFooObject {
    protected java.math.BigInteger first;
    protected double second;
    protected boolean third;
. . .
```

In the `HelloClient` program, the `testComplexValueType` method invokes service's `changeComplexValueType` method. The `testComplexValueType` method creates a `ValueTypeWObjectMemberAObjectMemberArray` object and passes it as the parameter for `changeComplexValueType`. To extract the values in the `ChangeComplexValueTypeResponse` object returned by `changeComplexValueType`, the `testComplexValueType` method invokes several getter methods. Here is the source code for `testComplexValueType`:

```
public void testComplexValueType() throws Exception {
    BaseFooObject baseFoo =
       new BaseFooObject(new BigInteger("12345"), 04, true);
    int[] intArray = {1,4,7,9,11};
    IntArrayTest simpleArray = new IntArrayTest(intArray);

    ValueTypeWObjectMemberAObjectMemberArray param =
       new ValueTypeWObjectMemberAObjectMemberArray(
       new java.util.GregorianCalendar(),
       new java.util.GregorianCalendar(), 129, baseFoo,
       simpleArray, "fooString");
    ChangeComplexValueTypeResponse result1 =
    stub.changeComplexValueType
        (new ChangeComplexValueType(param));
    ValueTypeWObjectMemberAObjectMemberArray result =
        result1.getResult();
    BaseFooObject rfoo = result.getMyValueType();

    System.out.println
    ("\nRunning ComplexValuType from a staticstub client ");
    System.out.println
    ("Client output for testComplexValueType : " +
    rfoo.getFirst().toString());
    System.out.println
    ("Client output for testComplexValueType : " +
    rfoo.getSecond());
    System.out.println
    ("Client output for testComplexValueType : " +
    rfoo.isThird());

    System.out.println
    ("\nOriginal unsigned long in “ + “
    ValueTypeWObjectMemberAObjectMemberArray : 129");
    System.out.println
    ("Modified unsigned long in + “ +
    ValueTypeWObjectMemberAObjectMemberArray  : "
    + result.getLongUnsignedInt());
}
```

One-Way Invocations From a Static Stub

Most remote calls use the synchronous request/response model. For example, when the `HelloClient` program calls the `sayHello` method, it waits until the method completes before resuming execution. At a lower level, on the client side the JAX-RPC implementation sends a SOAP request to the service and then waits. While the client is waiting, the service receives the request, processes it, and sends back a SOAP response. When the client receives the SOAP response it resumes execution. With the synchronous model, the client always waits for the response message, even if the return type of the method is `void`.

In contrast, with the one-way model the client sends a SOAP request to the service and then continues execution upon receiving an HTTP response. However, the client does not wait for a SOAP response. The service processes the request but does not send back a SOAP response. Because the client does not receive a SOAP response, it does not know whether or not the service completed the remote call. This limitation might be acceptable for some applications, for example, a monitoring application that frequently checks the status of a system.

The One-Way Service

To demonstrate the one-way model, in this example the `HelloClient` program invokes the `oneWayValueType` method on a separate service named `HelloWorldOneWayService`. This service was created along with `HelloWorldService` when you typed `ant build` and `ant deploy`. Although the two services share the same WAR file, `HelloWorldOneWayService` has a separate WSDL file (`HelloWorldOneWayService.wsdl`) and `wscompile` configuration files. The `wscompile` command-line options are the same for both services.

In the programming model for this example, the WSDL file already exists before you run `wscompile` to generate the server-side files. If you were to start with a service endpoint interface, you would use the `-f:useonewayoperations` option of `wscompile` when generating the WSDL file.

The `HelloWorldOneWayService` has the following endpoint address URL:

```
http://localhost:8080/jaxrpc-DoclitHelloService/oneway
```

The oneWayValueType Method

The `HelloClient` invokes `oneWayValueType` in the following code:

```
. . .
onewayStub = (HelloOneWayIF_Stub)
(new HelloWorldOneWayService_Impl().getHelloOneWayIFPort());
valueType = new ValueType(true,new Integer(23),"Test Data");
. . .
onewayStub.oneWayValueType(new OneWayValueType(valueType));
. . .
```

On the server side, `HelloOnewWayImpl` implements the method as follows:

```
package hello;

public class HelloOneWayImpl implements HelloOneWayIF {

    public void oneWayValueType(OneWayValueType evt) {
        ValueType vt = evt.getValueType_1();
       System.out.println("OneWay boolean value from client :"
           + vt.isBoolProperty());
       System.out.println("OneWay integer value from client :"
           + vt.getIntegerProperty().intValue());
       System.out.println("OneWay string value from client : "
           + vt.getStringProperty());
    }
}
```

The parameter of the method is `OneWayValueType`, a wrapper class for `ValueType`. The source code and class files, generated by `wscompile`, are in the `build/classes/client/` and `build/classes/server/` subdirectories.

Advanced Dynamic Proxy Example

This section shows how to build and run a dynamic proxy client that uses doc/literal and is WS-I compliant. If you are unfamiliar with dynamic proxies, you should first read the information in an earlier section, Dynamic Proxy Client Example (page 469).

Building and Running the Advanced Dynamic Proxy Example

Because this client invokes methods on the service described in the section Advanced Static Stub Example (page 488), you must deploy the service before proceeding. To build and run the dynamic proxy client, go to the *<INSTALL>*/jwstutorial13/examples/jaxrpc/advanced/dynamic/ directory and type the following:

```
ant build
ant run
```

The run task executes the ProxyHelloClient program, which should display the following lines:

```
Running echo String from a dii client.
Response is:  Duke says: Java is Everywhere!

Running SimpleValueType from a dii client using WSDL

Original ValueType is:
Echoing the boolean set in ValueType by client :true
Echoing the integer set in ValueType by client :23
Echoing the string set in ValueType by client  :Test Data

The response from the Server is:
Echoing the boolean set in ValueType by server :false
Echoing the integer set in ValueType by server :54
Echoing the string set in ValueType by server  :Server Entry :
Test Data
Running ChangeComplexValueType from a dii client.

Running ComplexValuType from a dii client using WSDL
Client output for testComplexValueType : 12345
Client output for testComplexValueType : 4.0
Client output for testComplexValueType : true

Original unsigned long in
ValueTypeWObjectMemberAObjectMemberArray : 129
Modified unsigned long in
ValueTypeWObjectMemberAObjectMemberArray  : 258

Running int[] from a dii client using WSDL
Client output for testSimpleIntArray   : true
```

Generating the Dynamic Proxy Client Files with wscompile

This example has the same programming model as that described in the section Client Programming Model for the Advanced Static Stub Example (page 492). However, in this dynamic proxy example the `ant build` task executes the `generate-service-interface` subtask, which runs `wscompile` as follows:

```
wscompile -keep -import -f:wsi conf/config-client.xml
```

This `wscompile` command generates the service endpoint interface and value types needed by the client. Generated wrapper classes are required because the client invokes operations that are literal (not encoded). For descriptions of the command options, see Table 12–1.

The `wscompile` command reads the service description from the WSDL file specified by the `conf/config-client.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
    <wsdl
        location="http://localhost:8080/jaxrpc-
DocumentLitHelloService/hello?WSDL"
        packageName="hello" />
</configuration>
```

The preceding `wscompile` command reads the same WSDL file as the command shown in the section Generating the Static Stub Client Files with wscompile (page 493). The two commands generate the same service endpoint interface and value types. For information about these generated files, see the section Service Endpoint Interface Generated by wscompile (page 494), as well as the subsequent sections on wrapper classes. Client developers should not write their own code as a substitute for the generated files. Instead, they should either rely on the files generated by `wscompile` or on files made available by the service developers.

The ProxyHelloClient Code

The source code for this client is in the *<INSTALL>*`/jwstutorial13/examples/jaxrpc/advanced/dynamic/src/client/` directory.

To set up the dynamic proxy, the `ProxyHelloClient` program performs these steps:

1. Sets program variables to match corresponding values in the WSDL file.

   ```
   private static String BODY_NAMESPACE_VALUE =
       “http://hello.org/wsdl”;
   String serviceName = “HelloWorldService”;
   String portName = “HelloIFPort”;
   ```

   Table 12–3 identifies the XML elements in the WSDL file that match the preceding variables

   .

Table 12–3 `ProxyHelloClient` Variables and Corresponding WSDL Elements

Program Variable	**WSDL Element**
`BODY_NAMESPACE_VALUE`	`<definitions>`
`serviceName`	`<service>`
`portName`	`<port>`

2. Sets the endpoint URL that denotes the location of the WSDL file that is deployed with the service.

   ```
   String UrlString =
   “http://localhost:8080/jaxrpc-DocumentLitHelloService/
   hello?WSDL”;
   ```

   The endpoint address and the WSDL must be supplied by the service developer or deployer.

3. Creates the service.

   ```
   URL helloWsdlUrl = new URL(UrlString);

   Service helloService =
       serviceFactory.createService(helloWsdlUrl,
       new QName(BODY_NAMESPACE_VALUE, serviceName));
   ```

   The `createService` method invocation has two parameters: a `QName` (qualified name) representing the name of the service and a URL designating the location of the WSDL file. At runtime, the service will be configured with information fetched from the WSDL file.

4. Creates the dynamic proxy.

```
HelloIF proxy = null;
...
proxy = (HelloIF) helloService.getPort(
    new QName(BODY_NAMESPACE_VALUE, portName),
    hello.HelloIF.class);
```

The `wscompile` tool generated `HelloIF.class`, the service endpoint interface. Because the client code refers to `HelloIF.class`, you must run `wscompile` before compiling the client.

To invoke the `sayHello` method, the program does the following:

1. Instantiates the parameter of the remote call.

```
SayHello request = new SayHello(" Duke says: " +
    "Java is Everywhere!");
```

The `SayHello` and `SayHelloResponse` wrapper classes are also generated by `wscompile`.

2. Invokes `sayHello` on the dynamic proxy:

```
SayHelloResponse response = proxy.sayHello(request);
```

3. Gets and prints the string returned by `sayHello`.

```
System.out.println("Response is: " + response.getResult());
```

Advanced DII Client Example

This section demonstrates two DII clients:

- `DIIHelloClient` - relies on the deployed WSDL during runtime
- `DIINoWSDLHelloClient` - does not rely on a WSDL during runtime

During development, `wscompile` reads the deployed WSDL and generates the value types needed by `DIIHelloClient` and `DIINoWSDLHelloClient`. Both clients use doc/literal, are WS-I compliant, and invoke methods on the service deployed in the section Advanced Static Stub Example (page 488). For an introduction to DII, see the section Dynamic Invocation Interface (DII) Client Example (page 472).

Building and Running the Advanced DII Example

To build the clients, go to the `<INSTALL>/jwstutorial13/examples/jaxrpc/advanced/dii/` directory and type the following:

```
ant build
```

To run `DIIHelloClient`, type:

```
ant run
```

The `DIIHelloClient` program should display the following lines:

```
Running echo String from a dii client.
Response is:  Duke says: Java is Everywhere!

Running SimpleValueType from a dii client using WSDL

Original ValueType is:
Echoing the boolean set in ValueType by client :true
Echoing the integer set in ValueType by client :23
Echoing the string set in ValueType by client  :Test Data

The response from the Server is:
Echoing the boolean set in ValueType by server :false
Echoing the integer set in ValueType by server :54
Echoing the string set in ValueType by server  :Server Entry :
Test Data
Running ChangeComplexValueType from a dii client.

Running ComplexValuType from a dii client using WSDL
Client output for testComplexValueType : 12345
Client output for testComplexValueType : 4.0
Client output for testComplexValueType : true

Original unsigned long in
ValueTypeWObjectMemberAObjectMemberArray : 129
Modified unsigned long in
ValueTypeWObjectMemberAObjectMemberArray  : 258

Running int[] from a dii client using WSDL
Client output for testSimpleIntArray   : true
```

To run `DIINoWSDLHelloClient`, type:

```
ant run-no-wsdl
```

The `DIINoWSDLHelloClient` programs displays the same lines as `DIIHelloClient`, except for the following:

```
Running ComplexValueType from a dii client not using WSDL
```

Generating the DII Client Files with wscompile

The `ant build` task executes the `generate-service-interface` subtask, which runs this `wscompile` command:

```
wscompile -keep -import -f:wsi conf/config-client.xml
```

The `wscompile` command and `config-client.xml` file of this example are identical to those used in Generating the Dynamic Proxy Client Files with wscompile (page 507). Unlike the dynamic proxy client, these DII clients do not need the service endpoint interface (`HelloIF`) generated by `wscompile`. However, like all clients that invoke operations that are literal (not encoded), the DII clients need the wrapper classes that `wscompile` generates. For information on wrapper classes, see the section Advanced Static Stub Example (page 488). Client developers should not write their own code as a substitute for the generated files.

The DIIHelloClient Code

At runtime, the `DIIHelloClient` program uses information in the deployed WSDL file to automatically configure the `Call` object on which it invokes remote methods. The items automatically configured include parameters, return types, and the target endpoint address.

The source code for DIIHelloClient is in the `<INSTALL>/jwstutorial13/examples/jaxrpc/advanced/dii/src/client/` directory.

To set up the service, `DIIHelloClient` does the following:

1. Initializes program variables.

```
private static String BODY_NAMESPACE_VALUE =
    “http://hello.org/wsdl”;
private static String ENCODING_STYLE_PROPERTY =
    “javax.xml.rpc.encodingstyle.namespace.uri”;
```

```
String UrlString =
“http://localhost:8080/jaxrpc-DocumentLitHelloService/
hello?WSDL”;
```

```
String serviceName = “HelloWorldService”;
String portName = “HelloIFPort”;
Service service = null;
QName port = null;
```

The `UrlString` object denotes the location of the WSDL file that will be accessed by the client at runtime. The WSDL file specifies the target endpoint address in the `<soap:address>` element. See Table 12–3 for more information on the WSDL elements that match the other initialized variables.

2. Creates the service.

```
service =
    factory.createService(new java.net.URL(UrlString),
    new QName(BODY_NAMESPACE_VALUE, serviceName));
```

Note that the first parameter of `createService` designates the URL of the WSDL file. At runtime, the service will be configured with information fetched from the WSDL.

3. Creates a `QName` object that represents the service port.

```
port = new QName(BODY_NAMESPACE_VALUE, portName);
```

To set up the `Call` object and make the remote invocation, `DIIHelloClient` performs these steps:

1. Creates the `Call` object.

```
QName operation = new QName(“sayHello”);
Call call = service.createCall(port, operation);
```

In this step, the client creates a `Call` object for the port (`HelloIFPort`) and the remote method (`sayHello`). In a later step, the client will invoke the `sayHello` method on the `Call` object.

2. Sets properties on the `Call` object.

```
call.setProperty(Call.SOAPACTION_USE_PROPERTY
    new Boolean(true));
call.setProperty(Call.SOAPACTION_URI_PROPERTY, ““);

call.setProperty(ENCODING_STYLE_PROPERTY, ““);
call.setProperty(Call.OPERATION_STYLE_PROPERTY,
     “document”);
```

The previous two lines of code set the encoding/operation style to doc/literal. To learn more about these properties, refer to the SOAP and WSDL documents listed in Further Information (page 516).

3. Instantiates and loads the parameter request.

```
SayHello request = new SayHello(" Duke says: ",
    "Java is Everywhere!");
Object[] params = {request};
```

`SayHello` is a wrapper class generated by `wscompile`.

4. Invokes the remote `sayHello` method.

```
SayHelloResponse response =
    (SayHelloResponse) call.invoke(params);
```

Like the `SayHello` parameter class, the `SayHelloResponse` class is generated by `wscompile`.

5. Gets and prints the string returned by `sayHello`.

```
System.out.println("Response is: " + response.getResult());
```

The DIINoWSDLHelloClient Code

Unlike the `DIIWSDLHelloClient` program described in the preceding section, the `DIINoWSDLHelloClient` of this section does not configure the `Call` object at runtime with information retrieved from the WSDL file. Because `DIINoWSDLHelloClient` does not access the WSDL file, it must programatically configure the `Call` object with the following:

- target endpoint address
- operation
- parameters
- return type

To prepare the service, the `DIINoWSDLHelloClient` program performs these steps:

1. Initializes program variables.

```
private static String BODY_NAMESPACE_VALUE =
    "http://hello.org/wsdl";
private static String TYPE_NAMESPACE_VALUE =
    "http://hello.org/types";
private static String ENCODING_STYLE_PROPERTY =
    "javax.xml.rpc.encodingstyle.namespace.uri";
```

```
String UrlString =
"http://localhost:8080/jaxrpc-DocumentLitHelloService/
hello?WSDL";

String serviceName = "HelloWorldService";
String portName = "HelloIFPort";
Service service = null;
QName port = null;
```

`DIINoWSDLHelloClient` initializes these variables the same way as `DII-WSDLHelloClient`, with the exception of `UrlString`. In DIINoWSDL-HelloClient, `UrlString` designates the target endpoint address, not the WSDL file location.

2. Creates the service.

```
ServiceFactory factory = ServiceFactory.newInstance();
service = factory.createService(new QName(serviceName));
```

This `createService` method invocation does not include a parameter for the WSDL file.

3. Creates a `QName` object that represents the service port.

```
port = new QName(portName);
```

To configure the `Call` object and invoke the `sayHello` method, `DIINoWSDLHelloClient` does the following:

1. Creates the Call object.

```
QName operation =
    new QName(BODY_NAMESPACE_VALUE, "sayHello");
Call call = service.createCall(port, operation);
```

2. Sets the endpoint address of the target service.

```
call.setTargetEndpointAddress(endpoint);
```

In the WSDL file, the endpoint address is in the `<soap:address>` element.

3. Sets properties on the `Call` object.

```
call.setProperty(Call.SOAPACTION_USE_PROPERTY,
    new Boolean(true));
call.setProperty(Call.SOAPACTION_URI_PROPERTY, "");
call.setProperty(ENCODING_STYLE_PROPERTY, "");
call.setProperty(Call.OPERATION_STYLE_PROPERTY,
    "document");
```

4. Creates the `QName` object that names the request parameter.

```
QName REQUEST_QNAME =
    new QName(TYPE_NAMESPACE_VALUE, "sayHello");
```

The `TYPE_NAMESPACE_VALUE` string designates the namespace for the data types defined within the WSDL file. The WSDL file specifies the type information for `sayHello` under the `<types>` element, in the `<message>` element named `HelloIF_sayHello`.

5. Adds a parameter to the `Call` object.

```
call.addParameter("parameters", REQUEST_QNAME,
    hello.SayHello.class, ParameterMode.IN);
```

The `parameters` argument matches the name of the WSDL `<part>` subelement that is contained in the `<message>` element named `HelloIF_sayHello`. For DII clients that don't access the WSDL file at runtime, `addParameter` must specify the parameter class, in this case `hello.SayHello.class`.

6. Specifies the `QName` object for the call response.

```
QName RESPONSE_QNAME =
    new QName(TYPE_NAMESPACE_VALUE, "sayHelloResponse");
```

7. Sets the return type on the `Call` object.

```
call.setReturnType(RESPONSE_QNAME,
     hello.SayHelloResponse.class);
```

Note that `setReturnType` specifies `SayHelloResponse.class` for the return type.

8. Instantiates and loads the parameter request.

```
SayHello request = new SayHello(" Duke says: ",
    "Java is Everywhere!");
Object[] params = {request};
```

9. Invokes the remote `sayHello` method.

```
SayHelloResponse response =
    (SayHelloResponse) call.invoke(params);
```

10. Gets and prints the string returned by `sayHello`.

```
System.out.println("Response is: " + response.getResult());
```

Further Information

For more information about JAX-RPC and related technologies, refer to the following:

- Java API for XML-based RPC 1.1 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`
- WS-I Basic Profile 1.0
  `http://www.ws-i.org`

13

SOAP with Attachments API for Java

SOAP with Attachments API for Java (SAAJ) is used mainly for the SOAP messaging that goes on behind the scenes in JAX-RPC and JAXR implementations. Secondarily, it is an API that developers can use when they choose to write SOAP messaging applications directly rather than using JAX-RPC. The SAAJ API allows you to do XML messaging from the Java platform: By simply making method calls using the SAAJ API, you can create, send, and consume XML messages over the Internet. This chapter will help you learn how to use the SAAJ API.

The SAAJ API conforms to the Simple Object Access Protocol (SOAP) 1.1 specification and the SOAP with Attachments specification. The SOAP with Attachments API for Java (SAAJ) 1.2 specification defines the `javax.xml.soap` package, 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 Objects, page 520.)

Note: The `javax.xml.messaging` package, defined in the Java API for XML Messaging (JAXM) 1.1 specification, is not part of the Java Web Services Developer

Pack and is not discussed in this chapter. The JAXM API is available as a separate download from `http://java.sun.com/xml/jaxm/`.

This chapter starts with an overview of messages and connections, which gives some of the conceptual background behind the SAAJ API to help you understand why certain things are done the way they are. Next the tutorial shows you how to use the basic SAAJ API, giving examples and explanations of the more commonly used features. The code examples in the last part of the tutorial show you how to build an application, and the case study in The Coffee Break Application (page 1001) includes SAAJ code for both sending and consuming a SOAP message.

Overview of SAAJ

This overview presents a high level view of how SAAJ 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 SAAJ from two perspectives:

- Messages
- Connections

Messages

SAAJ messages follow SOAP standards, which prescribe the format for messages and also specify some things that are required, optional, or not allowed. With the SAAJ API, you can create XML messages that conform to the SOAP 1.1 and WS-I Basic Profile 1.0 specifications simply by making Java API calls.

The Structure of an XML Document

Note: For more complete information on XML documents, see Chapters 5 and 6.

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 to be in every SOAP message.

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, the `SOAPPart` class to represent the SOAP part, the `SOAPEnvelope` interface to represent the SOAP envelope, and so on. Figure 13–1 illustrates the structure of a SOAP message with no attachments.

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. The `SOAPBody` object, which always follows the `SOAPHeader` object if there is one, provides a simple way to send information intended for the ultimate recipient. For example, if there is a `SOAPFault` object (see Using SOAP Faults, page 543), it must be in the `SOAPBody` object.

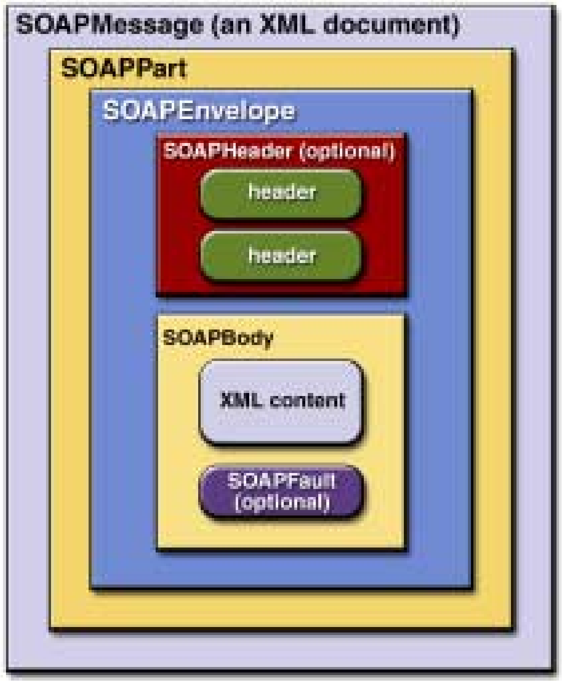


Figure 13–1 `SOAPMessage` Object with No Attachments

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 a binary file, your message must have an attachment part for it. Note that an attachment part can contain any kind of content, so it can contain data in XML format as well. Figure 13–2 shows the high-level structure of a SOAP message that has two attachments.

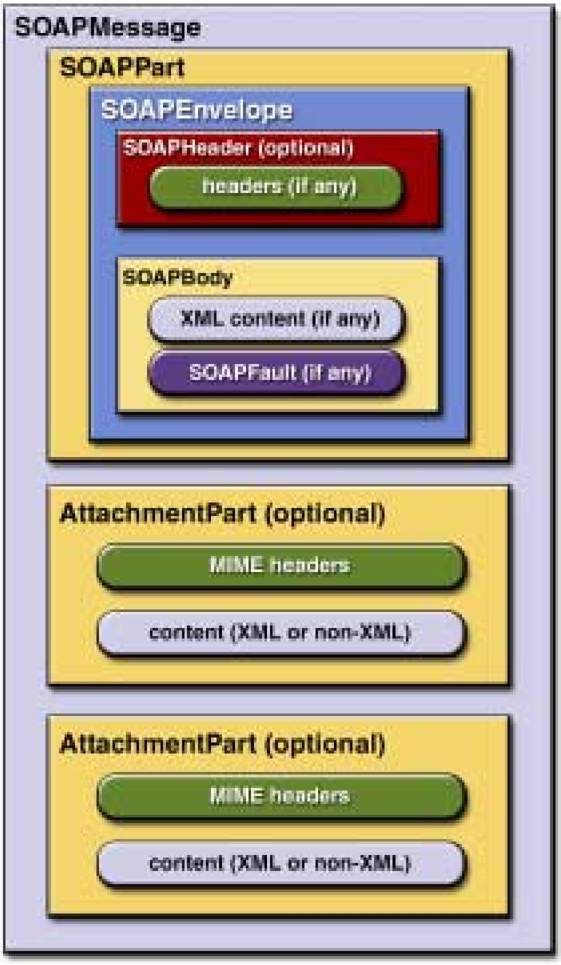


Figure 13–2 `SOAPMessage` Object with Two `AttachmentPart` Objects

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.

If a `SOAPMessage` object has one or more attachments, each `AttachmentPart` object must have 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 are optional but 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.

SAAJ and DOM

At SAAJ 1.2, the SAAJ APIs extend their counterparts in the `org.w3c.dom` package:

- The `Node` interface extends the `org.w3c.dom.Node` interface.
- The `SOAPElement` interface extends both the `Node` interface and the `org.w3c.dom.Element` interface.
- The `SOAPPart` class implements the `org.w3c.dom.Document` interface.
- The `Text` interface extends the `org.w3c.dom.Text` interface.

Moreover, the `SOAPPart` of a `SOAPMessage` is also a DOM Level 2 `Document`, and can be manipulated as such by applications, tools and libraries that use DOM. See Chapter 8 for details about DOM. See Adding Content to the SOAP-Part Object (page 532) and Adding a Document to the SOAP Body (page 534) for details on how to use DOM documents with the SAAJ API.

Connections

All SOAP messages are sent and received over a connection. With the SAAJ API, the connection is represented by a `SOAPConnection` object, which goes from the sender directly to its destination. This kind of connection is called a *point-to-point* connection because it goes from one endpoint to another endpoint. Messages sent using the SAAJ API are called *request-response messages*. They are sent over a `SOAPConnection` object with the method `call`, which sends a message (a request) and then blocks until it receives the reply (a response).

SOAPConnection Objects

The following code fragment creates the `SOAPConnection` object `connection`, and then, after creating and populating the message, uses `connection` to send the message. As stated previously, all messages sent over a `SOAPConnection` object are sent with the method `call`, which both sends the message and blocks until it receives the response. Thus, the return value for the method `call` is the `SOAPMessage` object that is the response to the message that was sent. The parameter `request` is the message being sent; `endpoint` represents where it is being sent.

```
SOAPConnectionFactory factory =
  SOAPConnectionFactory.newInstance();
SOAPConnection connection = factory.createConnection();

. . .// create a request message and give it content

java.net.URL endpoint =
  new URL("http://fabulous.com/gizmo/order");
SOAPMessage response = connection.call(request, endpoint);
```

Note that the second argument to the method `call`, which identifies where the message is being sent, can be a `String` object or a `URL` object. Thus, the last two lines of code from the preceding example could also have been the following:

```
String endpoint = "http://fabulous.com/gizmo/order";
SOAPMessage response = connection.call(request, endpoint);
```

A Web service implemented for request-response messaging must return a response to any message it receives. The response is a `SOAPMessage` object, just as the request is a `SOAPMessage` object. When the request 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.

Now that you have some background on SOAP messages and SOAP connections, in the next section you will see how to use the SAAJ API.

Tutorial

This tutorial will walk you through how to use the SAAJ API. First, it covers the basics of creating and sending a simple SOAP message. Then you will learn more details about adding content to messages, including how to create SOAP faults and attributes. Finally, you will learn how to send a message and retrieve the content of the response. After going through this tutorial, you will know how to perform the following tasks:

- Creating and Sending a Simple Message
- Adding Content to the Header
- Adding Content to the SOAP Body
- Adding Content to the SOAPPart Object
- Adding a Document to the SOAP Body
- Manipulating Message Content Using SAAJ or DOM APIs
- Adding Attachments
- Adding Attributes
- Using SOAP Faults

In the section Code Examples (page 548), you will see the code fragments from earlier parts of the tutorial in runnable applications, which you can test yourself. To see how the SAAJ API can be used in server code, see the SAAJ part of the case study (SAAJ Distributor Service, page 1011), which shows an example of both the client and server code for a Web service application.

A SAAJ client can send request-response messages to Web services that are implemented to do request-response messaging. This section demonstrates how you can do this.

Creating and Sending a Simple Message

This section covers the basics of creating and sending a simple message and retrieving the content of the response. It includes the following topics:

- Creating a Message
- Parts of a Message
- Accessing Elements of a Message
- Adding Content to the Body
- Getting a SOAPConnection Object
- Sending a Message
- Getting the Content of a Message

Creating a Message

The first step is to create a message, which you do using a `MessageFactory` object. The SAAJ API provides a default implementation of the `MessageFactory` class, thus making it easy to get an instance. The following code fragment illustrates getting an instance of the 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`.

Parts of a Message

A `SOAPMessage` object is required to have certain elements, and, as stated previously, 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 is optional and may be deleted if it is not needed. However, if there is one, it must precede the `SOAPBody` object. 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 Using SOAP Faults (page 543) 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. There are two ways to do this. The `SOAPMessage` object `message`, created in the previous code fragment, is the place to start.

The first way to access the parts of the message is to work your way through the structure of the message. The message contains a `SOAPPart` object, so you use the `getSOAPPart` method of `message` to retrieve it:

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

Next you can use the `getEnvelope` method of `soapPart` to retrieve the `SOAPEnvelope` object that it contains.

```
SOAPEnvelope envelope = soapPart.getEnvelope();
```

You can now use the `getHeader` and `getBody` methods of `envelope` to retrieve its empty `SOAPHeader` and `SOAPBody` objects.

```
SOAPHeader header = envelope.getHeader();
SOAPBody body = envelope.getBody();
```

The second way to access the parts of the message is to retrieve the message header and body directly, without retrieving the `SOAPPart` or `SOAPEnvelope`. To do so, use the `getSOAPHeader` and `getSOAPBody` methods of `SOAPMessage`:

```
SOAPHeader header = message.getSOAPHeader();
SOAPBody body = message.getSOAPBody();
```

This example of a SAAJ client does not use a SOAP header, so you can delete it. (You will see more about headers later.) 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 so that it is uniquely identified.

One way to create `Name` objects is by using `SOAPEnvelope` methods, so you can use the variable `envelope` from the previous code fragment to create the `Name` object for your new element. Another way to create `Name` objects is to use `SOAPFactory` methods, which are useful if you do not have access to the `SOAPEnvelope`.

Note: The `SOAPFactory` class also lets you create XML elements when you are not creating an entire message or do not have access to a complete `SOAPMessage` object. For example, JAX-RPC implementations often work with XML fragments rather than complete `SOAPMessage` objects. Consequently, they do not have access to a `SOAPEnvelope` object, which makes using a `SOAPFactory` object to create `Name` objects very useful. 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 of SOAP Faults (page 543) and Creating and Populating a SOAPFault Object (page 545).

`Name` objects associated with `SOAPBodyElement` or `SOAPHeaderElement` 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 `message`, uses a `SOAPFactory` to create a `Name` object for the element to be added, and adds a new `SOAPBodyElement` object to `body`.

```
SOAPBody body = message.getSOAPBody();
SOAPFactory soapFactory = SOAPFactory.newInstance();
Name bodyName = soapFactory.createName("GetLastTradePrice",
  "m", "http://wombat.ztrade.com");
SOAPBodyElement bodyElement = body.addBodyElement(bodyName);
```

At this point, `body` contains a `SOAPBodyElement` object identified by the `Name` object `bodyName`, but there is still no content in `bodyElement`. 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 because child elements inherit the prefix and URI from the parent element.

```
Name name = soapFactory.createName("symbol");
SOAPElement symbol = bodyElement.addChildElement(name);
symbol.addTextNode("SUNW");
```

You might recall that the headers and content in a `SOAPPart` object must be in XML format. The SAAJ 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 SAAJ 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.

SAAJ code:

```
SOAPMessage message = messageFactory.createMessage();
SOAPHeader header = message.getSOAPHeader();
SOAPBody body = message.getSOAPBody();
```

XML it produces:

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body>
    . . .
  </SOAP-ENV:Body>
</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 is an example of an attribute for the SOAP envelope element. Because a SOAP Envelope element always contains this attribute with this value, a SOAPMessage object comes with it automatically included. xmlns stands for “XML namespace,” and its value is the URI of the namespace associated with Envelope.

The next line is an empty SOAP header. We could remove it by calling header.detachNode after the getSOAPHeader call.

The next two lines mark the beginning and end of the SOAP body, represented in SAAJ by a SOAPBody object. The next step is to add content to the body.

SAAJ code:

```
Name bodyName = soapFactory.createName("GetLastTradePrice",
  "m", "http://wombat.ztrade.com");
SOAPBodyElement bodyElement = body.addBodyElement(bodyName);
```

XML it produces:

```
<m:GetLastTradePrice
 xmlns:m="http://wombat.ztrade.com">
 . . . .
</m:GetLastTradePrice>
```

These lines are what the SOAPBodyElement bodyElement in your code represents. GetLastTradePrice"is its local name, m is its namespace prefix, and http://wombat.ztrade.com is its namespace URI.

SAAJ code:

```
Name name = soapFactory.createName("symbol");
SOAPElement symbol = bodyElement.addChildElement(name);
symbol.addTextNode("SUNW");
```

XML it produces:

```
<symbol>SUNW</symbol>
```

The String “SUNW” is the text node for the element <symbol>. This String object is the message content that your recipient, the stock quote service, receives.

Getting a SOAPConnection Object

The SAAJ API is focused primarily on creating messages. Once you have a message, you can send it using various mechanisms (JMS or JAXM, for example). The SAAJ API does, however, provide a simple mechanism for request-response messaging.

To send a message, a SAAJ client may use a `SOAPConnection` object. A `SOAPConnection` object 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 `SOAPConnectionFactory` class with a default implementation. You can get an instance of this implementation with the following line of code.

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

Now you can use `soapConnectionFactory` to create a `SOAPConnection` object.

```
SOAPConnection connection =
  soapConnectionFactory.createConnection();
```

You will use `connection` to send the message that you created.

Sending a Message

A SAAJ client calls the `SOAPConnection` method `call` on a `SOAPConnection` object to send a message. The `call` 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 = connection.call(message, endpoint);
```

The content of the message you sent is 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.

```
connection.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: Either you use the `Message` object to get the `SOAPBody` object, or you access the `SOAPBody` object through the `SOAPPart` and `SOAPEnvelope` objects.

Then you access the `SOAPBody` object's `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 `SOAPPart` 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 `getChildElements` on `soapBody`. Passing `bodyName` to `getChildElements` 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`.

```
SOAPBody soapBody = response.getSOAPBody();
java.util.Iterator iterator =
  soapBody.getChildElements(bodyName);
SOAPBodyElement bodyElement =
  (SOAPBodyElement)iterator.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 (iterator.hasNext()) {
  SOAPBodyElement bodyElement =
    (SOAPBodyElement)iterator.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 very basic request-response message and get the content from the response. The next sections provide more detail on adding content to messages.

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 can create using the message's `SOAPEnvelope` object or a `SOAPFactory` object.

For example, suppose you want to add a conformance claim header to the message to state that your message conforms to the WS-I Basic Profile.

The following code fragment retrieves the `SOAPHeader` object from `message` and adds a new `SOAPHeaderElement` object to it. This `SOAPHeaderElement` object contains the correct qualified name and attribute for a WS-I conformance claim header.

```
SOAPHeader header = message.getSOAPHeader();
Name headerName = soapFactory.createName("Claim",
  "wsi", "http://ws-i.org/schemas/conformanceClaim/");
SOAPHeaderElement headerElement =
  header.addHeaderElement(headerName);
headerElement.addAttribute(soapFactory.createName(
  "conformsTo"), "http://ws-i.org/profiles/basic1.0/");
```

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`.

A conformance claim header has no content. This code produces the following XML header:

```
<SOAP-ENV:Header>
  <wsi:Claim conformsTo="http://ws-i.org/profiles/basic1.0/"
   xmlns:wsi="http://ws-i.org/schemas/conformanceClaim/"/>
</SOAP-ENV:Header>
```

For more information about creating SOAP messages that conform to WS-I, see the Messaging section of the WS-I Basic Profile.

For a different kind of header, you might want to add content to `headerElement`. The following line of code uses the method `addTextNode` to do this.

```
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 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 following example shows adding multiple `SOAPElement` objects and adding text to each of them. The code first creates the `SOAPBodyElement` object `purchaseLineItems`, which has a fully qualified name 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 name, but child elements added to it, such as `SOAPElement` objects, may have `Name` objects with only the local name.

```
SOAPBody body = soapFactory.getSOAPBody();
Name bodyName = soapFactory.createName("PurchaseLineItems",
  "PO", "http://sonata.fruitsgalore.com");
SOAPBodyElement purchaseLineItems =
  body.addBodyElement(bodyName);
```

```
Name childName = soapFactory.createName("Order");
SOAPElement order =
  purchaseLineItems.addChildElement(childName);

childName = soapFactory.createName("Product");
SOAPElement product = order.addChildElement(childName);
product.addTextNode("Apple");

childName = soapFactory.createName("Price");
SOAPElement price = order.addChildElement(childName);
price.addTextNode("1.56");

childName = soapFactory.createName("Order");
SOAPElement order2 =
  purchaseLineItems.addChildElement(childName);

childName = soapFactory.createName("Product");
SOAPElement product2 = order2.addChildElement(childName);
product2.addTextNode("Peach");

childName = soapFactory.createName("Price");
SOAPElement price2 = order2.addChildElement(childName);
price2.addTextNode("1.48");
```

The SAAJ 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, SAAJ 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 the JAXP API to build a DOMSource object that is passed to the SOAPPart.setContent method. The first three lines of code get a DocumentBuilderFactory object and use it to create the DocumentBuilder object builder. Because SOAP messages use namespaces, you should set the NamespaceAware property for the factory to true. Then builder parses the content file to produce a Document object.

```
DocumentBuilderFactory dbFactory =
  DocumentBuilderFactory.newInstance();
dbFactory.setNamespaceAware(true);
DocumentBuilder builder = dbFactory.newDocumentBuilder();
Document document =
  builder.parse("file:///music/order/soap.xml");
DOMSource domSource = new DOMSource(document);
```

The following two lines of code access the SOAPPart object (using the SOAPMessage object message) and set the new Document 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);
```

The XML file you use to set the content of the SOAPPart object must include Envelope and Body elements, like this:

```
<SOAP-ENV:Envelope
xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
  ...
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

You will see other ways to add content to a message in the sections Adding a Document to the SOAP Body (page 534) and Adding Attachments (page 535).

Adding a Document to the SOAP Body

In addition to setting the content of the entire SOAP message to that of a `DOMSource` object, you can add a DOM document directly to the body of the message. This capability means that you do not have to create a `javax.xml.transform.Source` object. After you parse the document, you can add it directly to the message body:

```
SOAPBody body = message.getSOAPBody();
SOAPBodyElement docElement = body.addDocument(document);
```

Manipulating Message Content Using SAAJ or DOM APIs

Because SAAJ nodes and elements implement the DOM `Node` and `Element` interfaces, you have many options for adding or changing message content:

- Use only DOM APIs
- Use only SAAJ APIs
- Use SAAJ APIs and then switch to using DOM APIs
- Use DOM APIs and then switch to using SAAJ APIs

The first three of these cause no problems. Once you have created a message, whether or not you have imported its content from another document, you can start adding or changing nodes using either SAAJ or DOM APIs.

But if you use DOM APIs and then switch to using SAAJ APIs to manipulate the document, any references to objects within the tree that were obtained using DOM APIs are no longer valid. If you must use SAAJ APIs after using DOM APIs, you should set all of your DOM typed references to null, because they can become invalid. For more information about the exact cases in which references become invalid, see the SAAJ API documentation.

The basic rule is that you can continue manipulating the message content using SAAJ APIs as long as you want to, but once you start manipulating it using DOM, you should not use SAAJ APIs after that.

Adding Attachments

An `AttachmentPart` object can contain any type of content, including XML. 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, SAAJ 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`. This method takes two parameters, the first being a Java `Object` containing the content and the second being a `String` giving the content type. The Java Object may be a `String`, a stream, a `javax.xml.transform.Source` object, or a `javax.activation.DataHandler` object. The Java `Object` being added in the following code fragment is a `String`, which is plain text, so the second argument must 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 it 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 `attachment` is now part of `message`.

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.

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:

```
URL url = new URL("http://greatproducts.com/gizmos/img.jpg");
DataHandler dataHandler = new DataHandler(url);
AttachmentPart attachment =
  message.createAttachmentPart(dataHandler);
attachment.setContentId("attached_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. The `SOAPMessage` class provides two versions of the method `getAttachments` for retrieving its `AttachmentPart` objects. When it is given no argument, the method `SOAPMessage.getAttachments` returns a `java.util.Iterator` object over all the `AttachmentPart` objects in a message. When `getAttachments` is given a `MimeHeaders` object, which is a list of MIME headers, it returns an iterator over the `AttachmentPart` objects that have a header that matches one of the headers in the list. The following code uses the `getAttachments` method that takes no arguments and thus retrieves all of the `AttachmentPart` objects in the `SOAPMessage` object `message`. Then it prints out the content ID, content type, and content of each `AttachmentPart` object.

```
java.util.Iterator iterator = message.getAttachments();
while (iterator.hasNext()) {
  AttachmentPart attachment =
    (AttachmentPart)iterator.next();
  String id = attachment.getContentId();
  String type = attachment.getContentType();
  System.out.print("Attachment " + id +
    " has content type " + type);
  if (type == "text/plain") {
    Object content = attachment.getContent();
    System.out.println("Attachment " +
      "contains:\n" + content);
  }
}
```

Adding Attributes

An XML element may have one or more attributes that give information about that element. An attribute consists of a name for the attribute followed immediately by an equals sign (=) and its value.

The `SOAPElement` interface provides methods for adding an attribute, for getting the value of an attribute, and for removing an attribute. For example, in the following code fragment, the attribute named `id` is added to the `SOAPElement` object person. Because person is a SOAPElement object rather than a SOAPBodyElement object or SOAPHeaderElement object, it is legal for its Name object to contain only a local name.

```
Name attributeName = envelope.createName("id");
person.addAttribute(attributeName, "Person7");
```

These lines of code will generate the first line in the following XML fragment.

```
<person id="Person7">
  ...
</person>
```

The following line of code retrieves the value of the attribute whose name is id.

```
String attributeValue =
  person.getAttributeValue(attributeName);
```

If you had added two or more attributes to person, the previous line of code would have returned only the value for the attribute named id. If you wanted to retrieve the values for all of the attributes for person, you would use the method getAllAttributes, which returns an iterator over all of the values. The following lines of code retrieve and print out each value on a separate line until there are no more attribute values. Note that the method Iterator.next returns a Java Object, which is cast to a Name object so that it can be assigned to the Name object attributeName. (The examples in DOMExample.java and DomSrcExample.java (page 558) use code similar to this.)

```
Iterator iterator = person.getAllAttributes();
while (iterator.hasNext()){
  Name attributeName = (Name) iterator.next();
  System.out.println("Attribute name is " +
    attributeName.getQualifiedName());
  System.out.println("Attribute value is " +
    element.getAttributeValue(attributeName));
}
```

The following line of code removes the attribute named id from person. The variable successful will be true if the attribute was removed successfully.

```
boolean successful = person.removeAttribute(attributeName);
```

In this section you saw how to add, retrieve, and remove attributes. This information is general in that it applies to any element. The next section discusses attributes that may be added only to header elements.

Header Attributes

Attributes that appear in a `SOAPHeaderElement` object determine how a recipient processes a message. You can think of header attributes as offering a way to extend a message, giving information about such things as authentication, transaction management, payment, and so on. A header attribute refines the meaning of the header, while the header refines the meaning of the message contained in the SOAP Body.

The SOAP 1.1 specification defines two attributes that can appear only in `SOAPHeaderElement` objects: `actor` and `mustUnderstand`. The next two sections discuss these attributes.

See HeaderExample.java (page 557) for an example that uses the code shown in this section.

The Actor Attribute

The attribute `actor` is optional, but if it is used, it must appear in a `SOAPHeaderElement` object. Its purpose is to indicate the recipient of a header element. The default actor is the message's ultimate recipient; that is, if no actor attribute is supplied, the message goes directly to the ultimate recipient.

An actor is an application that can both receive SOAP messages and forward them to the next actor. The ability to specify one or more actors as intermediate recipients makes it possible to route a message to multiple recipients and to supply header information that applies specifically to each of the recipients.

For example, suppose that a message is an incoming purchase order. Its `SOAPHeader` object might have `SOAPHeaderElement` objects with actor attributes that route the message to applications that function as the order desk, the shipping desk, the confirmation desk, and the billing department. Each of these applications will take the appropriate action, remove the `SOAPHeaderElement` objects relevant to it, and send the message on to the next actor.

Note: Although the SAAJ API provides the API for adding these attributes, it does not supply the API for processing them. For example, the actor attribute requires that there be an implementation such as a messaging provider service to route the message from one actor to the next.

An actor is identified by its URI. For example, the following line of code, in which `orderHeader` is a `SOAPHeaderElement` object, sets the actor to the given URI.

```
orderHeader.setActor("http://gizmos.com/orders");
```

Additional actors may be set in their own `SOAPHeaderElement` objects. The following code fragment first uses the `SOAPMessage` object `message` to get its `SOAPHeader` object `header`. Then `header` creates four `SOAPHeaderElement` objects, each of which sets its `actor` attribute.

```
SOAPHeader header = message.getSOAPHeader();
SOAPFactory soapFactory = SOAPFactory.newInstance();

String nameSpace = "ns";
String nameSpaceURI = "http://gizmos.com/NSURI";

Name order = soapFactory.createName("orderDesk",
  nameSpace, nameSpaceURI);
SOAPHeaderElement orderHeader =
  header.addHeaderElement(order);
orderHeader.setActor("http://gizmos.com/orders");

Name shipping =
  soapFactory.createName("shippingDesk",
     nameSpace, nameSpaceURI);
SOAPHeaderElement shippingHeader =
  header.addHeaderElement(shipping);
shippingHeader.setActor("http://gizmos.com/shipping");

Name confirmation =
  soapFactory.createName("confirmationDesk",
     nameSpace, nameSpaceURI);
SOAPHeaderElement confirmationHeader =
  header.addHeaderElement(confirmation);
confirmationHeader.setActor(
  "http://gizmos.com/confirmations");

Name billing = soapFactory.createName("billingDesk",
  nameSpace, nameSpaceURI);
SOAPHeaderElement billingHeader =
  header.addHeaderElement(billing);
billingHeader.setActor("http://gizmos.com/billing");
```

The SOAPHeader interface provides two methods that return a java.util.Iterator object over all of the SOAPHeaderElement objects with an actor that matches the specified actor. The first method, examineHeaderElements, returns an iterator over all of the elements with the specified actor.

```
java.util.Iterator headerElements =
  header.examineHeaderElements("http://gizmos.com/orders");
```

The second method, extractHeaderElements, not only returns an iterator over all of the SOAPHeaderElement objects with the specified actor attribute but also detaches them from the SOAPHeader object. So, for example, after the order desk application has done its work, it would call extractHeaderElements to remove all of the SOAPHeaderElement objects that applied to it.

```
java.util.Iterator headerElements =
  header.extractHeaderElements("http://gizmos.com/orders");
```

Each SOAPHeaderElement object may have only one actor attribute, but the same actor may be an attribute for multiple SOAPHeaderElement objects.

Two additional SOAPHeader methods, examineAllHeaderElements and extractAllHeaderElements, allow you to examine or extract all the header elements, whether or not they have an actor attribute. For example, you could use the following code to display the values of all the header elements:

```
Iterator allHeaders =
  header.examineAllHeaderElements();
while (allHeaders.hasNext()) {
  SOAPHeaderElement headerElement =
    (SOAPHeaderElement)allHeaders.next();
  Name headerName =
    headerElement.getElementName();
  System.out.println("\nHeader name is " +
    headerName.getQualifiedName());
  System.out.println("Actor is " +
    headerElement.getActor());
}
```

The mustUnderstand Attribute

The other attribute that must be added only to a SOAPHeaderElement object is mustUnderstand. This attribute says whether or not the recipient (indicated by the actor attribute) is required to process a header entry. When the value of the mustUnderstand attribute is true, the actor must understand the semantics of the header entry and must process it correctly to those semantics. If the value is `false`, processing the header entry is optional. A `SOAPHeaderElement` object with no `mustUnderstand` attribute is equivalent to one with a `mustUnderstand` attribute whose value is `false`.

The `mustUnderstand` attribute is used to call attention to the fact that the semantics in an element are different from the semantics in its parent or peer elements. This allows for robust evolution, ensuring that the change in semantics will not be silently ignored by those who may not fully understand it.

If the actor for a header that has a `mustUnderstand` attribute set to `true` cannot process the header, it must send a SOAP fault back to the sender. (See the section Using SOAP Faults, page 543 for information.) The actor must not change state or cause any side-effects, so that to an outside observer, it appears that the fault was sent before any header processing was done.

The following code fragment creates a `SOAPHeader` object with a `SOAPHeaderElement` object that has a `mustUnderstand` attribute.

```
SOAPHeader header = message.getSOAPHeader();

Name name = soapFactory.createName("Transaction", "t",
  "http://gizmos.com/orders");

SOAPHeaderElement transaction = header.addHeaderElement(name);
transaction.setMustUnderstand(true);
transaction.addTextNode("5");
```

This code produces the following XML:

```
<SOAP-ENV:Header>
  <t:Transaction
     xmlns:t="http://gizmos.com/orders"
     SOAP-ENV:mustUnderstand="1">
    5
  </t:Transaction>
</SOAP-ENV:Header>
```

You can use the `getMustUnderstand` method to retrieve the value of the `MustUnderstand` attribute. For example, you could add the following to the code fragment at the end of the previous section:

```
System.out.println("MustUnderstand is " +
  headerElement.getMustUnderstand());
```

Using SOAP Faults

In this section, you will see how to use the API for creating and accessing a SOAP Fault element in an XML message.

Overview of SOAP Faults

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. Further, if there is a SOAP Fault element in the SOAP Body, there can be no other elements in the SOAP Body. This means that when you add a SOAP Fault element, you have effectively completed the construction of 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 kinds of 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 SAAJ 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.

If you are a client using the SAAJ API and are 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.

Another example of who might send a SOAP fault is an intermediate recipient, or actor. As stated in the section Adding Attributes, page 537, an actor that cannot process a header that has a `mustUnderstand` attribute with a value of `true` must return a SOAP fault to the sender.

A `SOAPFault` object contains the following elements:

- A **fault code** — always required

  The fault code must be a fully qualified name, which means that it must contain a prefix followed by a local name. The SOAP 1.1 specification defines a set of fault code local name values in section 4.4.1, which a developer may extend to cover other problems. The default fault code local names defined in the specification relate to the SAAJ 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 `true`, 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 The Actor Attribute, page 539.
- 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 `SOAPFault` 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.

```
SOAPBody body = message.getSOAPBody();
```

With the `SOAPBody` object `body` in hand, you can use it to create a `SOAPFault` object. The following line of code both creates a `SOAPFault` object and adds it to `body`.

```
SOAPFault fault = body.addFault();
```

The `SOAPFault` interface provides convenience methods that create an element, add the new element to the `SOAPFault` object, and add a text node all in one operation. For example, in the following lines of code, the method `setFaultCode` creates a `faultcode` element, adds it to `fault`, and adds a `Text` node with the value `"SOAP-ENV:Server"` by specifying a default prefix and the namespace URI for a SOAP envelope.

```
Name faultName =
  soapFactory.createName("Server",
    "", SOAPConstants.URI_NS_SOAP_ENVELOPE);
fault.setFaultCode(faultName);
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();

Name faultName = soapFactory.createName("Client",
  "", SOAPConstants.URI_NS_SOAP_ENVELOPE);
fault.setFaultCode(faultName);
fault.setFaultString("Message does not have necessary info");

Detail detail = fault.addDetail();

Name entryName = soapFactory.createName("order",
  "PO", "http://gizmos.com/orders/");
DetailEntry entry = detail.addDetailEntry(entryName);
entry.addTextNode("Quantity element does not have a value");

Name entryName2 = soapFactory.createName("confirmation",
  "PO", "http://gizmos.com/confirm");
DetailEntry entry2 = detail.addDetailEntry(entryName2);
entry2.addTextNode("Incomplete address: no zip code");
```

See SOAPFaultTest.java (page 564) for an example that uses code like that shown in this section.

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, `newMessage` 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 = newMessage.getSOAPBody();
if ( body.hasFault() ) {
  SOAPFault newFault = body.getFault();
  Name code = newFault.getFaultCodeAsName();
  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.getQualifiedName());
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 `entries`, 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` objects as long as there are any.

```
Detail newDetail = newFault.getDetail();
if ( newDetail != null) {
    Iterator entries = newDetail.getDetailEntries();
    while ( entries.hasNext() ) {
        DetailEntry newEntry =
            (DetailEntry)entries.next();
        String value = newEntry.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 contents. 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.

See SOAPFaultTest.java (page 564) for an example that uses code like that shown in this section.

Code Examples

The first part of this tutorial used code fragments to walk you through the fundamentals of using the SAAJ 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 run the programs `MyUddiPing.java`, `HeaderExample.java`, `DOMExample.java`, `Attachments.java`, and `SOAPFaultTest.java`.

You do not have to start Tomcat in order to run these examples.

Request.java

The class `Request.java` puts together the code fragments used in Tutorial (page 521) 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.

```
import javax.xml.soap.*;
import java.util.*;
import java.net.URL;

public class Request {
  public static void main(String[] args){
     try {
        SOAPConnectionFactory soapConnectionFactory =
           SOAPConnectionFactory.newInstance();
        SOAPConnection connection =
           soapConnectionFactory.createConnection();
        SOAPFactory soapFactory =
           SOAPFactory.newInstance();

        MessageFactory factory =
           MessageFactory.newInstance();
        SOAPMessage message = factory.createMessage();

        SOAPHeader header = message.getSOAPHeader();
        SOAPBody body = message.getSOAPBody();
        header.detachNode();
```

```
          Name bodyName = soapFactory.createName(
            "GetLastTradePrice", "m",
            "http://wombats.ztrade.com");
          SOAPBodyElement bodyElement =
            body.addBodyElement(bodyName);

          Name name = soapFactory.createName("symbol");
          SOAPElement symbol =
            bodyElement.addChildElement(name);
          symbol.addTextNode("SUNW");

          URL endpoint = new URL
            ("http://wombat.ztrade.com/quotes");
          SOAPMessage response =
            connection.call(message, endpoint);

          connection.close();

          SOAPBody soapBody = response.getSOAPBody();

          Iterator iterator =
            soapBody.getChildElements(bodyName);
          SOAPBodyElement bodyElement =
            (SOAPBodyElement)iterator.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` would have to be a valid existing URI, which is not true in this case. However, the application in the next section is one that you can run.

MyUddiPing.java

The program `MyUddiPing.java` is another example of a SAAJ client application. It sends a request to a Universal Description, Discovery and Integration (UDDI) service and gets back the response. A 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 MyUddiPing 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, MyUddiPing demonstrates a request being sent and a response being received.

Setting Up

The myuddiping example is in the following directory:

```
<INSTALL>/jwstutorial13/examples/saaj/myuddiping/
```

Note: *<INSTALL>* is the directory where you installed the tutorial bundle.

In the myuddiping directory, you will find two files and the src directory. The src directory contains one source file, MyUddiPing.java.

The file uddi.properties contains the URL of the destination (a UDDI test registry) and the proxy host and proxy port of the sender. Edit this file to supply the correct proxy host and proxy port if you access the Internet from behind a firewall. If you are not sure what the values for these are, consult your system administrator or another person with that information. The typical value of the proxy port is 8080.

The file build.xml is the Ant build file for this example. It includes the file *<INSTALL>*/jwstutorial13/examples/saaj/common/targets.xml, which contains a set of targets common to all the SAAJ examples.

The prepare target creates a directory named build. To invoke the prepare target, 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
```

Examining MyUddiPing

We will go through the file `MyUddiPing.java` a few lines at a time, concentrating on the last section. This is the part of the application that accesses only the content you want from the XML message returned by the UDDI registry.

The first few lines of code import the packages used in the application.

```
import javax.xml.soap.*;
import java.net.*;
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. The usage message mentions only one argument; the other is supplied by the `build.xml` target.

```
public class MyUddiPing {
  public static void main(String[] args) {
    try {
      if (args.length != 2) {
        System.err.println("Argument required: " +
          "-Dbusiness-name=<name>");
        System.exit(1);
      }
```

The following lines create a `java.util.Properties` object 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 enum = myprops.propertyNames();
      while (enum.hasMoreElements()) {
        String s = (String)enum.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 soapConnectionFactory =
   SOAPConnectionFactory.newInstance();
SOAPConnection connection =
   soapConnectionFactory.createConnection();
MessageFactory messageFactory =
   MessageFactory.newInstance();

SOAPMessage message =
   messageFactory.createMessage();
```

The next lines of code retrieve the SOAPHeader and SOAPBody objects from the message and remove the header.

```
SOAPHeader header = message.getSOAPHeader();
SOAPBody body = message.getSOAPBody();
header.detachNode();
```

The following lines of code create the UDDI find_business message. The first line gets a SOAPFactory instance that we will use to create names. The next line adds the SOAPBodyElement with a fully qualified name, including the required namespace for a UDDI version 2 message. The next lines add two attributes to the new element: the required attribute generic, with the UDDI version number 2.0, and the optional attribute maxRows, with the value 100. Then the code adds a child element with the Name object name and adds text to the element with the method addTextNode. The text added is the business name you will supply at the command line when you run the application.

```
SOAPFactory soapFactory =
   SOAPFactory.newInstance();
SOAPBodyElement findBusiness =
   body.addBodyElement(soapFactory.createName(
      "find_business", "",
      "urn:uddi-org:api_v2"));
findBusiness.addAttribute(soapFactory.createName(
   "generic"), "2.0");
findBusiness.addAttribute(soapFactory.createName(
   "maxRows"), "100");
SOAPElement businessName =
   findBusiness.addChildElement(
      soapFactory.createName("name"));
businessName.addTextNode(args[1]);
```

The next 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.

```
message.saveChanges();
```

The following lines display the message that will be sent:

```
System.out.println("\n--- Request Message ---\n");
message.writeTo(System.out);
```

The next line of code creates the `java.net.URL` object that represents the destination for this message. It gets the value of the property named `URL` from the system property file.

```
URL endpoint = new URL(
  System.getProperties().getProperty("URL"));
```

Next the message `message` is sent to the destination that `endpoint` represents, which is the UDDI test registry. The `call` method will block until it gets a `SOAPMessage` object back, at which point it returns the reply.

```
SOAPMessage reply =
  connection.call(message, endpoint);
```

In the next lines of code, the first line prints out a line giving the URL of the sender (the test registry), and the others display the returned message.

```
System.out.println("\n\nReceived reply from: " +
  endpoint);
System.out.println("\n---- Reply Message ----\n");
reply.writeTo(System.out);
```

The returned message is the complete SOAP message, an XML document, as it looks when it comes over the wire. It is a `businessList` that follows the format specified in `http://uddi.org/pubs/DataStructure-V2.03-Published-20020719.htm#_Toc25130802`.

As interesting as it is to see the XML that is actually transmitted, the XML document format does not make it easy to see the text that is the message's content. To remedy this, the last part of `MyUddiPing.java` contains code that prints out just the text content of the response, making it much easier to see the information you want.

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.

```
SOAPBody replyBody = reply.getSOAPBody();
```

Next the code displays a message describing the content:

```
System.out.println("\n\nContent extracted from " +
  "the reply message:\n");
```

To display the content of the message, the code uses the known format of the reply message. First it gets all the reply body's child elements named `businessList`:

```
Iterator businessListIterator =
  replyBody.getChildElements(
    soapFactory.createName("businessList",
      "", "urn:uddi-org:api_v2"));
```

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.

An immediate child of a `SOAPBody` object is a `SOAPBodyElement` object.

We know that the reply can contain only one `businessList` element, so the code then retrieves this one element by calling the iterator's `next` method. Note that the method `Iterator.next` returns an `Object`, which has to be cast to the specific kind of object you are retrieving. Thus, the result of calling `businessListIterator.next` is cast to a `SOAPBodyElement` object:

```
SOAPBodyElement businessList =
  (SOAPBodyElement)businessListIterator.next();
```

The next element in the hierarchy is a single `businessInfos` element, so the code retrieves this element the same way it retrieved the `businessList`. Children of `SOAPBodyElement` objects and all child elements from there down are `SOAPElement` objects.

```
Iterator businessInfosIterator =
   businessList.getChildElements(
      soapFactory.createName("businessInfos",
         "", "urn:uddi-org:api_v2"));

SOAPElement businessInfos =
   (SOAPElement)businessInfosIterator.next();
```

The `businessInfos` element contains zero or more `businessInfo` elements. If the query returned no businesses, the code prints a message saying that none were found. If the query returned businesses, however, the code extracts the name and optional description by retrieving the child elements with those names. 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. Accordingly, the loop ends when there are no more child elements to retrieve.

```
Iterator businessInfoIterator =
   businessInfos.getChildElements(
      soapFactory.createName("businessInfo",
         "", "urn:uddi-org:api_v2"));

if (! businessInfoIterator.hasNext()) {
   System.out.println("No businesses found " +
      "matching the name '" + args[1] +
      "'.");
} else {
   while (businessInfoIterator.hasNext()) {
      SOAPElement businessInfo = (SOAPElement)
         businessInfoIterator.next();
      // Extract name and description from the
      // businessInfo
      Iterator nameIterator =
         businessInfo.getChildElements(
            soapFactory.createName("name",
               "", "urn:uddi-org:api_v2"));
      while (nameIterator.hasNext()) {
         businessName =
            (SOAPElement)nameIterator.next();
         System.out.println("Company name: " +
            businessName.getValue());
      }
      Iterator descriptionIterator =
         businessInfo.getChildElements(
```

```
            soapFactory.createName(
               "description", "",
               "urn:uddi-org:api_v2"));
        while (descriptionIterator.hasNext()) {
            SOAPElement businessDescription =
               (SOAPElement)
               descriptionIterator.next();
            System.out.println("Description: " +
               businessDescription.getValue());
        }
        System.out.println("");
    }
```

Running MyUddiPing

Make sure you have edited the `uddi.properties` file and compiled `MyUddiPing.java` as described in Setting Up (page 550).

With the code compiled, you are ready to run `MyUddiPing`. The `run` target takes two arguments, but you need to supply only one of them. The first argument is the file `uddi.properties`, which is supplied by a property set in `build.xml`. The other 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 any value set for that property in the `build.xml` file.

Use the following command to run the example:

```
ant run -Dbusiness-name=food
```

Output similar to the following will appear after the full XML message:

```
Content extracted from the reply message:

Company name: Food
Description: Test Food

Company name: Food Manufacturing

Company name: foodCompanyA
Description: It is a food company sells biscuit
```

If you want to run `MyUddiPing` again, you may want 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
```

HeaderExample.java

The example `HeaderExample.java`, based on the code fragments in the section Adding Attributes (page 537), creates a message with several headers. It then retrieves the contents of the headers and prints them out. You will find the code for `HeaderExample` in the following directory:

```
<INSTALL>/jwstutorial13/examples/saaj/headers/src/
```

Running HeaderExample

To run `HeaderExample`, you use the file `build.xml` that is in the directory `<INSTALL>/jwstutorial13/examples/saaj/headers/`.

To run `HeaderExample`, use the following command:

```
ant run
```

This command executes the `prepare`, `build`, and `run` targets in the `build.xml` and `targets.xml` files.

When you run `HeaderExample`, you will see output similar to the following:

```
----- Request Message ----

<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header>
<ns:orderDesk SOAP-ENV:actor="http://gizmos.com/orders"
xmlns:ns="http://gizmos.com/NSURI"/>
<ns:shippingDesk SOAP-ENV:actor="http://gizmos.com/shipping"
xmlns:ns="http://gizmos.com/NSURI"/>
<ns:confirmationDesk
SOAP-ENV:actor="http://gizmos.com/confirmations"
xmlns:ns="http://gizmos.com/NSURI"/>
<ns:billingDesk SOAP-ENV:actor="http://gizmos.com/billing"
xmlns:ns="http://gizmos.com/NSURI"/>
<t:Transaction SOAP-ENV:mustUnderstand="1"
```

```
xmlns:t="http://gizmos.com/orders">5</t:Transaction>
</SOAP-ENV:Header><SOAP-ENV:Body/></SOAP-ENV:Envelope>
Header name is ns:orderDesk
Actor is http://gizmos.com/orders
MustUnderstand is false

Header name is ns:shippingDesk
Actor is http://gizmos.com/shipping
MustUnderstand is false

Header name is ns:confirmationDesk
Actor is http://gizmos.com/confirmations
MustUnderstand is false

Header name is ns:billingDesk
Actor is http://gizmos.com/billing
MustUnderstand is false

Header name is t:Transaction
Actor is null
MustUnderstand is true
```

DOMExample.java and DomSrcExample.java

The examples `DOMExample.java` and `DOMSrcExample.java` show how to add a DOM document to a message and then to traverse its contents. They show two different ways to do this:

- `DomExample.java` creates a DOM document and adds it to the body of a message.
- `DomSrcExample.java` creates the document, uses it to create a `DOMSource` object, and then sets the `DOMSource` object as the content of the message's SOAP part.

You will find the code for `DOMExample` and `DOMSrcExample` in the following directory:

```
<INSTALL>/jwstutorial13/examples/saaj/dom/src/
```

Examining DOMExample

`DOMExample` first creates a DOM document by parsing an XML document, almost exactly like the JAXP example `DomEcho01.java` in the directory `<INSTALL>/jwstutorial13/examples/jaxp/dom/samples/`. The file it parses is one that you specify on the command line.

```
static Document document;
...
  DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
  factory.setNamespaceAware(true);
  try {
    DocumentBuilder builder =
      factory.newDocumentBuilder();
    document = builder.parse( new File(args[0]) );
    ...
```

Next, the example creates a SOAP message in the usual way. Then it adds the document to the message body:

```
SOAPBodyElement docElement =
  body.addDocument(document);
```

This example does not change the content of the message. Instead, it displays the message content and then uses a recursive method, `getContents`, to traverse the element tree using SAAJ APIs and display the message contents in a readable form.

```
public void getContents(Iterator iterator,
  String indent) {

  while (iterator.hasNext()) {
    SOAPElement element =
      (SOAPElement)iterator.next();
    Name name = element.getElementName();
    System.out.println(indent + "Name is " +
      name.getQualifiedName());
    String content = element.getValue();
    if (content != null) {
      System.out.println(indent + "Content is " +
        content);
    }
    Iterator attrs = element.getAllAttributes();
    while (attrs.hasNext()){
      Name attrName = (Name)attrs.next();
```

```
            System.out.println(indent +
               " Attribute name is " +
               attrName.getQualifiedName());
            System.out.println(indent +
               " Attribute value is " +
               element.getAttributeValue(attrName));
         }
         if (content == null) {
            Iterator iter2 = element.getChildElements();
            getContents(iter2, indent + " ");
         }
      }
   }
```

Examining DOMSrcExample

`DOMSrcExample` differs from `DomExample` in only a few ways. First, after it parses the document, it uses the document to create a `DOMSource` object. This code is the same as that of `DomExample` except for the last line:

```
static DOMSource domSource;
...
try {
   DocumentBuilder builder =
      factory.newDocumentBuilder();
   document = builder.parse( new File(args[0]) );
   domSource = new DOMSource(document);
   ...
```

Then, after `DomSrcExample` creates the message, it does not get the header and body and add the document to the body, as `DOMExample` does. Instead, it gets the SOAP part and sets the `DOMSource` object as its content:

```
// Create a message
SOAPMessage message =
  messageFactory.createMessage();

// Get the SOAP part and set its content to domSource
SOAPPart soapPart = message.getSOAPPart();
soapPart.setContent(domSource);
```

The example then uses the `getContents` method to obtain the contents of both the header (if it exists) and the body of the message.

The most important difference between these two examples is the kind of document you can use to create the message. Because `DOMExample` adds the document to the body of the SOAP message, you can use any valid XML file to create the document. But because `DOMSrcExample` makes the document the entire content of the message, the document must already be in the form of a valid SOAP message, not just any XML document.

Running DOMExample and DOMSrcExample

To run `DOMExample` and `DOMSrcExample`, you use the file `build.xml` that is in the directory `<INSTALL>/jwstutorial13/examples/saaj/dom/`. This directory also contains several sample XML files you can use:

- `domsrc1.xml`, an example that has a SOAP header (the contents of the `HeaderExample` output) and the body of a UDDI query
- `domsrc2.xml`, an example of a reply to a UDDI query (specifically, some sample output from the `MyUddiPing` example), but with spaces added for readability
- `uddimsg.xml`, similar to `domsrc2.xml` except that it is only the body of the message and contains no spaces
- `slide.xml`, similar to the `slideSample01.xml` file in `<INSTALL>/jwstutorial13/examples/jaxp/dom/samples/`

To run `DOMExample`, use a command like the following:

```
ant run-dom -Dxml-file=uddimsg.xml
```

After running `DOMExample`, you will see output something like the following:

```
Running DOMExample.
Name is businessList
Attribute name is generic
Attribute value is 2.0
Attribute name is operator
Attribute value is www.ibm.com/services/uddi
Attribute name is truncated
Attribute value is false
Attribute name is xmlns
Attribute value is urn:uddi-org:api_v2
...
```

To run `DOMSrcExample`, use a command like the following:

```
ant run-domsrc -Dxml-file=domsrc2.xml
```

When you run `DOMSrcExample`, you will see output that begins like the following:

```
run-domsrc:
  Running DOMSrcExample.
  Body contents:
  Content is:

  Name is businessList
   Attribute name is generic
   Attribute value is 2.0
   Attribute name is operator
   Attribute value is www.ibm.com/services/uddi
   Attribute name is truncated
   Attribute value is false
   Attribute name is xmlns
   Attribute value is urn:uddi-org:api_v2
   ...
```

If you run `DOMSrcExample` with the file `uddimsg.xml` or `slide.xml`, you will see runtime errors.

Attachments.java

The example `Attachments.java`, based on the code fragments in the sections Creating an AttachmentPart Object and Adding Content (page 535) and Accessing an AttachmentPart Object (page 537), creates a message with a text attachment and an image attachment. It then retrieves the contents of the attachments and prints out the contents of the text attachment. You will find the code for `Attachments` in the following directory:

```
<INSTALL>/jwstutorial13/examples/saaj/attachments/src/
```

`Attachments` first creates a message in the usual way. It then creates an `AttachmentPart` for the text attachment:

```
AttachmentPart attachment1 = message.createAttachmentPart();
```

After it reads input from a file into a string named `stringContent`, it sets the content of the attachment to the value of the string and the type to text-plain, and also sets a content ID.

```
attachment1.setContent(stringContent, "text/plain");
attachment1.setContentId("attached_text");
```

It then adds the attachment to the message:

```
message.addAttachmentPart(attachment1);
```

The example uses a `javax.activation.DataHandler` object to hold a reference to the graphic that constitutes the second attachment. It creates this attachment using the form of the `createAttachmentPart` method that takes a `DataHandler` argument.

```
// Create attachment part for image
URL url = new URL("file:///./xml-pic.jpg");
DataHandler dataHandler = new DataHandler(url);
AttachmentPart attachment2 =
  message.createAttachmentPart(dataHandler);
attachment2.setContentId("attached_image");

message.addAttachmentPart(attachment2);
```

The example then retrieves the attachments from the message. It displays the `contentId` and `contentType` attributes of each attachment and the contents of the text attachment. Because the example does not display the contents of the graphic, the URL does not have to refer to an actual graphic, although in this case it does.

Running Attachments

To run `Attachments`, you use the file `build.xml` that is in the directory `<INSTALL>/jwstutorial13/examples/saaj/attachments/`.

To run `Attachments`, use the following command:

```
ant run -Dfile=path_name
```

Specify any text file as the *`path_name`* argument. The `attachments` directory contains a file named `addr.txt` that you can use:

```
ant run -Dfile=addr.txt
```

When you run `Attachments` using this command line, you will see output like the following:

```
Running Attachments.
Attachment attached_text has content type text/plain
Attachment contains:
Update address for Sunny Skies, Inc., to
10 Upbeat Street
Pleasant Grove, CA 95439

Attachment attached_image has content type image/jpeg
```

SOAPFaultTest.java

The example `SOAPFaultTest.java`, based on the code fragments in the sections Creating and Populating a SOAPFault Object (page 545) and Retrieving Fault Information (page 546), 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:

```
<INSTALL>/jwstutorial13/examples/saaj/fault/src/
```

Running SOAPFaultTest

To run `SOAPFaultTest`, you use the file `build.xml` that is in the directory `<INSTALL>/jwstutorial13/examples/saaj/fault/`.

To run `SOAPFaultTest`, use the following command:

```
ant run
```

When you run `SOAPFaultTest`, you will see output like the following (line breaks have been inserted in the message for readability):

```
Here is what the XML message looks like:

<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header/><SOAP-ENV:Body>
<SOAP-ENV:Fault><faultcode>SOAP-ENV:Client</faultcode>
<faultstring>Message does not have necessary info</faultstring>
<faultactor>http://gizmos.com/order</faultactor>
<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>
</detail></SOAP-ENV:Fault>
</SOAP-ENV:Body></SOAP-ENV:Envelope>

SOAP fault contains:
   Fault code = SOAP-ENV:Client
   Local name = Client
   Namespace prefix = SOAP-ENV, bound to
http://schemas.xmlsoap.org/soap/envelope/
   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
```

Further Information

For more information about SAAJ, SOAP, and WS-I, see the following:

- SAAJ 1.2 specification, available from
  `http://java.sun.com/xml/downloads/saaj.html`
- SAAJ website:
  `http://java.sun.com/xml/saaj/`
- Simple Object Access Protocol (SOAP) 1.1 Specification:
  `http://www.w3.org/TR/SOAP/`
- WS-I Basic Profile:
  `http://www.ws-i.org/Profiles/Basic/2003-01/BasicProfile-1.0-WGAD.html`
- JAXM website:
  `http://java.sun.com/xml/jaxm/`

14

Java API for XML Registries

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 1095) 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 provides a brief overview of JAXR. It 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 public 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. The Registry Server includes a database based on the native XML database Xindice, which is part of the Apache XML project. This database provides the repository for registry data. The Registry Server does not support messages defined in the UDDI Version 2.0 Replication Specification. See The Java WSDP Registry Server (page 1089) for more information.

Note: If you use the Java WSDP Registry Server to test JAXR applications that you develop using the J2EE 1.4 Application Server, make sure that in your `PATH` you place the J2EE 1.4 Application Server `bin` directories before the Java WSDP `bin` directories.

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 J2EE Application Server 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 14–1 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.

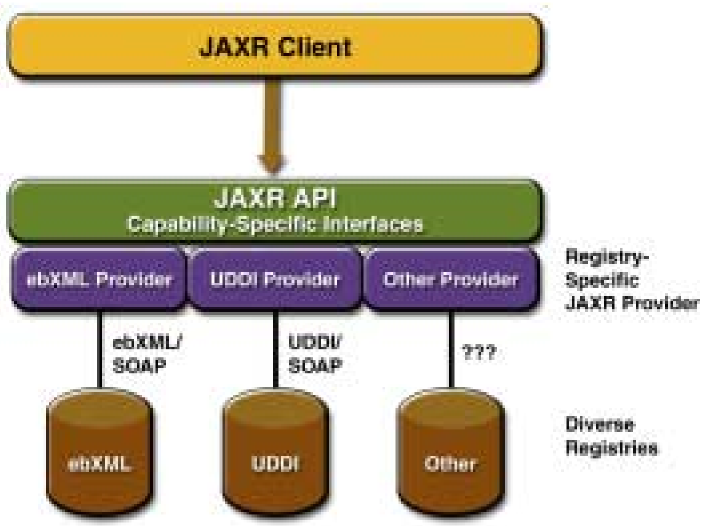


Figure 14–1 JAXR Architecture

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. It covers the following topics:

- Establishing a Connection
- Querying a Registry
- Managing Registry Data
- Using Taxonomies in JAXR Clients

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 595). The examples are in the directory `<INSTALL>/jwstutorial13/examples/jaxr/`. (`<INSTALL>` is the directory where you installed the tutorial bundle.) Each example directory has a `build.xml` file that refers to a `targets.xml` file and a `build.properties` file in the directory `<INSTALL>/jwstutorial13/examples/jaxr/common/`.

Establishing a Connection

The first task a JAXR client must complete is to establish a connection to a registry. Establishing a connection involves the following tasks:

- Preliminaries: Getting Access to a Registry
- Creating or Looking Up a Connection Factory
- Creating a Connection
- Setting Connection Properties
- Obtaining and Using a RegistryService Object

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://test.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.

You do not have to register with the Java WSDP Registry Server in order to add or update data. You can use the default user name and password, `testuser` and `testuser`.

Note: The JAXR API has been tested with the Microsoft and IBM registries and with the Java WSDP Registry Server, 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/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 14–1 and Table 14–2 list and describe these properties.

Table 14–1 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

Table 14–1 Standard JAXR Connection Properties

Property Name and Description	Data Type	Default 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 593) for an example	String	None

Table 14–2 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	String	None
`com.sun.xml.registry.http.proxyPort` Specifies the HTTP proxy port to be used for accessing external registries; usually 8080	String	None
`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

Table 14–2 Implementation-Specific JAXR Connection Properties

Property Name and Description	Data Type	Default Value
`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

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/publishapi");
ConnectionFactory factory = (ConnectionFactory)
   context.lookup("java:comp/env/eis/JAXR");
factory.setProperties(props);
connection = factory.createConnection();
```

- The `postalAddressScheme` and `useCache` properties may be set in a `<sysproperty>` tag in a `build.xml` file for the Ant tool. For example:

```
<sysproperty key="useCache" 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 590).

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
- Finding Organizations by Classification
- Finding Services and ServiceBindings

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`.)

```
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.

Managing registry data involves the following tasks:

- Getting Authorization from the Registry
- Creating an Organization
- Adding Classifications
- Adding Services and Service Bindings to an Organization
- Publishing an Organization
- Publishing a Specification Concept
- Removing Data from the Registry

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 publishing 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 577). 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 1914");
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);
```

Publishing 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);
  }
}
```

Publishing a Specification Concept

A service binding may have a technical specification that describes how to access the service. An example of such a specification is a WSDL document. To publish the location of a service's specification (if the specification is a WSDL document), you create a Concept object and then add the URL of the WSDL document to the Concept object as an ExternalLink object. The following code fragment shows how to create a concept for the WSDL document associated with the simple web service example in Creating a Web Service with JAX-RPC (page 459). First, you call the `createConcept` method to create a concept named HelloConcept. After setting the description of the concept, you create an external link to the URL of the Hello service's WSDL document, then add the external link to the concept.

```
Concept specConcept =
  blcm.createConcept(null, "HelloConcept", "");
InternationalString s =
  blcm.createInternationalString(
    "Concept for Hello Service");
specConcept.setDescription(s);
ExternalLink wsdlLink =
  blcm.createExternalLink(
    "http://localhost:8080/hello-jaxrpc/hello?WSDL",
    "Hello WSDL document");
specConcept.addExternalLink(wsdlLink);
```

Next, you classify the `Concept` object as a WSDL document. To do this for a UDDI registry, you search the registry for the well-known classification scheme `uddi-org:types`. (The UDDI term for a classification scheme is `tModel`.) Then you create a classification using the name and value `wsdlSpec`. Finally, you add the classification to the concept.

```
String schemeName = "uddi-org:types";
ClassificationScheme uddiOrgTypes =
  bqm.findClassificationSchemeByName(null, schemeName);

Classification wsdlSpecClassification =
    blcm.createClassification(uddiOrgTypes,
    "wsdlSpec", "wsdlSpec");
specConcept.addClassification(wsdlSpecClassification);
```

Finally, you save the concept using the `saveConcepts` method, similarly to the way you save an organization:

```
Collection concepts = new ArrayList();
concepts.add(specConcept);
BulkResponse concResponse = blcm.saveConcepts(concepts);
```

Once you have published the concept, you normally add the concept for the WSDL document to a service binding. To do this, you could retrieve the key for the concept from the response returned by the saveConcepts method, using a code sequence very similar to that of finding the key for a saved organization.

```
String conceptKeyId = null;
Collection concExceptions = concResponse.getExceptions();
javax.xml.registry.infomodel.Key concKey = null;
if (concExceptions == null) {
  System.out.println("WSDL Specification Concept saved");

  Collection keys = concResponse.getCollection();
  Iterator keyIter = keys.iterator();
  if (keyIter.hasNext()) {
    concKey =
      (javax.xml.registry.infomodel.Key) keyIter.next();
    conceptKeyId = concKey.getId();
    System.out.println("Concept key is " + id);
  }
}
```

Then you could call the getRegistryObject method to retrieve the concept from the registry:

```
Concept specConcept =
  (Concept) bqm.getRegistryObject(conceptKeyId,
    LifeCycleManager.CONCEPT);
```

Next, you create a SpecificationLink object for the service binding and set the concept as the value of its SpecificationObject:

```
SpecificationLink specLink =
  blcm.createSpecificationLink();
specLink.setSpecificationObject(specConcept);
binding.addSpecificationLink(specLink);
```

Now when you publish the organization with its service and service bindings, you have also published a link to the WSDL document, so that the organization can be found in queries like those described in Finding Organizations by Classification (page 579).

If the concept was published by someone else and you don't have access to the key, you can find it using its name and classification. The code would look very similar to the code used to search for a WSDL document in Finding Organizations by Classification (page 579), except that you would also create a collection of name patterns and include that in your search. For example:

```
// Define name pattern
Collection namePatterns = new ArrayList();
namePatterns.add("HelloConcept");

BulkResponse br = bqm.findConcepts(null, namePatterns,
  classifications, null, null);
```

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`, `deleteConcepts`, 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 concepts, 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>/jaxr/lib/jaxr-impl.jar` file, but you can find copies of them in the directory `<JWSDP_HOME>/jaxr/docs/taxonomies/`. This directory also includes files that define the well-known taxonomies that the implementation of JAXR in the Java WSDP uses: `naics.xml`, `iso3166.xml`, and `unspsc.xml`.

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
   -Dcom.sun.xml.registry.userTaxonomyFilena-
   mes=c:\mydir\xxx.xml|c:\mydir\xxx2.xml
   ```

You can use a `<sysproperty>` tag to set this property in a `build.xml` file for a client program. Or, in your program, you can set the property as follows:

```
System.setProperty
("com.sun.xml.registry.userTaxonomyFilenames",
   "c:\mydir\xxx.xml|c:\mydir\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 named `MyPostalAddressScheme`, which you published to a registry with the UUID `uuid:f7922839-f1f7-9228-c97d-ce0b4594736c`.

```
<JAXRClassificationScheme id="uuid:f7922839-f1f7-9228-c97d-
ce0b4594736c" name="MyPostalAddressScheme">
```

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:f7922839-f1f7-9228-c97d-ce0b4594736c");
```

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," +
  "uuid:f7922839-f1f7-9228-c97d-
ce0b4594736c/StreetAddressNumber|" +
  "urn:uuid:PostalAddressAttributes/Street," +
  "urn:uuid:f7922839-f1f7-9228-c97d-
ce0b4594736c/StreetAddress|" +
  "urn:uuid:PostalAddressAttributes/City," +
  "urn:uuid:f7922839-f1f7-9228-c97d-ce0b4594736c/City|" +
  "urn:uuid:PostalAddressAttributes/State," +
  "urn:uuid:f7922839-f1f7-9228-c97d-ce0b4594736c/State|" +
  "urn:uuid:PostalAddressAttributes/PostalCode," +
  "urn:uuid:f7922839-f1f7-9228-c97d-ce0b4594736c/ZipCode|" +
  "urn:uuid:PostalAddressAttributes/Country," +
  "urn:uuid:f7922839-f1f7-9228-c97d-ce0b4594736c/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 Java WSDP Registry Server for queries and updates; you can specify any other UDDI version 2 registry.

The client examples, in the `<INSTALL>/jwstutorial13/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

- `JAXRPublishConcept.java` shows how to publish a concept for a WSDL document
- `JAXRPublishHelloOrg.java` shows how to publish an organization with a service binding that refers to a WSDL document
- `JAXRDeleteConcept.java` shows how to delete a concept
- `JAXRGetMyObjects.java` lists all the objects that you own in a registry

The `<INSTALL>/jwstutorial13/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 `<INSTALL>/jwstutorial13/examples/jaxr/JAXRExamples.properties` as follows.

1. Edit the following lines 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 Java WSDP Registry Server.

```
## Uncomment one pair of query and publish URLs.
## IBM:
#query.url=http://uddi.ibm.com/testregistry/inquiryapi
#publish.url=https://uddi.ibm.com/testregistry/publishapi
## Microsoft:
#query.url=http://test.uddi.microsoft.com/inquire
#publish.url=https://test.uddi.microsoft.com/publish
## Registry Server:
query.url=http://localhost:8080/RegistryServer/
publish.url=http://localhost:8080/RegistryServer/
```

   If you are using the Java WSDP Registry Server, and if it 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`. If Tomcat is using a nondefault port, change `8080` to the correct value for your system.

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 URLs of the SAP registry; feel free to add them.

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 572). 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.

The Java WSDP Registry Server gives you more freedom to experiment with JAXR. You can publish as many organizations, concepts, and classification schemes to it as you wish. However, this registry comes with an empty database, so you must publish data to it yourself before you can perform queries on the data.

2. Edit the following lines to specify the user name and password you obtained when you registered with the registry. The defaults are the Registry Server default username and 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, 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 Java WSDP 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. If you are running Tomcat on a system other than your own, or if it is running on a nondefault port, change the following lines:

   ```
   link.uri=http://localhost:8080/hello-jaxrpc/hello?WSDL
   ...
   wsdlorg.svcbnd.uri=http://localhost:8080/hello-jaxrpc/hello
   ```

   Specify the fully qualified host name instead of `localhost`, or change `8080` to the correct value for your system.

5. 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. The Ant targets that run the client examples will use the latest version of the file.

Compiling the Examples

To compile the programs, go to the `<INSTALL>/jwstutorial13/examples/jaxr/` directory. A `build.xml` file allows you to use the command

```
ant compile
```

to compile all the examples. The Ant tool creates a subdirectory called `build`.

The runtime classpath setting in the `build.xml` file includes JAR files in several directories in the Java WSDP installation. All JAXR client examples require this classpath setting.

Running the Examples

If you are running the examples with the Java WSDP Registry Server, start the Java WSDP Tomcat.

On Windows, from the Start menu, choose Programs→Java Web Services Developer Pack 1.*x*→Start Tomcat or use the following command:

```
startup
```

On a UNIX system, use the following command:

```
startup.sh
```

The Registry Server is a Web application that is loaded when Tomcat starts.

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.

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 -Dquery-string=coff run-query
```

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 displayed by the `JAXR-Publish` program as input to the `run-delete` target:

```
ant -Dkey-string=keyString run-delete
```

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 Java WSDP 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 (including the `uuid:` prefix). For the Java WSDP Registry Server, you may use any string that is formatted as a UUID.

For a given registry, you only need to publish 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 (including the `uuid:` prefix) as input to the `run-publish-postal` target:

   ```
   ant -Duuid-string=uuidstring run-publish-postal
   ```

   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 -Dquery-string=coffee
   -Duuid-string=uuidstring run-query-postal
   ```

   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 600) 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 -Duuid-string=uuidstring run-delete-scheme
```

For the public UDDI registries, deleting a classification scheme removes it from the registry logically but not physically. The classification scheme will still be visible if, for example, you call the method `QueryManager.getRegisteredObjects`. However, you can no longer use the classification scheme. Therefore, you may prefer not to delete the classification scheme from the registry, in case you want to use it again. The public registries normally allow you to own up to 10 of these objects.

Publishing a Concept for a WSDL Document

To publish the location of the WSDL document for the JAX-RPC Hello Service, first deploy the service as described in Creating a Web Service with JAX-RPC (page 459).

Then run the `JAXRPublishConcept` program using the `run-publish-concept` target:

```
ant run-publish-concept
```

The program output displays the UUID string of the new concept, which is named “Hello Concept.” You will use this string in the next section.

After you run the `JAXRPublishConcept` program, you can run `JAXRPublishHelloOrg` to publish an organization that uses this concept.

Publishing an Organization with a WSDL Document in its Service Binding

To run the `JAXRPublishHelloOrg` example, use the Ant target `run-publish-hello-org`. Specify the string returned from `JAXRPublishConcept` (including the `uuid:` prefix) as input to this target:

```
ant -Duuid-string=uuidstring run-publish-hello-org
```

The program output displays the string value of the key of the new organization, which is named “Hello Organization.”

After you publish the organization, run the `JAXRQueryByWSDLClassification` example to search for it. To delete it, run `JAXRDelete`.

Running the JAXRQueryByWSDLClassification Example

To run the JAXRQueryByWSDLClassification example, use the Ant target run-query-wsdl. Specify a query-string argument on the command line to search the registry for specification concepts whose names contain that string. For example, the following command line searches for concepts whose names contain the string “helloconcept” (searching is not case-sensitive):

```
ant -Dquery-string=helloconcept run-query-wsdl
```

This example finds the concept and organization you published. A common string like “hello” returns many results from the public registries and is likely to run for several minutes.

Deleting a Concept

To run the JAXRDeleteConcept program, specify the UUID string displayed by the JAXRPublishConcept program as input to the run-delete-concept target:

```
ant -Duuid-string=uuidString run-delete-concept
```

Deleting a concept from a public UDDI registry is similar to deleting a classification scheme: the concept is removed logically but not physically. Do not delete the concept until after you have deleted any organizations that refer to it.

Getting a List of Your Registry Objects

To get a list of the objects you own in the registry—organizations, classification schemes, and concepts—run the JAXRGetMyObjects program by using the run-get-objects target:

```
ant run-get-objects
```

If you run this program with the Java WSDP Registry Server, it returns all the standard UDDI taxonomies provided with the Registry Server, not just the objects you have created.

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:
  `http://ebxmlrr.sourceforge.net/`
- Java 2 Platform, Enterprise Edition:
  `http://java.sun.com/j2ee/`
- Java Technology and XML:
  `http://java.sun.com/xml/`
- Java Technology & Web Services:
  `http://java.sun.com/webservices/index.html`

15

Java Servlet Technology

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 1103).

The Example Servlets

This chapter uses the Duke's Bookstore application to illustrate the tasks involved in programming servlets. Table 15–1 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 15–1 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 15–1 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 `<INSTALL>/jwstutorial13/examples/web/bookstore1/` directory created when you unzip the tutorial bundle (see Building and Running the Examples, page xxv). A sample `bookstore1.war` is provided in `<INSTALL>/jwstutorial13/examples/web/provided-wars/`. To build, package, deploy, and run the example:

1. Build and package the bookstore common files as described in Duke's Bookstore Examples (page 89).
2. In a terminal window, go to `<INSTALL>/jwstutorial13/examples/web/bookstore1/`.
3. Run `ant build`. This target will spawn any necessary compilations and copy files to the `<INSTALL>/jwstutorial13/examples/web/bookstore1/build/` directory.
4. Start Tomcat.
5. Perform all the operations described in Accessing Databases from Web Applications, page 90.
6. Run `ant install-config`. The `install-config` target notifies Tomcat that the new context is available. See Installing Web Applications (page 78).
7. To run the application, open the bookstore URL `http://localhost:8080/bookstore1/bookstore`.

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. Run `ant deploy`. The `deploy` target copies the WAR to Tomcat and notifies Tomcat that the new context is available.

Troubleshooting

The Duke's Bookstore database access object 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 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 the database server hasn't been started.

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 `errorpage.html`, you will have to look in the server 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 615).
2. Invokes the `service` method, passing a request and response object. Service methods are discussed in Writing Service Methods (page 616).

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 636).

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 609) 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 15–2 Servlet Life Cycle Events

Object	Event	Listener Interface and Event Class
Web context (See Accessing the Web Context, page 632)	Initialization and destruction	`javax.servlet.ServletContextListener` and `ServletContextEvent`
	Attribute added, removed, or replaced	`javax.servlet.ServletContextAttributeListener` and `ServletContextAttributeEvent`
Session (See Maintaining Client State, page 633)	Creation, invalidation, activation, passivation, and timeout	`javax.servlet.http.HttpSessionListener`, `javax.servlet.http.HttpSessionActivationListener`, and `HttpSessionEvent`
	Attribute added, removed, or replaced	`javax.servlet.http.HttpSessionAttributeListener` and `HttpSessionBindingEvent`

Table 15–2 Servlet Life Cycle Events (Continued)

Object	Event	Listener Interface and Event Class
`Request`	A servlet request has started being processed by Web components	`javax.servlet.ServletRequestListener` and `ServletRequestEvent`
	Attribute added, removed, or replaced	`javax.servlet.ServletRequestAttributeListener` and `ServletRequestAttributeEvent`

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);
    counter = new Counter();
    context.setAttribute("orderCounter", counter);
  }

  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 629).

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 15–3 lists the scope objects.

Table 15–3 Scope Objects

Scope Object	Class	Accessible From
Web context	`javax.servlet.ServletContext`	Web components within a Web context. See Accessing the Web Context (page 632).
session	`javax.servlet.http.HttpSession`	Web components handling a request that belongs to the session. See Maintaining Client State (page 633).
request	subtype of `javax.servlet.ServletRequest`	Web components handling the request.
page	`javax.servlet.jsp.JspContext`	The JSP page that creates the object. See Implicit Objects (page 654).

Figure 15–1 shows the scoped attributes maintained by the Duke's Bookstore application.

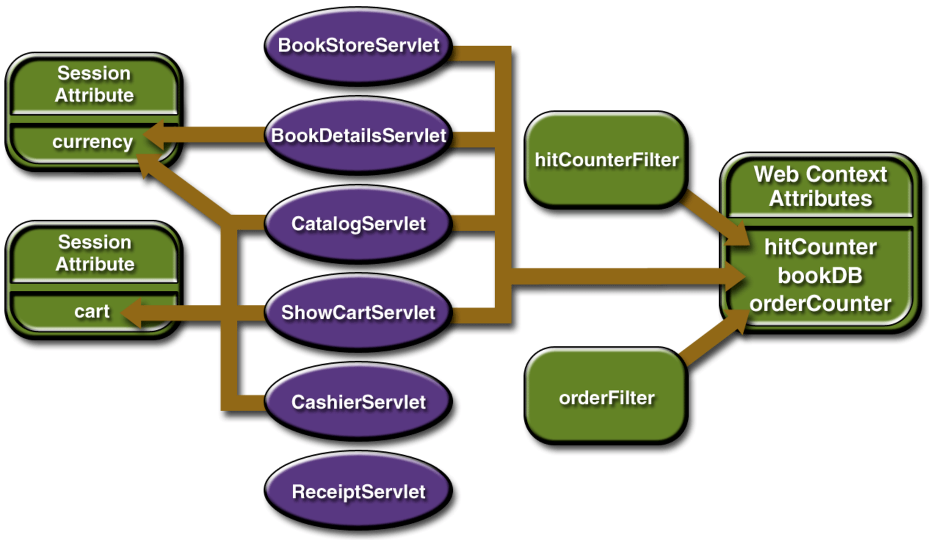


Figure 15–1 Duke's Bookstore Scoped Attributes

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. In addition, the Servlet 2.4 specification deprecates `SingleThreadModel`.

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.buy-`

Books 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 retrieve an output stream from the response, then fill in the response headers, and finally write any body content to the output stream. Response headers must always be set before the response has been committed. Any attempt to set/add headers after the response has been committed will be ignored by the Web container. 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 15–4, Table 15–5 gives some examples of how the URL will be parsed.

Table 15–4 Aliases

Pattern	Servlet
`/lawn/*`	`LawnServlet`
`/*.jsp`	`JSPServlet`

Table 15–5 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 with the `setContentType(String)` method. This method must be called before the response is committed. A registry of content type names is kept by the Internet Assigned Numbers Authority (IANA) at:

  `http://www.iana.org/assignments/media-types/`
- Indicate whether to buffer output with the `setBufferSize(int)` method. 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. The method must be called before any content is written or the response is committed.
- Set localization information such as locale and character encoding. See Chapter 23 for details.

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 or that a request has been redirected.
- 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 635).

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 630). 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:

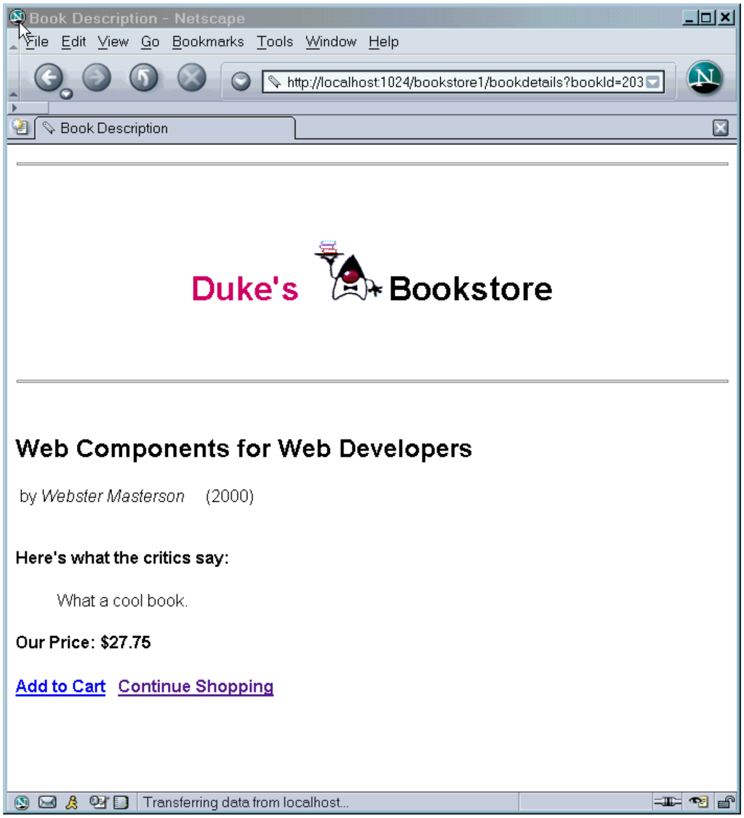


Figure 15–2 Book Details

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 `doFil-`

`ter` 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();
      System.out.println(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().getBytes().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 15–3 shows the entry page for Duke's Bookstore with the hit counter.

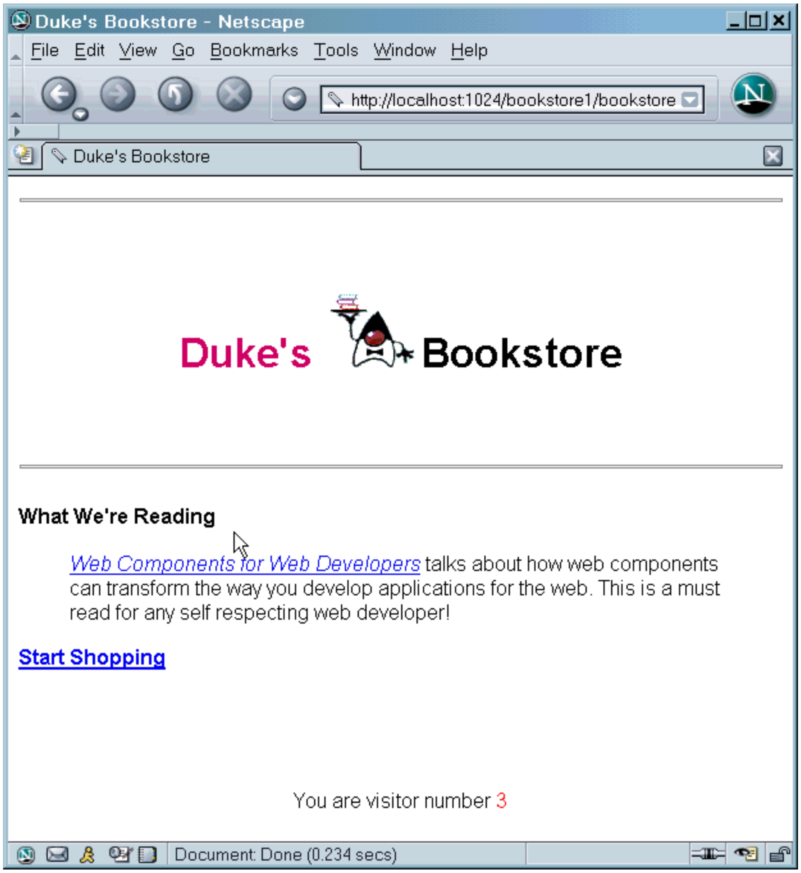


Figure 15–3 Duke's Bookstore

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 as follows

- Declare the filter. 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 name or by URL pattern.
- Constrain how the filter will be applied to requests by choosing one of the enumerated dispatcher options:
  - `REQUEST`-Only when the request come directly from the client.
  - `FORWARD`-Only when the request has been forwarded to a component (see Transferring Control to Another Web Component, page 631).
  - `INCLUDE`-Only when the request is being processed by a component that has been included (see Including Other Resources in the Response, page 630).
  - `ERROR`-Only when the request is being processed with the error page mechanism (see Handling Errors, page 611).

  You can direct the filter to be applied in any combination of the preceding situations by including multiple `dispatcher` elements. If no elements are specified, the default option is `REQUEST`.

The following elements show how to specify the Duke's Bookstore hit counter and order filters:

```
<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 15–6 summarizes the filter definition and mapping list for the Duke's Bookstore application. The filters are matched by servlet name and each filter chain contains only one filter.

Table 15–6 Duke's Bookstore Filter Definition and Mapping List

Filter	Class	Servlet
`HitCounterFilter`	`filters.HitCounterFilter`	`BookStoreServlet`
`OrderFilter`	`filters.OrderFilter`	`ReceiptServlet`

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 15–4, 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.

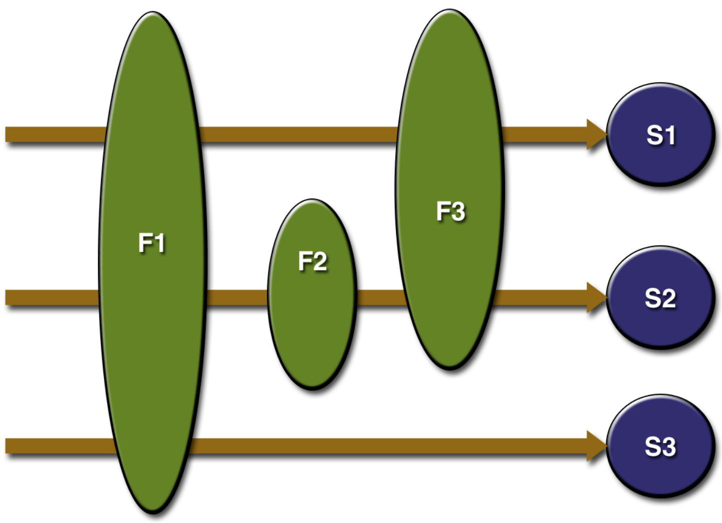


Figure 15–4 Filter to Servlet Mapping

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: indirectly and directly. A Web component indirectly invokes another Web resource when it embeds a URL that points to another Web component in content returned to a client. In the Duke's Bookstore application, most Web components contain embedded URLs that point to other Web components. For example, `ShowCartServlet` 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>  " + "<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>   <hr> <br> ");
  }
  public void doPost (HttpServletRequest request,
     HttpServletResponse response)
     throws ServletException, IOException {

     PrintWriter out = response.getWriter();
```

```
      out.println("<body bgcolor=\"#ffffff\">" +
      "<center>" + "<hr> <br>  " + "<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>   <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 URI is set to the path of the forwarded page. The original URI and its constituent parts are saved as a request attributes `javax.servlet.forward.[request_uri|context-path|servlet_path|path_info|query_string]`. The `Dispatcher` servlet, used by a version of the Duke's Bookstore application described in The Example JSP Pages (page 706), 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) {
     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 621). The filters store a counter as a context attribute. Recall from Controlling Concurrent Access to Shared Resources (page 613) 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 in the log.

```
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());
        ...
        System.out.println(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.

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 609). 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]MaxI-`

nactiveInterval 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>   <p><strong><a href=\"" +
  response.encodeURL(request.getContextPath() +
    "/bookcatalog") +
    "\">" + messages.getString("ContinueShopping") +
    "</a>      " +
    "<a href=\"" +
  response.encodeURL(request.getContextPath() +
    "/bookcashier") +
    "\">" + messages.getString("Checkout") +
    "</a>      " +
    "<a href=\"" +
  response.encodeURL(request.getContextPath() +
    "/bookshowcart?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 615):

```
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:

- Java Servlet 2.4 Specification
  `http://java.sun.com/products/servlet/download.html#specs`
- The Java Servlets Web site
  `http://java.sun.com/products/servlet`

16

JavaServer Pages Technology

JAVASERVER Pages (JSP) technology allows you to easily create Web content that has both static and dynamic components. JSP technology makes available 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
- An expression language 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 tutorial.

What Is a JSP Page?

A *JSP page* is a text 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.

The recommended file extension for the source file of a JSP page is `.jsp`. The page may be composed of a top file that includes other files that contain either a complete JSP page or a fragment of a JSP page. The recommended extension for the source file of a fragment of a JSP page is `.jspf`.

The JSP elements in a JSP page can be expressed in two syntaxes—standard and XML—though any given file can only use one syntax. A JSP page in XML syntax is an XML document and can be manipulated by tools and APIs for XML documents. This chapter and chapters through 19 document only the standard syntax. The JSP XML syntax is covered in Chapter 13. A syntax card and reference that summarizes both syntaxes is available at

```
http://java.sun.com/products/jsp/docs.html#syntax
```

Example

The Web page in Figure 16–1 is a form that allows you to select a locale and displays the date in a manner appropriate to the locale.

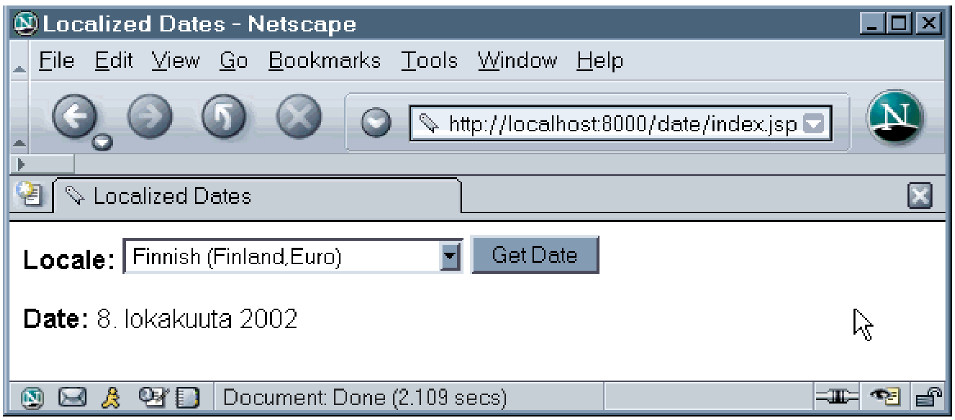


Figure 16–1 Localized Date Form

The source code for this example is in the `<*INSTALL*>/jwstutorial13/examples/web/date/` directory. 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 contain the following types of JSP constructs:

- A page directive (**`<%@page ... %>`**) sets the content type returned by the page.
- Tag library directives (**`<%@taglib ... %>`**) import custom tag libraries.
- **`jsp:useBean`** creates an object containing a collection of locales and initializes an identifier that points to that object.
- JSP expression language expressions (**`${ }`**) retrieve the value of object properties. The value of an are used to set tag attribute values.
- Custom tags set a variable (**`c:set`**), iterate over a collection of locale names (**`c:forEach)`**, and conditionally insert HTML text into the response (**`c:if`**, **`c:choose`**, **`c:when`**, **`c:otherwise`**).
- **`jsp:setProperty`** sets the value of an object property.
- A function (**`f:equals`**) tests the equality of an attribute and the current item of a collection. (Note: a built-in == operator is usually used to test equality).

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
    prefix="c" %>
<%@ taglib uri="/functions" prefix="f" %>
<html>
<head><title>Localized Dates</title></head>
<body bgcolor="white">
<jsp:useBean id="locales" scope="application"
  class="mypkg.MyLocales"/>

<form name="localeForm" action="index.jsp" method="post">
<c:set var="selectedLocaleString" value="${param.locale}" />
<c:set var="selectedFlag"
  value="${!empty selectedLocaleString}" />
<b>Locale:</b>
<select name=locale>
<c:forEach var="localeString" items="${locales.localeNames}" >
<c:choose>
  <c:when test="${selectedFlag}">
    <c:choose>
      <c:when
        test="${f:equals(selectedLocaleString,
          localeString)}" >
        <option selected>${localeString}</option>
      </c:when>
      <c:otherwise>
```

```
            <option>${localeString}</option>
          </c:otherwise>
        </c:choose>
      </c:when>
      <c:otherwise>
        <option>${localeString}</option>
      </c:otherwise>
    </c:choose>
    </c:forEach>
    </select>
    <input type="submit" name="Submit" value="Get Date">
    </form>

    <c:if test="${selectedFlag}" >
      <jsp:setProperty name="locales"
        property="selectedLocaleString"
        value="${selectedLocaleString}" />
      <jsp:useBean id="date" class="mypkg.MyDate"/>
      <jsp:setProperty name="date" property="locale"
        value="${locales.selectedLocale}"/>
      <b>Date: </b>${date.date}
    </c:if>
    </body>
    </html>
```

A sample `date.war` is provided in `<INSTALL>/jwstutorial13/examples/web/provided-wars/`. To build, package, deploy, and execute this example:

1. In a terminal window, go to `<INSTALL>/jwstutorial13/examples/web/date/`.
2. Run `ant build`. This target will spawn any necessary compilations and copy files to the `<INSTALL>/jwstutorial13/examples/web/date/build/` directory.
3. Start Tomcat.
4. Run `ant install`. The `install` target notifies Tomcat that the new context is available.
5. Set the character encoding in your browser to UTF-8.
6. Open the URL `http://localhost:8080/date` in a browser.

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 The Example Servlets (page 606) as a JSP page:

Table 16–1 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	`bookcatalog.jsp`
Add a book to the shopping cart	`bookcatalog.jsp` and `bookdetails.jsp`
Get detailed information on a specific book	`bookdetails.jsp`
Display the shopping cart	`bookshowcart.jsp`
Remove one or more books from the shopping cart	`bookshowcart.jsp`
Buy the books in the shopping cart	`bookcashier.jsp`
Receive an acknowledgement for the purchase	`bookreceipt.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 664). 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 the data access object.

```
package database;
public class BookDB {
  private String bookId = "0";
  private BookDBAO database = null;

  public BookDB () throws Exception {
  }
  public void setBookId(String bookId) {
    this.bookId = bookId;
  }
  public void setDatabase(BookDAO database) {
    this.database = database;
  }
  public BookDetails getBookDetails()
    throws Exception {
    return (BookDetails)database.getBookDetails(bookId);
  }
  ...
}
```

This version of the Duke's Bookstore application is organized along the Model-View-Controller (MVC) architecture. The MVC architecture is a widely-used architectural approach for interactive applications that separates functionality among application objects so as to minimize the degree of coupling between the objects. To achieve this, it divides applications into three layers: model, view, and controller. Each layer handles specific tasks and has responsibilities to the other layers:

- The model represents business data and business logic or operations that govern access and modification of this business data. The model notifies views when it changes and provides the ability for the view to query the model about its state. It also provides the ability for the controller to access application functionality encapsulated by the model. In the Duke's Bookstore application, the shopping cart and database helper object contain the business logic for the application.
- The view renders the contents of a model. It gets data from the model and specifies how that data should be presented. It updates data presentation when the model changes. A view also forwards user input to a controller. The Duke's Bookstore JSP pages format the data stored in the session-scoped shopping cart and the page-scoped database helper object.

- The controller defines application behavior. It dispatches user requests and 18selects views for presentation. It interprets user inputs and maps them into actions to be performed by the model. In a Web application, user inputs are HTTP GET and POST requests. A controller selects the next view to display based on the user interactions and the outcome of the model operations. In the Duke's Bookstore application, the `Dispatcher` servlet is the controller. It examines the request URL, creates and initializes a session-scoped JavaBeans component—the shopping cart—and dispatches requests to view JSP pages.

Note: When employed in a Web application, the MVC architecture is often referred to as a Model-2 architecture. The bookstore example discussed in the previous chapter, which intermixes presentation and business logic, follows what is known as a Model-1 architecture. The Model-2 architecture is the recommended approach to designing Web applications.

In addition, this version of the application uses several custom tags from the JavaServer Pages Standard Tag Library (JSTL) (see Chapter):

- `c:if` and `c:choose`, `c:when`, and `c:otherwise` for flow control
- `c:set` for setting scoped variables
- `c:url` for encoding URLs
- `fmt:message`, `fmt:formatNumber`, and `fmt:formatDate` for providing locale-sensitive messages, numbers, and dates

Custom tags are the preferred mechanism for performing a wide variety of dynamic processing tasks, including accessing databases, using enterprise services such as e-mail and directories, and flow control. In earlier versions of JSP technology, such tasks were performed with JavaBeans components in conjunction with scripting elements (discussed in Chapter 19). Though still available in JSP 2.0, scripting elements tend to make JSP pages more difficult to maintain because they mix presentation and logic, which is discouraged in page design. Custom tags are introduced in Using Custom Tags (page 669) and described in detail in Chapter 18.

Finally, this version of the example contains an applet to generate a dynamic digital clock in the banner. See Including an Applet (page 674) for a description of the JSP element that generates HTML for downloading the applet.

1. The source code for the application is located in the `<INSTALL>/jwstutorial13/examples/web/bookstore2/` directory (see Building

and Running the Examples, page xxv). A sample `bookstore2.war` is provided in `<INSTALL>/jwstutorial13/examples/web/provided-wars/`. To build, package, deploy, and run the example:

1. Build and package the bookstore common files as described in Duke's Bookstore Examples (page 89).
2. In a terminal window, go to `<INSTALL>/jwstutorial13/examples/web/bookstore2/`.
3. Run `Ant build`. This target will spawn any necessary compilations and copy files to the `<INSTALL>/jwstutorial13/examples/web/bookstore2/build/` directory.
4. Start Tomcat.
5. Perform all the operations described in Accessing Databases from Web Applications, page 90.
6. Run `ant install-config`. The `install` target notifies Tomcat that the new context is available. See Installing Web Applications (page 78).

7. Open the bookstore URL `http://localhost:8080/bookstore2/bookstore`. Click on the Start Shopping link and you will see the screen in Figure 16–2:

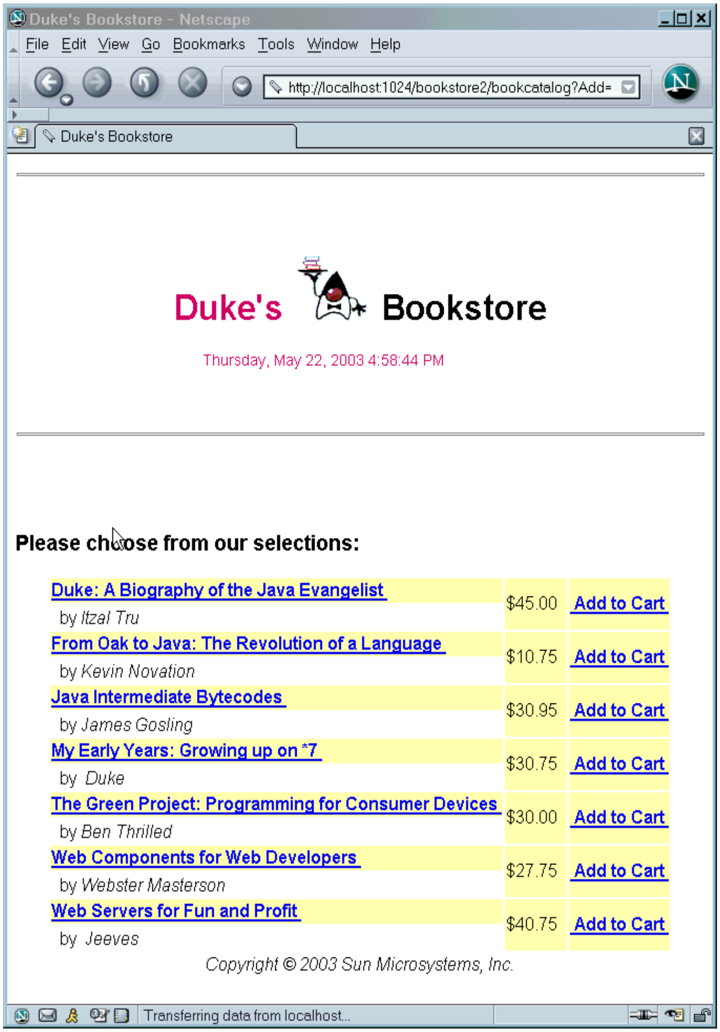


Figure 16–2 Book Catalog

See Troubleshooting (page 608) for help with diagnosing common problems related to the database server. If the messages in your pages appear as strings of the form `??? Key ???`, the likely cause is that you have not provided the correct resource bundle basename as a context parameter.

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. You will notice that many sections in this chapter refer to classes and methods described in Chapter 15.

When a request is mapped to a JSP page, the Web container first checks whether the JSP page's servlet is older than the JSP page. If the servlet is older, the Web container 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 response stream. 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 Chapter 19 for details.
- Expression language expressions are passed as parameters to calls to the JSP expression evaluator.
- `jsp:[set|get]Property` elements are converted into method calls to JavaBeans components.
- `jsp:[include|forward]` elements are converted to invocations of the Java Servlet API.
- The `jsp:plugin` element is converted to browser-specific markup for activating an applet.
- Custom tags are converted into calls to the tag handler that implements the custom tag.

In the Java WSDP, the source for the servlet created from a JSP page named *pageName* is in the file:

```
<JWSDP_HOME>/work/Catalina/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/Catalina/localhost/context_root/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 is encountered during either phase, the server will return `JasperException` and a message that includes the name of the JSP page 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 608):

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 prelude.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" %>
```

This directive makes an object of type javax.servlet.jsp.ErrorData 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. You access the error data object in an EL expression via the page context. Thus, ${pageContext.errorData.statusCode} is used to retrieve the status code and ${pageContext.errorData.throwable} retrieves the exception. If the exception is generated during the evaluation of an EL expression, you can retrieve the root cause of the exception with the expression ${pageContext.errorData.throwable.rootCause}. For example, the error page for the Duke's Bookstore is:

```
<%@ page isErrorPage="true" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
  prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt"
  prefix="fmt" %>
<html>
<head>
<title><fmt:message key="ServerError"/></title>
</head>
<body bgcolor="white">
<h3>
<fmt:message key="ServerError"/>
</h3>
<p>
${pageContext.errorData.throwable}
<c:choose>
  <c:when test="${!empty
     pageContext.errorData.throwable.cause}">
```

```
      : ${pageContext.errorData.throwable.cause}
    </c:when>
    <c:when test="${!empty
      pageContext.errorData.throwable.rootCause}">
      : ${pageContext.errorData.throwable.rootCause}
    </c:when>
  </c:choose>
  </body>
  </html>
```

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.

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 content type at the beginning of your JSP page. The purpose of the `contentType` directive is to allow the browser to correctly interpret the resulting content. So, 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:

```
http://www.iana.org/assignments/media-types/
```

Response and Page Encoding

You also use the `contentType` attribute to specify the encoding of the response. For example, the date application specifies that the page should be encoded using UTF-8, an encoding that supports almost all locales, with the following `page` directive:

```
<%@ page contentType="text/html; charset=UTF-8" %>
```

If the response encoding weren't set, the localized dates would not be rendered correctly.

To set the source encoding of the page itself, you would use the following `page` directive.

```
<%@ page pageEncoding="UTF-8" %>
```

You can also set the page encoding of a set of JSP pages. The value of the page encoding varies depending on the configuration specified in the JSP configuration section of the Web application deployment descriptor (see Declaring Page Encodings, page 679).

Creating Dynamic Content

You create dynamic content by accessing Java programming language object properties.

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, session, or application. Many of the objects are defined by the Java Servlet technology underlying JSP technology and are discussed at length in Chapter 15. The section Implicit Objects (page 659) explains how you access implicit objects using the JSP expression language.

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. The main way to create and use application-specific objects within a JSP page is to use JSP standard tags discussed in JavaBeans

Components (page 663) to create JavaBeans components and set their properties, and EL expressions to access their properties. You can also access JavaBeans components and other objects in scripting elements, which are described in Chapter 19.

Shared Objects

The conditions affecting concurrent access to shared objects described in Controlling Concurrent Access to Shared Resources (page 613) 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 the `isThreadSafe` attribute 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 context object (see Implicit Objects, page 659).

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. Futhermore, it is not recommended to set `isThreadSafe` to `false` because the JSP page's generated servlet will implement the `javax.servlet.SingleThreadModel` interface and since the Servlet 2.4 specification deprecates `SingleThreadModel`, the generated servlet will contain deprecated code.

Expression Language

A primary feature of JSP technology version 2.0 is its support for an expression language. An expression language makes it possible to easily access application data stored in JavaBeans components. For example, the JSP expression language allows a page author to access a bean using a simple syntax such as

```
${name}
```

for a simple variable or

```
${name.foo.bar}
```

for a nested property.

The `test` attribute of the following conditional tag is supplied with an EL expression that compares the number of items in the session-scoped bean named `cart` with 0:

```
<c:if test="${sessionScope.cart.numberOfItems > 0}">
  ...
</c:if>
```

The JSP expression evaluator is responsible for handling EL expressions, which may include literals and are enclosed by the `${ }` characters. For example:

```
<c:if test="${bean1.a < 3}" >
  ...
</c:if>
```

Any value that does not begin with `${` is treated as a literal that is parsed to the expected type using the `PropertyEditor` for the type:

```
<c:if test="true" >
...
</c:if>
```

Literal values that contain the `${` characters must be escaped as follows:

```
<mytags:example attr1="an expression is ${'${'}true}" />
```

Deactivating Expression Evaluation

Since the pattern that identifies EL expressions—`${ }`—was not reserved in the JSP specifications before JSP 2.0, there may be applications where such a pattern is intended to pass through verbatim. To prevent the pattern from being evaluated, EL evaluation can be deactivated.

To deactivate the evaluation of EL expressions you specify the `isELIgnored` attribute of the `page` directive

```
<%@ page isELIgnored ="true|false" %>
```

The valid values of this attribute are `true` and `false`. If `true`, EL expressions are ignored when they appear in template text or tag attributes. If `false`, EL expressions are evaluated by the container.

The default value varies depending on the version of the Web application deployment descriptor. The default mode for JSP pages delivered using a Servlet 2.3 or earlier descriptor is to ignore EL expressions; this provides backwards compatibility. The default mode for JSP pages delivered with a Servlet 2.4 descriptor is to evaluate EL expressions; this automatically provides the default that most applications want. You can also deactivate EL expression evaluation for a group of JSP pages (see Deactivating EL Evaluation, page 677).

Using Expressions

EL Expressions can be used in two situations:

- In template text
- In any standard or custom tag attribute that can accept an expression

The value of an expression in template text is computed and inserted into the current output. An expression *will not* be evaluated if the body of the tag is declared to be `tagdependent` (see body-content Attribute, page 717).

Three ways to set a tag attribute value:

- With a single expression construct:

  ```
  <some:tag value="${expr}"/>
  ```

  The expression is evaluated and the result is coerced to the attribute's expected type.

- With one or more expressions separated or surrounded by text:

  ```
  <some:tag value="some${expr}${expr}text${expr}"/>
  ```

  The expressions are evaluated from left to right. Each expression is coerced to a `String` and then concatenated with any intervening text. The resulting `String` is then coerced to the attribute's expected type.

- With only text:

  ```
  <some:tag value="sometext"/>
  ```

  In this case, the attribute's `String` value is coerced to the attribute's expected type.

Expressions used to set attribute values are evaluated in the context of an expected type. If the result of the expression evaluation does not match the expected type exactly a type conversion will be performed. For example, the expression `${1.2E4 + 1.4}` provided as the value of an attribute of type `float`, will result in the following conversion: `Float.valueOf("1.2E4 + 1.4").floatValue()`. See Section JSP2.8 of the JSP 2.0 Specification for the complete type conversion rules.

Variables

The JSP container evaluates a variable that appears in an expression by looking up its value according to the behavior of `PageContext.findAttribute(String)`. For example, when evaluating the expression `${product}`, the container will look for `product` in the page, request, session, and application scopes and will return its value. If `product` is not found, `null` is returned. A variable that matches one of the implicit objects described in Implicit Objects (page 659) will return that implicit object instead of the variable's value.

Properties of variables are accessed using the `.` operator, and may be nested arbitrarily.

The JSP expression language unifies the treatment of the `.` and `[]` operators. `expr-a.expr-b` is equivalent to `a["expr-b"]`; that is, the expression `expr-b` is used to construct a literal whose value is the identifier, and then the `[]` operator is used with that value.

To evaluate `expr-a[expr-b]`, evaluate `expr-a` into `value-a` and evaluate `expr-b` into `value-b`. If either `value-a` or `value-b` is null, return null.

- If `value-a` is a `Map`, return `value-a.get(value-b)`. If `!value-a.containsKey(value-b)`, then return `null`.
- 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 the coercion couldn't be performed, an error is returned. If the `get` call returns an `IndexOutOfBoundsException`, `null` is returned. If the `get` call returns another exception, an error is returned.
- If `value-a` is a JavaBeans object, coerce `value-b` to `String`. If `value-b` is a readable property of `value-a`, then return the result of a `get` call. If the `get` method throws an exception, an error is returned.

Implicit Objects

The JSP expression language defines a set of implicit objects:

- `pageContext` - The context for the JSP page. Provides access to various objects including:
  - `servletContext` - The context for the JSP page's servlet and any Web components contained in the same application. See Accessing the Web Context (page 632).
  - `session` - The session object for the client. See Maintaining Client State (page 633).
  - `request` - The request triggering the execution of the JSP page. See Getting Information from Requests (page 617).
  - `response` - The response returned by the JSP page. See Constructing Responses, page 619).

In addition, several implicit objects are available that allow easy access to the following objects:

- `param` - maps a request parameter name to a single value
- `paramValues` - maps a request parameter name to an array of values
- `header` - maps a request header name to a single value
- `headerValues` - maps a request header name to an array of values
- `cookie` - maps a cookie name to a single cookie
- `initParam` - maps a context initialization parameter name to a single value

Finally, there are objects that allow access to the various scoped variables described in Using Scope Objects (page 612).

- `pageScope` - maps page-scoped variable names to their values
- `requestScope` - maps request-scoped variable names to their values
- `sessionScope` - maps session-scoped variable names to their values
- `applicationScope` - maps application-scoped variable names to their values

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 `pageContext` attribute containing some other value.

Literals

The JSP expression language defines the following literals:

- Boolean: `true` and `false`
- Integer: 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

In addition to the `.` and `[]` operators discussed in Variables (page 658), the JSP expression language provides the following operators:

- Arithmetic: `+`, `-` (binary), `*`, `/` and `div`, `%` and `mod`, `-`(unary)
- 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.
- Conditional: `A ? B : C`. Evaluate `B` or `C`, depending on the result of the evaluation of `A`.

The precedence of operators highest to lowest, left to right is:

- `[] .`
- `()` - Used to change the precedence of operators.
- `-` (unary) `not ! empty`
- `* / div % mod`
- `+ -` (binary)
- `< > <= >= lt gt le ge`
- `== != eq ne`
- `&& and`
- `|| or`
- `? :`

Reserved Words

The following words are reserved for the JSP expression language and should not be used as identifiers.

```
and  eq   gt   true    instanceof
or   ne   le   false   empty
not  lt   ge   null    div     mod
```

Note that many of these words are not in the language now, but they may be in the future, so you should avoid using them.

Examples

Table 16–2 contains example EL expressions and the result of evaluating the expressions.

Table 16–2 Example Expressions

EL Expression	Result
${1 > (4/2)}	false
${4.0 >= 3}	true
${100.0 == 100}	true
${(10*10) ne 100}	false
${'a' < 'b'}	true
${'hip' gt 'hit'}	false
${4 > 3}	true
${1.2E4 + 1.4}	12001.4
${3 div 4}	0.75
${10 mod 4}	2
${!empty param.Add}	True if the request parameter named Add is null or an empty string.
${pageContext.request.contextPath}	The context path

Table 16–2 Example Expressions (Continued)

EL Expression	Result
`${sessionScope.cart.numberOfItems}`	The value of the `numberOfItems` property of the session-scoped attribute named `cart`
`${param['mycom.productId']}`	The value of the request parameter named `mycom.productId`
`${header["host"]}`	The host
`${departments[deptName]}`	The value of the entry named `deptName` in the `departments` map
`${requestScope['javax.servlet.forward.servlet_path']}`	The value of the request-scoped attribute named `javax.servlet.forward.servlet_path`

Functions

The JSP expression language allows you to define a function that can be invoked in an expression. Functions are defined using the same mechanisms as custom tags (See Using Custom Tags, page 669 and Chapter 18).

Using Functions

Functions can appear in template text and tag attribute values.

To use a function in a JSP page, you import the tag library containing the function using a `taglib` directive. Then, you preface the function invocation with the prefix declared in the directive.

For example, the date example page `index.jsp` imports the `/functions` library and invokes the function `equals` in an expression:

```
<%@ taglib prefix="f" uri="/functions"%>
...
     <c:when
        test="${f:equals(selectedLocaleString,
           localeString)}" >
```

Defining Functions

To define a function you program it as a public static method in a public class. The `mypkg.MyLocales` class in the `date` example defines a function that tests the equality of two `Strings` as follows:

```
package mypkg;
public class MyLocales {

   ...
   public static boolean equals( String l1, String l2 ) {
      return l1.equals(l2);
   }
}
```

Then, you map the function name as used in the EL expression to the defining class and function signature in a TLD. The following `functions.tld` file in the date example maps the `equals` function to the class containing the implementation of the function `equals` and the signature of the function:

```
<function>
   <name>equals</name>
   <function-class>mypkg.MyLocales</function-class>
   <function-signature>boolean equals( java.lang.String,
      java.lang.String )</function-signature>
</function>
```

A tag library can only have one `function` element with any given `name` element.

JavaBeans Components

JavaBeans components are Java classes that can be easily reused and composed together into applications. Any Java class that follows certain design conventions is a JavaBeans component.

JavaServer Pages technology directly supports using JavaBeans components with standard JSP language elements. You can easily create and initialize beans and get and set the values of their properties.

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

A property does not have to be implemented by an instance variable. It must simply be accessible using public methods that conform to the following 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 `database.BookDBAO`. The JSP pages `showcart.jsp` and `cashier.jsp` access the bean `cart.ShoppingCart`, which represents a user's shopping cart.

The `BookDB` bean has two writable properties, `bookId` and `database`, and three readable properties, `bookDetails`, `numberOfBooks`, and `books`. These latter properties do not correspond to any instance variables, but are a function of the `bookId` and `database` properties.

```
package database
public class BookDB {
private String bookId = "0";
private BookDBAO database = null;
public BookDB () {
}
```

```
    public void setBookId(String bookId) {
    this.bookId = bookId;
    }
    public void setDatabase(BookDBAO database) {
    this.database = database;
    }
    public BookDetails getBookDetails() throws
      BookNotFoundException {
      return (BookDetails)database.getBookDetails(bookId);
    }
    public Collection getBooks() throws BooksNotFoundException {
      return database.getBooks();
    }
    public void buyBooks(ShoppingCart cart)
      throws OrderException {
      database.buyBooks(cart);
    }
    public int getNumberOfBooks() throws BooksNotFoundException {
      return database.getNumberOfBooks();
    }
}
```

Creating and Using a JavaBeans Component

You declare that your JSP page will use a JavaBeans component using a `jsp:useBean` element. There are two forms:

```
<jsp:useBean id="beanName"
  class="fully_qualified_classname" scope="scope"/>
```

and

```
<jsp:useBean id="beanName"
  class="fully_qualified_classname" scope="scope">
  <jsp:setProperty .../>
</jsp:useBean>
```

The second form 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 612). The value of the `id` attribute determines the *name* of the bean in the scope and the *identifier* used to reference the bean in EL expressions, other JSP elements, and scripting expressions (see Chapter 19). The value supplied for the `class` attribute must be a fully-qualified class name. Note that beans cannot be in the unnamed package. Thus the format of the value must be *package_name.class_name*.

The following element creates an instance of `mypkg.myLocales` if none exists, stores it as an attribute of the application scope, and makes the bean available throughout the application by the identifier `locales`:

```
<jsp:useBean id="locales" scope="application"
  class="mypkg.MyLocales"/>
```

Setting JavaBeans Component Properties

The standard way to set JavaBeans component properties in a JSP page is with the `jsp:setProperty` element. The syntax of the `jsp:setProperty` element depends on the source of the property value. Table 16–3 summarizes the various ways to set a property of a JavaBeans component using the `jsp:setProperty` element.

Table 16–3 Valid Bean Property Assignments from String Values

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="*"/>

Table 16–3 Valid Bean Property Assignments from String Values (Continued)

Value Source	**Element Syntax**

Expression	`<jsp:setProperty name="`*`beanName`*`" property="`*`propName`*`" value="`*`expression`*`"/>`
	
	`<jsp:setProperty name="`*`beanName`*`" property="`*`propName`*`" >`
	`<jsp:attribute name="value">`
	`expression`
	`</jsp:attribute>`
	`</jsp:setProperty>`

	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 16–4. 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 16–4 Valid Property Value Assignments from String Values

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)`

Table 16–4 Valid Property Value Assignments from String Values (Continued)

Property Type	Conversion on String Value
`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 use an expression to set the value of a property whose type is a compound Java programming language type. 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 to set the current book from a request parameter in the database helper bean in `bookstore2/web/bookdetails.jsp`:

```
<c:set var="bid" value="${param.bookId}"/>
<jsp:setProperty name="bookDB" property="bookId"
  value="${bid}" />
```

The following fragment from the page `bookstore2/web/bookshowcart.jsp` illustrates how to initialize a `BookDB` bean with a `database` object. Because the initialization is nested in a `useBean` element, it is only executed when the bean is created.

```
<jsp:useBean id="bookDB" class="database.BookDB" scope="page">
  <jsp:setProperty name="bookDB" property="database"
    value="${bookDBAO}" />
</jsp:useBean>
```

Retrieving JavaBeans Component Properties

The main way to retrieve JavaBeans component properties is with the JSP expression language. Thus, to retrieve a book title, the Duke's Bookstore application uses the following expression:

```
${bookDB.bookDetails.title}
```

Another way to retrieve component properties is to use the `jsp:getProperty` element. This element converts the value of the property into a `String` and inserts the value into the response stream:

```
<jsp:getProperty name="beanName" property="propName"/>
```

Note that *`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. Although the preferred approach to getting properties is to use an EL expression, the `getProperty` element is available if you need to disable expression evaluation.

Using Custom Tags

Custom tags are user-defined JSP language elements that encapsulate recurring tasks. Custom tags are distributed in a *tag library*, which defines a set of related custom tags and contains the objects that implement the tags.

Custom tags have the syntax

```
<prefix:tag attr1="value" ... attrN="value" />
```

or

```
<prefix:tag attr1="value" ... attrN="value" >
  body
</prefix:tag>
```

where `prefix` distinguishes tags for a library, `tag` is the tag identifier, and `attr1 ... attrN` are attributes that modify the behavior of the tag.

To use a custom tag in a JSP page, you must:

- Declare the tag library containing the tag
- Make the tag library implementation available to the Web application

See Chapter 16 for detailed information on the different types of tags and how to implement tags.

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 from that tag library is used. If you forget to include the `taglib` directive for a tag library in a JSP page, the JSP compiler will treat any invocation of a custom tag from that library as template data, and simply insert the text of the custom tag call into the response.

```
<%@ taglib prefix="tt" [tagdir=/WEB-INF/tags/dir | uri=URI ] %>
```

The `prefix` attribute defines the prefix that distinguishes tags defined by a given tag library from those provided by other tag libraries.

If the tag library is defined with tag files (see Encapsulating Reusable Content using Tag Files, page 714), you supply the `tagdir` attribute to identify the location of the files. The value of the attribute must start with `/WEB-INF/tags/` and a translation error will occur if the value points to a directory that doesn't exist or if used in conjunction with the `uri` attribute.

The `uri` attribute refers to a URI that uniquely identifies the tag library descriptor (TLD), a document that describes the tag library (See Tag Library Descriptors, page 728).

Tag library descriptor file names must have the extension `.tld`. TLD files are stored in the `WEB-INF` directory or subdirectory of the WAR file or in the `META-INF/` directory or subdirectory of a tag library packaged in a JAR. You can reference a TLD directly or indirectly.

The following `taglib` directive directly references a TLD filename:

```
<%@ taglib prefix="tlt" uri="/WEB-INF/iterator.tld"%>
```

This `taglib` directive uses a short logical name to indirectly reference the TLD:

```
<%@ taglib prefix="tlt" uri="/tlt"%>
```

The `iterator` example defines and uses a simple iteration tag. The JSP pages use a logical name to reference the TLD. A sample `iterator.war` is provided in `<INSTALL>/jwstutorial13/examples/web/provided-wars/`. To build the example:

1. In a terminal window, go to `<INSTALL>/jwstutorial13/examples/web/iterator/`.
2. Run `ant build`. This target will spawn any necessary compilations and copy files to the `<INSTALL>/jwstutorial13/examples/web/iterator/build/` directory.

You map a logical name to an absolute location in the Web application deployment descriptor. The iterator example specifies the mapping of the logical name `/tlt` to the absolute location `/WEB-INF/iterator.tld`, with the following descriptor element:

```
<jsp-config>
  <taglib>
    <taglib-uri>/tlt</taglib-uri>
    <taglib-location>/WEB-INF/iterator.tld</taglib-location>
  </taglib>
</jsp-config>
```

You can also reference a TLD in a `taglib` directive with an absolute URI. For example, the absolute URIs for the JSTL library are:

- Core: `http://java.sun.com/jsp/jstl/core`
- XML: `http://java.sun.com/jsp/jstl/xml`
- Internationalization: `http://java.sun.com/jsp/jstl/fmt`
- SQL: `http://java.sun.com/jsp/jstl/sql`
- Functions: `http://java.sun.com/jsp/jstl/functions`

When you reference a tag library with an absolute URI that exactly matches the URI declared in the `taglib` element of the TLD (see Tag Library Descriptors, page 728), you do not have to add the `taglib` element to `web.xml`; the JSP container automatically locates the TLD inside the JSTL library implementation.

Including the Tag Library Implementation

In addition to declaring the tag library, you also need to make the tag library implementation available to the Web application. There are several ways to do this. Tag library implementations can be included in a WAR in an unpacked format: tag files are packaged in the `/WEB-INF/tag/` directory and tag handler classes are packaged in the `/WEB-INF/classes/` directory of the WAR. Tag libraries already packaged into a JAR file are included in the `/WEB-INF/lib/` directory of the WAR. Finally, an application server may load a tag library into all the Web applications running on the server. For example, in the Java WSDP, the JSTL TLDs `and` libraries are distributed in the archives standard.jar and jstl.jar in `<`*`JWSDP_HOME`*`>/jstl/lib/`. If you copy these archives to the directory `<`*`JWSDP_HOME`*`>/common/lib`, they will automatically be loaded into the classpath of all Web applications running on Tomcat.

In the `iterator` example, the Ant build script compiles the `iterator` tag library implementation into the `/WEB-INF/classes/` directory. To install the `iterator` example into Tomcat:

1. Start Tomcat.
2. Run `ant install`. The `install` target notifies Tomcat that the new context is available.

To run the `iterator` application, open the URL `http://localhost:8080/iterator` in a browser.

Reusing Content in JSP Pages

There are many mechanisms for reusing JSP content in a JSP page. Three mechanisms that can be categorized as direct reuse—the `include` directive, preludes and codas, and the `jsp:include` element—are discussed below. An indirect method of content reuse occurs when a tag file is used to define a custom tag that is used by many Web applications. Tag files are discussed in the section Encapsulating Reusable Content using Tag Files (page 714) in Chapter 18.

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 Duke's Bookstore application pages could include the file `banner.jspf` which contains the banner content, with the following directive:

```
<%@ include file="banner.jspf" %>
```

Another way to do a static include is with the prelude and coda mechanism described in Defining Implicit Includes (page 679). This is the approach used by the Duke's Bookstore application.

Because you must put an `include` directive in each file that reuses the resource referenced by the directive, this approach has its limitations. Preludes and codas can only be applied to the beginning and end of pages. For a more flexible approach to building pages out of content chunks, see A Template Tag Library (page 750).

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" />
```

The `hello1` application discussed in Packaging Web Applications (page 77) includes the page that generates the response with the following statement:

```
<jsp:include page="response.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 631). 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>
```

When doing `jsp:include` or `jsp:forward`, the included page or forwarded page will see the original request object, with the original parameters augmented with the new parameters and new values taking precedence over existing values when applicable.For example, if the request has a parameter `A=foo` and a parameter `A=bar` is specified for forward, the forwarded request shall have `A=bar,foo`. Note that the new parameter has precedence.

The scope of the new parameters is the `jsp:include` or `jsp:forward` call; that is in the case of an `jsp:include` the new parameters (and values) will not apply after the 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 override the default URL where the plug-in can be downloaded.

The `jsp:params` element specifies 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 `/template/prelude.jspf` creates the banner that displays a dynamic digital clock generated by `DigitalClock`:

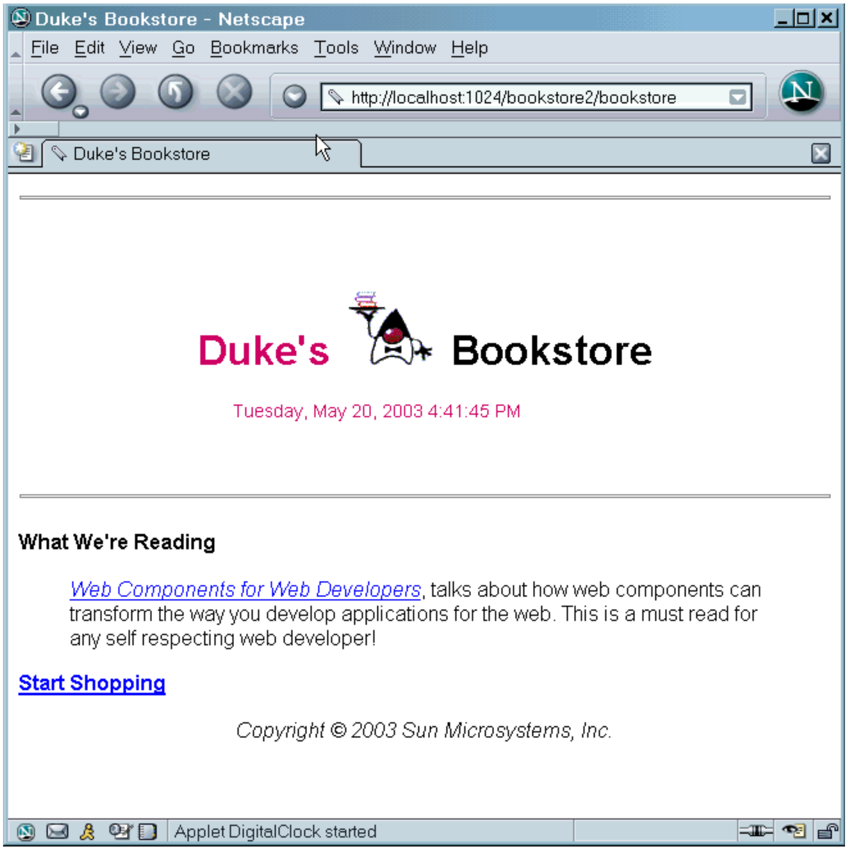


Figure 16–3 Duke’s Bookstore with Applet

The `jsp:plugin` element used to download the applet follows:

```
<jsp:plugin
  type="applet"
  code="DigitalClock.class"
  codebase="/bookstore2"
  jreversion="1.4"
  align="center" height="25" width="300"
  nspluginurl="http://java.sun.com/j2se/1.4.2/download.html"
  iepluginurl="http://java.sun.com/j2se/1.4.2/download.html" >
  <jsp:params>
    <jsp:param name="language"
```

```
          value="${pageContext.request.locale.language}" />
        <jsp:param name="country"
          value="${pageContext.request.locale.country}" />
        <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>
```

Setting Properties for Groups of JSP Pages

It is possible to specify certain properties for a group of JSP pages:

- Expression language evaluation
- Treatment of scripting elements (see Disabling Scripting, page 759)
- Page encoding
- Automatic prelude and coda includes

A JSP property group is defined by naming the group and specifying one or more URL patterns; all the properties in the group apply to the resources that match any of the URL patterns. If a resource matches URL patterns in more than one group, the pattern that is most specific applies.To define a property group, you add the `jsp-property-group` element within a `jsp-config` element to the Web application deployment descriptor (see Example, page 679). The following sections discuss the properties and how they are interpreted for various combinations of group properties, individual page directives, and Web application deployment descriptor version.

Deactivating EL Evaluation

Each JSP page has a default mode for EL expression evaluation. The default value varies depending on the version of the Web application deployment descriptor. The default mode for JSP pages delivered using a Servlet 2.3 or earlier descriptor is to ignore EL expressions; this provides backwards compatibility. The default mode for JSP pages delivered with a Servlet 2.4 descriptor is to evaluate EL expressions; this automatically provides the default that most applications want. For tag files (see Encapsulating Reusable Content using Tag Files, page 714), the default is to always evaluate expressions.

You can override the default mode through the `isELIgnored` attribute of the `page` directive in JSP pages and the `isELIgnored` attribute of the `tag` directive in tag files. The default mode can also be explicitly changed by adding the `el-ignored` element within a `jsp-property-group` element to the Web application deployment descriptor. Table 16–5 summarizes the EL evaluation settings for JSP pages and their meanings:

Table 16–5 EL Evaluation Settings for JSP Pages

JSP Configuration	Page Directive `isELIgnored`	EL Encountered
Unspecified	Unspecified	Evaluated if 2.4 `web.xml` Ignored if <= 2.3 `web.xml`
`false`	Unspecified	Evaluated
`true`	Unspecified	Ignored
Overridden by page directive	`false`	Evaluated
Overridden by page directive	`true`	Ignored

Table 16–6 summarizes the EL evaluation settings for tag files and their meanings:

Table 16–6 EL Evaluation Settings for Tag Files

Tag Directive `isELIgnored`	EL Encountered
Unspecified	Evaluated
`false`	Evaluated
`true`	Ignored

Declaring Page Encodings

You set the page encoding of a group of JSP pages by adding the `page-encoding` element within a `jsp-property-group` element to the Web application deployment descriptor. Valid values are the same as the `pageEncoding` attribute of the `page` directive. A translation-time error results if you define the page encoding of a JSP page with one value in the JSP configuration element and then give it a different value in a `pageEncoding` directive.

Defining Implicit Includes

You can implicitly include preludes and codas for a group of JSP pages by adding `include-prelude` and `include-coda` elements respectively within a `jsp-property-group` element to the Web application deployment descriptor. Their values are context-relative paths that must correspond to elements in the Web application. When the elements are present, the given paths are automatically included (as in an `include` directive) at the beginning and end of each JSP page in the property group respectively. When there is more than one include or coda element in a group, they are included in the order they appear. When more than one JSP property group applies to a JSP page, the corresponding elements will be processed in the same order as they appear in the JSP configuration section.

For example, the Duke's Bookstore uses the files `/template/prelude.jspf` and `/template/coda.jspf` to include the banner and other boilerplate in each screen. Preludes and codas can only put the included code at the beginning and end of each file. For a more flexible approach to building pages out of content chunks, see A Template Tag Library (page 750).

Example

The following JSP configuration element from Duke's Bookstore declares all files named `/*.jsp` are in a property group named `bookstore2`. It specifies that the EL should not be ignored, that scripting is valid, that none of the pages are in XML syntax, and declares prelude and coda files.

```
<jsp-config>
  <jsp-property-group>
    <display-name>bookstore2</display-name>
    <url-pattern>/*.jsp</url-pattern>
    <el-ignored>false</el-ignored>
    <scripting-invalid>false</scripting-invalid>
    <is-xml>false</is-xml>
```

```
      <include-prelude>/template/prelude.jspf</include-prelude>
      <include-coda>/template/coda.jspf</include-coda>
    </jsp-property-group>
  </jsp-config>
```

Further Information

For further information on JavaServer Pages technology see:

- JavaServer Pages 2.0 Specification
  `http://java.sun.com/products/jsp/download.html#specs`
- The JavaServer Pages Web site
  `http://java.sun.com/products/jsp`

17

JavaServer Pages Standard Tag Library

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 set of tags. This standardization allows you to learn a single set of tags and use them on multiple JSP containers. Also, a standard tag library is more likely to have an optimized 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.

This chapter demonstrates JSTL through excerpts from the JSP version of the Duke's Bookstore application discussed in the previous chapter. It assumes that you are familiar with the material in the Using Custom Tags (page 669) 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 to replace the JavaBeans component database helper object 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 `<INSTALL>/jwstutorial13/examples/web/bookstore4/` directory created when you unzip the tutorial bundle (see About the Examples, page xxv). A sample `bookstore4.war` is provided in `<INSTALL>/jwstutorial13/examples/web/provided-wars/`. To build, package, deploy, and run the example:

1. Build and package the bookstore common files as described in Duke's Bookstore Examples (page 89).
2. In a terminal window, go to `<INSTALL>/jwstutorial13/examples/web/bookstore4/`.
3. Run `ant copy`. This target will copy files to the `<INSTALL>/jwstutorial13/examples/web/bookstore4/build/` directory.
4. Start Tomcat.
5. Perform all the operations described in Accessing Databases from Web Applications, page 90.
6. Run `ant install-config`. The `install-config` target notifies Tomcat that the new context is available. See Installing Web Applications (page 78).
7. Open the bookstore URL `http://localhost:8080/bookstore4/bookstore`.

See Troubleshooting (page 608) for help with diagnosing common problems.

Using JSTL

JSTL includes a wide variety of tags that fit into discrete functional areas. To reflect this, as well as to give each area its own namespace, JSTL is exposed as multiple tag libraries. The URIs for the libraries are:

- Core: `http://java.sun.com/jsp/jstl/core`
- XML: `http://java.sun.com/jsp/jstl/xml`
- Internationalization: `http://java.sun.com/jsp/jstl/fmt`
- SQL: `http://java.sun.com/jsp/jstl/sql`
- Functions: `http://java.sun.com/jsp/jstl/functions`

Table 17–1 summarizes these functional areas along with the prefixes used in this tutorial.

Table 17–1 JSTL Tags

Area	Subfunction	Prefix
Core	Variable Support	`c`
	Flow Control	
	URL Management	
	Miscellaneous	
XML	Core	`x`
	Flow Control	
	Transformation	
I18n	Locale	`fmt`
	Message formatting	
	Number and date formatting	
Database	SQL	`sql`
Functions	Collection length	`fn`
	String manipulation	

Thus, the tutorial references the JSTL core tags in JSP pages with the following `taglib`:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
     prefix="c" %>
```

In addition to declaring the tag libraries, tutorial examples access the JSTL API and implementation. In the Java WSDP, the JSTL TLDs `and` libraries are distributed in the archives standard.jar and jstl.jar in `<`*`JWSDP_HOME`*`>/jstl/lib/`.

This chapter does not explain every tag attribute, only the most commonly-used ones. Please refer to the TLD files contained in `appserver-jstl.jar` or the JSTL specification for a complete list of the JSTL tags and their attributes.If you copy these archives to the directory `<JWSDP_HOME>/common/lib`, they will automatically be loaded into the classpath of all Web applications running on Tomcat.

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 conditional tags employ this mode of collaboration.

Explicit collaboration happens when a tag exposes information to its environment. JSTL tags expose information as JSP EL variables; 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>
```

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`.

For situations where you want to use an EL variable exposed by a JSTL tag in an expression in the page's scripting language (see Chapter 19), you use the standard JSP element `jsp:useBean` to declare a scripting variable.

For example, `bookshowcart.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 exposed as an EL variable (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>
```

Core Tags

Table 17–2 summarizes the core tags, which 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 17–2 Core Tags

Area	Function	Tags	Prefix
Core	Variable Support	`remove` `set`	`c`

	Flow Control	`choose`
		  `when`
		  `otherwise`
		`forEach`
		`forTokens`
		`if`	

	URL Management	`import`
		  `param`
		`redirect`
		  `param`
		`url`
		  `param`	

	Miscellaneous	`catch` `out`	

Variable Support Tags

The `set` tag sets the value of an EL variable or the property of an EL variable in any of the JSP scopes (page, request, session, application). If the variable does not already exist, it is created.

The JSP EL variable or property 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 EL variable named `bookID` with the value of the request parameter named `Remove`:

```
<c:set var="bookId" value="${param.Remove}"/>
```

To remove an EL variable, you use the `remove` tag. When the bookstore JSP page `bookreceipt.jsp` is invoked, the shopping session is finished, so the `cart` session attribute is removed as follows:

```
<c:remove var="cart" scope="session"/>
```

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.quantity}
     </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 `bookcatalog.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>
      ${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">
    ${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 691), `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 doc="${xml}" var="booklist"
  scope="application" />
```

The `param` tag, analogous to the `jsp:param` tag (see jsp:param Element, page 674), can be used with `import` to specify request parameters.

In Session Tracking (page 635) 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, `bookcatalog.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="${bookId}" />
</c:url>
<p><strong><a href="${url}">
```

`The redirect` tag sends an HTTP redirect to the client. The `redirect` tag takes `param` subtags for including parameters in the returned URL.

Miscellaneous Tags

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 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.

The `out` tag evaluates an expression and outputs the result of the evaluation to the current `JspWriter` object. The syntax and attributes are

```
<c:out value="value" [escapeXml="{true|false}"]
  [default="defaultValue"] />
```

If the result of the evaluation is a `java.io.Reader` object, data is first read from the `Reader` object and then written into the current `JspWriter` object. The special processing associated with `Reader` objects improves performance when large amount of data must be read and then written to the response.

If `escapeXml` is true, the character conversions listed in Table 17–3 are applied:

Table 17–3 Character Conversions

Character	Character Entity Code
<	<
>	>
&	&
'	'
"	"

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 303).

Table 17–4 XML Tags

Area	Function	Tags	Prefix

XML	Core	`out`
		`parse`
		`set`
			`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 JSP 2.0 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 JSP expression language discussed in Implicit Objects (page 659). Table 17–5 shows some examples of using the scopes.

Table 17–5 Example XPath Expressions

XPath Expression	Result
`$sessionScope:profile`	The session-scoped EL variable named `profile`
`$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 of the bookstore database which is retrieved from another Web application. The directions for building and deploying this version of the application are in The Example JSP Document (page 538). A sample `bookstore5.war` is provided in `<INSTALL>/jwstutorial13/examples/web/provided-wars/`.

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 EL variable 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="${applicationScope:booklist == null}" >
  <c:import url="${initParam.booksURL}" var="xml" />
  <x:parse doc="${xml}" var="booklist" scope="application" />
</c:if>
```

The `set` and `out` tags parallel the behavior described in Variable Support Tags (page 686) and Miscellaneous Tags (page 690) for the XPath local expression language. The `set` tag evaluates an XPath expression and sets the result into a JSP EL variable specified by attribute `var`. 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 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 an EL variable. For example, `bookdetails.jsp` stores an EL variable containing a book price, which is later provided as the value of 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 686) for the XPath expression language.

The JSP page `bookcatalog.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="${bookId}" />
        <c:param name="Clear" value="0" />
      </c:url>
      <a href="${url}">
      <strong><x:out select="$book/title"/> 
      </strong></a></td>
    <td bgcolor="#ffffaa" rowspan=2>
      <c:set var="price">
        <x:out select="$book/price"/>
      </c:set>
      <fmt:formatNumber value="${price}" type="currency"/>
       
    </td>
    <td bgcolor="#ffffaa" rowspan=2>
    <c:url var="url" value="/catalog" >
      <c:param name="Add" value="${bookId}" />
    </c:url>
    <p><strong><a href="${url}"> 
      <fmt:message key="CartAdd"/> </a>
    </td>
  </tr>
  <tr>
    <td bgcolor="#ffffff">
      <fmt:message key="By"/> <em>
      <x:out select="$book/firstname"/> 
      <x:out select="$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 `doc`. If the `doc` 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

Chapter 23 covers how to design Web applications so that they conform 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, and formatting and parsing data elements such as numbers, currencies, dates, and times in a locale-sensitive or customized manner. Table 17–6 lists the tags.

Table 17–6 Internationalization Tags

Area	Function	Tags	Prefix
I18n	Setting Locale	`setLocale` `requestEncoding`	`fmt`

	Messaging	`bundle`
		`message`
		  `param`
		`setBundle`	

	Number and Date Formatting	`formatNumber`
		`formatDate`
		`parseDate`
		`parseNumber`
		`setTimeZone`
		`timeZone`	

JSTL i18n tags use a localization context to localize their data. A *localization context* contains a locale and a resource bundle instance. To specify the localization context, you define the context parameter `javax.servlet.jsp.jstl.fmt.localizationContext`, whose value can be a `javax.servlet.jsp.jstl.fmt.LocalizationContext` or a `String`. A `String` context parameter is interpreted as the name of a resource bundle basename. For the Duke's Bookstore application, the context parameter is the `String messages.BookstoreMessages`, which is set with `deploytool` in the Context tab of the WAR inspector. This setting can be overridden in a JSP page by using the JSTL `fmt:setBundle` tag. When a request is received, JSTL automatically sets the locale based on the value retrieved from the request header and chooses the correct resource bundle using the basename specified in the context parameter.

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.

message Tag

The `message` tag is used to output localized strings. The following tag from `bookcatalog.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 `bookshowcart.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 `bookreceipt.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="${now.time + 432000000}" />
<fmt:message key="ShipDate"/>
<fmt:formatDate value="${now}" type="date"
  dateStyle="full"/>.
```

SQL Tags

The JSTL SQL tags listed in Table 17–7 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	Prefix
Database		`setDataSource`	`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 of 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 `bookcatalog.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 book`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="${bookRow[7]}" />

      <c:if test="${item.quantity > inventory.quantity}">
        <c:set var="sufficientInventory" value="false" />
        <h3><font color="red" size="+2">
        <fmt:message key="OrderError"/>
        There is insufficient inventory for
        <i>${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"/>
        ${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="${books.rows}">
  <h2>${book.title}</h2>
   <fmt:message key="By"/> <em>${book.firstname}
  ${book.surname}</em>  
  (${book.year})<br>   <br>
  <h4><fmt:message key="Critics"/></h4>
  <blockquote>${book.description}</blockquote>
  <h4><fmt:message key="ItemPrice"/>:
  <fmt:formatNumber value="${book.price}" type="currency"/>
  </h4>
</c:forEach>
```

The following excerpt from `bookcatalog.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 `bookcatalog.jsp` to the versions in JavaServer Pages Technology (page 641) and Custom Tags in JSP Pages (page 705) 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="${bookRow[3]}" />
        <jsp:setProperty name="addedBook" property="price"
          value="${bookRow[4])}" />
        <jsp:setProperty name="addedBook" property="year"
          value="${bookRow[6]}" />
        <jsp:setProperty name="addedBook"
          property="description"
          value="${bookRow[7]}" />
        <jsp:setProperty name="addedBook" property="inventory"
          value="${bookRow[8]}" />
      </jsp:useBean>
      <% cart.add(bid, addedBook); %>
      ...
    </c:forEach>
```

Functions

Table 17–8 lists the JSTL functions

Table 17–8 Functions

Area	Function	Tags	Prefix
Functions	Collection length	`length`	fn

	String manipulation	`toUpperCase, toLowerCase`
		`substring, substringAfter, substringBefore`
		`trim`
		`replace`
		`indexOf, startsWith, endsWith,`
		`contains, containsIgnoreCase`
		`split, join`
		`escapeXml`	

While the `java.util.Collection` interface defines a `size` method, it does not conform to the JavaBeans design pattern for properties and cannot be accessed via the JSP expression language. The `length` function can be applied to any collection supported by the `c:forEach` and returns the length of the collection. When applied to a `String`, it returns the number of characters in the string.

For example, the `index.jsp` page of the `hello1` application introduced in Chapter 4 uses the `fn:length` function and `c:if` tag to determine whether to include a response page:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
  prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions"
  prefix="fn" %>
<html>
<head><title>Hello</title></head>
...
<input type="text" name="username" size="25">
<p></p>
<input type="submit" value="Submit">
<input type="reset" value="Reset">
</form>

<c:if test="${fn:length(param.username) > 0}" >
  <%@include file="response.jsp" %>
</c:if>
</body>
</html>
```

The rest of the JSTL functions are concerned with string manipulation:

- `toUpperCase`, `toLowerCase` - Changes the capitalization of a string.
- `substring`, `substringBefore`, `substringAfter` - Gets a subset of a string.
- `trim` - Trims whitespace from a string.
- `replace` - Replaces characters in a string.
- `indexOf`, `startsWith`, `endsWith`, `contains`, `containsIgnoreCase` - Checks if a string contains another string.
- `split` - Splits a string into an array.
- `join` - Joins a collection into a string.
- `escapeXml` - Escapes XML characters in a string.

Further Information

For further information on JSTL see:

- The JSTL 1.1 Specification. This chapter documents the maintenance release of the JSTL Specification.

`http://java.sun.com/products/jsp/jstl/index.html#specs`

- The JSTL Web site

  `http://java.sun.com/products/jsp/jstl`

18

Custom Tags in JSP Pages

THE standard JSP tags 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. 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 flow control. Custom tags increase productivity because they can be reused across more than one application.

Custom tags are distributed in a *tag library*, which defines a set of related custom tags and contains the objects that implement the tags. The object that implements a custom tag is called a *tag handler*. JSP technology defines two types of tag handlers: simple and classic. Simple tag handlers can only be used for tags that do not use scripting elements in attribute values or the tag body. Classic tag handlers must be used if scripting elements are required. Simple tag handlers are covered in this chapter and classic tag handlers are discussed in Chapter 19.

You can write simple tag handlers with the JSP language or with the Java language. A *tag file* is a source file containing a reusable fragment of JSP code that is translated into a simple tag handler by the Web container. Tag files can be used to develop custom tags that are presentation-centric or that can take advantage of existing tag libraries, or by page authors who do not know Java. For occasions when the flexibility of the Java programming language is needed to define the tag, JSP technology provides a simple API for developing a tag handler in the Java programming language.

This chapter assumes you are familiar with the material in Chapter 16, especially the section Using Custom Tags (page 669). 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 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.
- Pass variables back to the calling page.
- Access all the objects available to JSP pages.
- Communicate with each other. You can create and initialize a JavaBeans component, create a public EL variable that refers to that bean in one tag, and then use the bean in another tag.
- Be nested within one another and communicate via private variables.

The Example JSP Pages

This chapter describes the tasks involved in 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 645) rewritten to take advantage of several new custom tags:

- A `catalog` tag for rendering the book catalog
- A `shipDate` tag for rendering the ship date of an order
- A template library for ensuring a common look and feel among all screens and composing screens out of content chunks

The last section in the chapter, Examples (page 749), describes several tags in detail: a simplified iteration tag and the set of tags in the `tutorial-template` tag library.

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 18–1 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 Page column in Table 16–1).
- `Dispatcher`, a servlet, which processes requests and forwards to `template.jsp`.

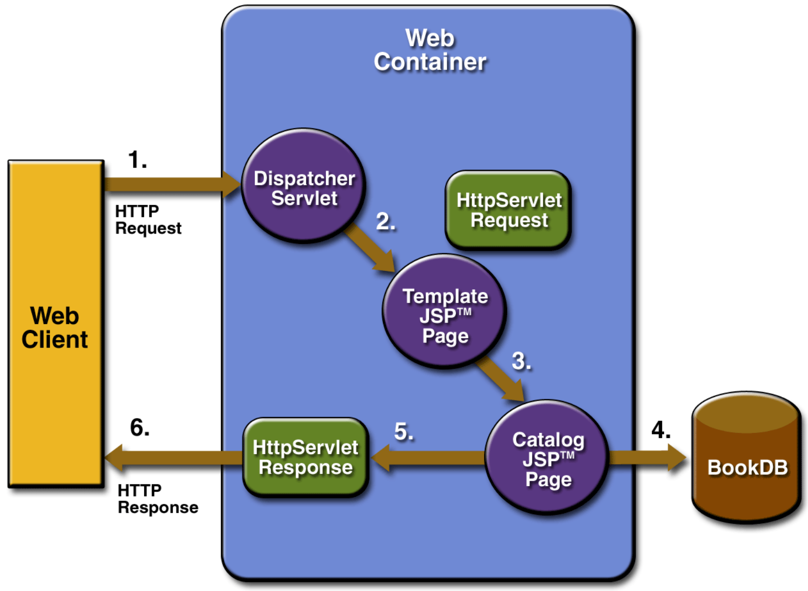


Figure 18–1 Request Flow Through Duke's Bookstore Components

The source code for the Duke's Bookstore application is located in the `<INSTALL>/jwstutorial13/examples/web/bookstore3/` directory created when you unzip the tutorial bundle (see About the Examples, page xxv). A sample `bookstore3.war` is provided in `<INSTALL>/jwstutorial13/examples/web/provided-wars/`. To build, package, deploy, and run the example:

1. Build and package the bookstore common files as described in Removing Web Applications (page 84).
2. In a terminal window, go to `<INSTALL>/jwstutorial13/examples/bookstore3/`.
3. Run `Ant build`. This target will spawn any necessary compilations and copy files to the `<INSTALL>/jwstutorial13/examples/web/bookstore3/build/` directory.
4. Start Tomcat.
5. Perform all the operations described in Accessing Databases from Web Applications, page 90.

6. Run `ant install-config`. The `install-config` target notifies Tomcat that the new context is available. See Installing Web Applications (page 78).
7. Open the bookstore URL `http://localhost:8080/bookstore3/bookstore`.

See Troubleshooting (page 608) for help with diagnosing common problems.

Types of Tags

JSP simple tags are invoked 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 /> or <tt:tag></tt:tag>
```

Tags with Attributes

A simple tag can have attributes. Attributes customize the behavior of a custom tag just as parameters customize the behavior of a method.

There are three types of attributes:

- Simple attributes
- Fragment attributes
- Dynamic attributes

Simple Attributes

Simple attributes are evaluated by the container prior to being passed to the tag handler. Simple attributes are listed in the start tag and have the syntax `attr="value"`. You can set a simple attribute value from a `String` constant, an EL expression, or with a `jsp:attribute` element (see jsp:attribute Element, page 711). The conversion process between the constants and expressions and attribute types follows the rules described for JavaBeans component properties in Setting JavaBeans Component Properties (page 666).

The Duke's Bookstore page `bookcatalog.jsp` calls the `catalog` tag which has two attributes. The first attribute, a reference to a book database object, is set by an EL expression. The second attribute, which sets the color of the rows in a table that represents the bookstore catalog, is set with a `String` constant.

```
<sc:catalog bookDB ="${bookDB}" color="#cccccc">
```

Fragment Attributes

A *JSP fragment* is a portion of JSP code passed to a tag handler that can be invoked as many times as needed. You can think of a fragment as a template that is used by a tag handler to produce customized content. Thus, unlike simple attributes which are evaluated by the container, fragment attributes are evaluated by tag handlers during tag invocation.

You declare an attribute to be a fragment by using the `fragment` attribute in a tag file `attribute` directive (see Declaring Tag Attributes in Tag Files, page 718) or by using the `fragment` subelement of the `attribute` TLD element (see Declaring Tag Attributes for Tag Handlers, page 736). You define the value of fragment attribute with a `jsp:attribute` element. When used to specify a fragment attribute, the body of the `jsp:attribute` element can only contain template text and standard and custom tags; it *cannot* contain scripting elements (see Chapter 19).

JSP fragments can be parametrized via expression language (EL) variables in the JSP code that composes the fragment. The EL variables are set by the tag handler, thus allowing the handler to customize the fragment each time it is invoked (see Declaring Tag Variables in Tag Files, page 719 and Declaring Tag Variables for Tag Handlers, page 737).

The `catalog` tag discussed earlier accepts two fragments: `normalPrice`, which is displayed for a product that's full price, and `onSale`, which is displayed for a product that's on sale.

```
<sc:catalog bookDB ="${bookDB}" color="#cccccc">
  <jsp:attribute name="normalPrice">
    <fmt:formatNumber value="${price}" type="currency"/>
  </jsp:attribute>
  <jsp:attribute name="onSale">
    <strike><fmt:formatNumber value="${price}"
      type="currency"/></strike><br/>
```

```
      <font color="red"><fmt:formatNumber value="${salePrice}"
        type="currency"/></font>
    </jsp:attribute>
  </sc:catalog>
```

The tag executes the `normalPrice` fragment, using the values for the `price` EL variable, if the product is full price. If the product is on sale, the tag executes the `onSale` fragment, using the `price` and `salePrice` variables.

Dynamic Attributes

A *dynamic attribute* is an attribute that is not specified in the definition of the tag. Dynamic attributes are primarily used by tags whose attributes are treated in a uniform manner, but whose names are not necessarily known at development time.

For example, this tag accepts an arbitrary number of attributes whose values are colors and outputs a bulleted list of the attributes colored according to the values:

```
<colored:colored color1="red" color2="yellow" color3="blue"/>
```

You can also set the value of dynamic attributes with an EL expression or using the `jsp:attribute` element.

jsp:attribute Element

The `jsp:attribute` element allows you to define the value of a tag attribute in the *body* of an XML element instead of in the value of an XML attribute.

For example, the Duke's Bookstore template page `screendefinitions.jsp` uses `jsp:attribute` to use the output of `fmt:message` to set the value of the `value` attribute of `tt:parameter`:

```
...
<tt:screen id="/bookcatalog">
  <tt:parameter name="title" direct="true">
    <jsp:attribute name="value" >
      <fmt:message key="TitleBookCatalog"/>
    </jsp:attribute>
  </tt:parameter>
  <tt:parameter name="banner" value="/template/banner.jsp"
    direct="false"/>
```

```
  <tt:parameter name="body" value="/bookcatalog.jsp"
    direct="false"/>
</tt:screen>
...
```

`jsp:attribute` accepts a `name` attribute and a `trim` attribute. The `name` attribute identifies which tag attribute is being specified. The optional `trim` attribute determines whether whitespace appearing at the beginning and end of the element body should be discarded or not. By default, the leading and trailing whitespace is discarded. The whitespace is trimmed when the JSP page is translated. If a body contains a custom tag that produces leading or trailing whitespace, that whitespace is preserved regardless of the value of the `trim` attribute.

An empty body is equivalent to specifying `""` as the value of the attribute.

The body of `jsp:attribute` is restricted according to the type of attribute being specified:

- For simple attributes that accept an EL expression, the body can be any JSP content.
- For simple attributes that do not accept an EL expression, the body can only contain template text.
- For fragment attributes, the body must not contain any scripting elements (See Chapter 19).

Tags with Bodies

A simple tag can contain custom and core tags, HTML text, and tag-dependent body content between the start and end tag.

In the following example, the Duke's Bookstore application page `bookshowcart.jsp` uses the JSTL `c:if` tag to print the body if the request contains a parameter named `Clear`:

```
<c:if test="${param.Clear}">
  <font color="#ff0000" size="+2"><strong>
  You just cleared your shopping cart!
  </strong><br> <br></font>
</c:if>
```

jsp:body Element

You can also specify the body of a simple tag explicitly using the `jsp:body` element. If one or more attributes are specified with the `jsp:attribute` element, then `jsp:body` is the only way to specify the body of the tag. If one or more `jsp:attribute` elements appear in the body of a tag invocation but you don't include a `jsp:body` element, the tag has an empty body.

Tags That Define Variables

A simple tag can define an EL variable that can be used within the calling page. In the following example, the `iterator` tag sets the value of the EL variable `departmentName` as it iterates through a collection of department names.

```
<tlt:iterator var="departmentName" type="java.lang.String"
    group="${myorg.departmentNames}">
  <tr>
    <td><a href="list.jsp?deptName=${departmentName}">
      ${departmentName}</a></td>
  </tr>
</tlt:iterator>
```

Communication Between Tags

Custom tags communicate with each other through shared objects. There are two types of shared objects: public and private.

In the following example, the `c:set` tag creates a public EL variable called `aVariable`, which is then reused by `anotherTag`.

```
<c:set var="aVariable" value="aValue" />
<tt:anotherTag attr1="${aVariable}" />
```

Nested tags can share private objects. In the next example, an object created by `outerTag` is available to `innerTag`. The inner tag retrieves its parent tag and then retrieves an object from the parent. Since the object is not named, the potential for naming conflicts is reduced.

```
<tt:outerTag>
  <tt:innerTag />
</tt:outerTag>
```

The Duke's Bookstore page `template.jsp` uses a set of cooperating tags that share public and private objects to define the screens of the application. These tags are described in A Template Tag Library (page 751).

Encapsulating Reusable Content using Tag Files

A tag file is a source file that contains a fragment of JSP code that is reusable as a custom tag. Tag files allow you to create custom tags using JSP syntax. Just as a JSP page gets translated into a servlet class and then compiled, a tag file gets translated into a tag handler and then compiled.

The recommended file extension for a tag file is `.tag`. As is the case with JSP files, the tag may be composed of a top file that includes other files that contain either a complete tag or a fragment of a tag file. Just as the recommended extension for a fragment of a JSP file is `.jspf`, the recommended extension for a fragment of a tag file is `.tagf`.

The following version of the Hello, World application introduced in Chapter 4 uses a tag to generate the response. The `response` tag, which accepts two attributes—a greeting string and a name—is encapsulated in `response.tag`:

```
<%@ attribute name="greeting" required="true" %>
<%@ attribute name="name" required="true" %>
<h2><font color="black">${greeting}, ${name}!</font></h2>
```

The highlighted line in `greeting.jsp` page invokes the `response` tag if the length of the `username` request parameter is greater than `0`:

```
<%@ taglib tagdir="/WEB-INF/tags" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
  prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions"
  prefix="fn" %>
<html>
<head><title>Hello</title></head>
<body bgcolor="white">
<img src="duke.waving.gif">
<c:set var="greeting" value="Hello" />
<h2>${greeting}, my name is Duke. What's yours?</h2>
<form method="get">
<input type="text" name="username" size="25">
<p></p>
```

```
<input type="submit" value="Submit">
<input type="reset" value="Reset">
</form>

<c:if test="${fn:length(param.username) > 0}" >
  <h:response greeting="${greeting}"
    name="${param.username}"/>
</c:if>
</body>
</html>
```

A sample `hello3.war` is provided in `<INSTALL>/jwstutorial13/examples/web/provided-wars/`. To build, package, deploy, and run the `hello3` application:

1. In a terminal window, go to `<INSTALL>/jwstutorial13/examples/web/hello3/`.
2. Run `ant build`. This target will spawn any necessary compilations and copy files to the `<INSTALL>/jwstutorial13/examples/web/hello3/build/` directory.
3. Start Tomcat.
4. Run `ant install`. The `install` target notifies Tomcat that the new context is available.
5. Open your browser to `http://localhost:8080/hello3`

Tag File Location

Tag files can be placed in one of two locations: in the `/WEB-INF/tags/` directory or subdirectory of a Web application or in a JAR file (see Packaged Tag Files, page 734) in the `/WEB-INF/lib/` directory of a Web application. Packaged tag files require a *tag library descriptor* (see Tag Library Descriptors, page 729), an XML document that contains information about a library as a whole and about each tag contained in the library. Tag files that appear in any other location are not considered tag extensions and are ignored by the Web container.

Tag File Directives

Directives are used to control aspects of tag file translation to a tag handler, specify aspects of the tag, attributes of the tag, and variables exposed by the tag. Table 18–1 lists the directives that you can use in tag files.

Table 18–1 Tag File Directives

Directive	Description
`taglib`	Identical to `taglib` directive (see Declaring Tag Libraries, page 670) for JSP pages.
`include`	Identical to `include` directive (see Reusing Content in JSP Pages, page 672) for JSP pages. Note that if the included file contains syntax unsuitable for tag files, a translation error will occur.
`tag`	Similar to the `page` directive in a JSP page, but applies to tag files instead of JSP pages. Like the `page` directive, a translation unit can contain more than one instance of the `tag` directive. All the attributes apply to the complete translation unit. However, there can be only one occurrence of any attribute/value defined by this directive in a given translation unit. With the exception of the `import` attribute, multiple attribute/value (re)definitions result in a translation error. Also used for declaring custom tag properties such as display name. See Declaring Tags (page 716).
`attribute`	Declares attributes of the custom tag defined in the tag file. See body-content Attribute (page 718).
`variable`	Declares an EL variable exposed by the tag to the calling page. See Declaring Tag Variables in Tag Files (page 719).

Declaring Tags

The `tag` directive is similar to the JSP page's `page` directive, but applies to tag files. Some of the elements in the `tag` directive appear in the `tag` element of a

TLD (see Declaring Tag Handlers, page 734). Table 18–2 lists the `tag` directive attributes.

Table 18–2 `tag` Directive Attributes

Attribute	Description
`display-name`	(optional) A short name that is intended to be displayed by tools. Defaults to the name of the tag file without the extension `.tag`.
`body-content`	(optional) Provides information on the content of the body of the tag. Can be either `empty`, `tagdependent`, or `scriptless`. A translation error will result if `JSP` or any other value is used. Defaults to `scriptless`. See body-content Attribute (page 718).
`dynamic-attributes`	(optional) Indicates whether this tag supports additional attributes with dynamic names. The value identifies a scoped attribute in which to place a `Map` containing the names and values of the dynamic attributes passed during invocation of the tag. A translation error results if the value of the `dynamic-attributes` of a `tag` directive is equal to the value of a `name-given` of a `variable` directive or the value of a `name` attribute of an `attribute` directive.
`small-icon`	(optional) Relative path, from the tag source file, of an image file containing a small icon that can be used by tools. Defaults to no small icon.
`large-icon`	(optional) Relative path, from the tag source file, of an image file containing a large icon that can be used by tools. Defaults to no large icon.
`description`	(optional) Defines an arbitrary string that describes this tag. Defaults to no description.
`example`	(optional) Defines an arbitrary string that presents an informal description of an example of a use of this action. Defaults to no example.
`language`	(optional) Carries the same syntax and semantics of the `language` attribute of the `page` directive.
`import`	(optional) Carries the same syntax and semantics of the `import` attribute of the `page` directive.

Table 18–2 `tag` Directive Attributes (Continued)

Attribute	Description
`pageEncoding`	(optional) Carries the same syntax and semantics of the `pageEncoding` attribute in the `page` directive.
`isELIgnored`	(optional) Carries the same syntax and semantics of the `isELIgnored` attribute of the `page` directive.

body-content Attribute

You specify the character of a tag's body content using the `body-content` attribute:

```
bodycontent="empty | scriptless | tagdependent"
```

You must declare the body content of tags that do not accept a body as `empty`. For tags that have a body there are two options. Body content containing custom and standard tags and HTML text is specified as `scriptless`. All other types of body content—for example, SQL statements passed to the query tag—is specified as `tagdependent`. If no attribute is specified, the default is `scriptless`.

Declaring Tag Attributes in Tag Files

You declare the attributes of a custom tag defined in a tag file with the `attribute` directive. A TLD has an analogous `attribute` element (see Declaring Tag Attributes for Tag Handlers, page 736). Table 18–3 lists the `attribute` directive attributes:

Table 18–3 `attribute` Directive Attributes

Attribute	Description
`description`	(optional) Description of the attribute. Defaults to no description.

Table 18–3 `attribute` Directive Attributes (Continued)

Attribute	Description
`name`	The unique name of the attribute being declared. A translation error results if more than one `attribute` directive appears in the same translation unit with the same `name`. A translation error results if the value of a `name` attribute of an `attribute` directive is equal to the value of `dynamic-attributes` attribute of a `tag` directive or the value of a `name-given` attribute of a `variable` directive.
`required`	(optional) Whether this attribute is required (`true`) or optional (`false`). Defaults to `false`.
`rtexprvalue`	(optional) Whether the attribute's value may be dynamically calculated at runtime by an expression. Defaults to `true`.
`type`	(optional) The runtime type of the attribute's value. Defaults to `java.lang.String`.
`fragment`	(optional) Whether this attribute is a fragment to be evaluated by the tag handler (true) or a normal attribute to be evaluated by the container prior to being passed to the tag handler. If this attribute is true: You do not specify the `rtexprvalue` attribute. The container fixes the `rtexprvalue` attribute at `true`. You do not specify the `type` attribute. The container fixes the `type` attribute at `javax.servlet.jsp.tagext.JspFragment`. Defaults to `false`.

Declaring Tag Variables in Tag Files

Tag attributes are used to customize tag behavior much like parameters are used to customize the behavior of object methods. In fact, using tag attributes and EL variables, is it possible to emulate various types of parameters—`IN`, `OUT`, and nested.

To emulate `IN` parameters, use tag attributes. A tag attribute is communicated between the calling page and the tag file when the tag is invoked. No further communication occurs between the calling page and tag file.

To emulate `OUT` or nested parameters, use EL variables. The variable is not initialized by the calling page, but set by the tag file. Each type of parameter is synchronized with the calling page at various points according to the scope of the variable. See Variable Synchronization (page 721) for details.

You declare an EL variable exposed by a tag file with the `variable` directive. A TLD has an analogous `variable` element (see Declaring Tag Variables for Tag Handlers, page 737). Table 18–4 lists the `variable` directive attributes:

Table 18–4 `variable` Directive Attributes

Attribute	Description
`description`	(optional) An optional description of this variable. Defaults to no description.
`name-given | name-from-attribute`	Defines an EL variable to be used in the page invoking this tag. Either `name-given` or `name-from-attribute` must be specified. If `name-given` is specified, the value is the name of the variable. If `name-from-attribute` is specified, the value is the name of an attribute whose (translation-time) value at of the start of the tag invocation will give the name of the variable. Translation errors arise in the following circumstances: 1. Specifying neither `name-given` or `name-from-attribute` or both. 2. If two `variable` directives have the same `name-given`. 3. If the value of `name-given` attribute of a `variable` directive is equal to the value of a `name` attribute of an `attribute` directive or the value of `dynamic-attributes` attribute of a `tag` directive.
`alias`	Defines a variable, local to the tag file, to hold the value of the EL variable. The container will synchronize this value with the variable whose name is given in `name-from-attribute`. Required when `name-from-attribute` is specified. A translation error results if used without `name-from-attribute`. A translation error results if the value of `alias` is the same as the value of a `name` attribute of an `attribute` directive or the `name-given` attribute of a `variable` directive.
`variable-class`	(optional) The name of the class of the variable. The default is `java.lang.String`.
`declare`	(optional) Whether the variable is declared or not. True is the default.

Table 18–4 `variable` Directive Attributes

Attribute	Description
`scope`	(optional) The scope of the variable. Can be either `AT_BEGIN`, `AT_END`, or `NESTED`. Defaults to `NESTED`.

Variable Synchronization

The Web container handles the synchronization of variables between a tag file and a calling page. Table 18–5 summarizes when and how each object is synchronized according to the object's scope.

Table 18–5 Variable Synchronization Behavior

	`AT_BEGIN`	`NESTED`	`AT_END`
Beginning of tag file	not synch.	save	not synch.
Before any fragment invocation via `jsp:invoke` or `jsp:doBody` (see Evaluating Fragments Passed to Tag Files, page 724)	tag→page	tag→page	not synch
End of tag file	tag→page	restore	tag→page

If `name-given` is used to specify the variable name, the name of the variable in the calling page and the name of the variable in the tag file are the same and are equal to the value of `name-given`.

The `name-from-attribute` and `alias` attributes of the `variable` directive can be used to customize the name of the variable in the calling page while using another name in the tag file. When using these attributes, the name of the variable in the calling page is set from the value of `name-from-attribute` at the time the tag was called. The name of the corresponding variable in the tag file is the value of `alias`.

Synchronization Examples

The following examples illustrate how variable synchronization works between a tag file and its calling page. All the example JSP pages and tag files reference the JSTL core tag library with the prefix `c`. The JSP pages reference a tag file located in `/WEB-INF/tags` with the prefix `my`.

AT_BEGIN Scope

In this example, the `AT_BEGIN` scope is used to pass the value of the variable named `x` to the tag's body and at the end of the tag invocation.

```
<%-- callingpage.jsp --%>
<c:set var="x" value="1"/>
${x} <%-- (x == 1) --%>
<my:example>
  ${x} <%-- (x == 2) --%>
</my:example>
${x} <%-- (x == 4) --%>

<%-- example.tag --%>
<%@ variable name-given="x" scope="AT_BEGIN" %>
${x} <%-- (x == null) --%>
<c:set var="x" value="2"/>
<jsp:doBody/>
${x} <%-- (x == 2) --%>
<c:set var="x" value="4"/>
```

NESTED Scope

In this example, the `NESTED` scope is used to make a variable named `x` available only to the tag's body. The tag sets the variable to `2` and this value is passed to the calling page before the body is invoked. Since the scope is `NESTED`, and the calling page also had a variable named x, its original value, 1, is restored when the tag completes.

```
<%-- callingpage.jsp --%>
<c:set var="x" value="1"/>
${x} <%-- (x == 1) --%>
<my:example>
  ${x} <%-- (x == 2) --%>
</my:example>
${x} <%-- (x == 1) --%>

<%-- example.tag --%>
<%@ variable name-given="x" scope="NESTED" %>
${x} <%-- (x == null) --%>
<c:set var="x" value="2"/>
<jsp:doBody/>
${x} <%-- (x == 2) --%>
<c:set var="x" value="4"/>
```

AT_END Scope

In this example, the AT_END scope is used to return a value to the page. The body of the tag is not affected.

```
<%-- callingpage.jsp --%>
<c:set var="x" value="1"/>
${x} <%-- (x == 1) --%>
<my:example>
  ${x} <%-- (x == 1) --%>
</my:example>
${x} <%-- (x == 4) --%>

<%-- example.tag --%>
<%@ variable name-given="x" scope="AT_END" %>
${x} <%-- (x == null) --%>
<c:set var="x" value="2"/>
<jsp:doBody/>
${x} <%-- (x == 2) --%>
<c:set var="x" value="4"/>
```

AT_BEGIN and name-from-attribute

In this example the AT_BEGIN scope is used to pass an EL variable to the tag's body, and make it available to the calling page at the end of the tag invocation.

The name of the variable is specified via the value of the attribute `var`. The variable is referenced by a local name, `result`, in the tag file.

```
<%-- callingpage.jsp --%>
<c:set var="x" value="1"/>
${x} <%-- (x == 1) --%>
<my:example var="x">
  ${x} <%-- (x == 2) --%>
  ${result} <%-- (result == null) --%>
  <c:set var="result" value="invisible"/>
</my:example>
${x} <%-- (x == 4) --%>
${result} <%-- (result == 'invisible') --%>

<%-- example.tag --%>
<%@ attribute name="var" required="true" rtexprvalue="false"%>
<%@ variable alias="result" name-from-attribute="var"
  scope="AT_BEGIN" %>
${x} <%-- (x == null) --%>
${result} <%-- (result == null) --%>
<c:set var="x" value="ignored"/>
<c:set var="result" value="2"/>
<jsp:doBody/>
${x} <%-- (x == 'ignored') --%>
${result} <%-- (result == 2) --%>
<c:set var="result" value="4"/>
```

Evaluating Fragments Passed to Tag Files

When a tag file is executed, the Web container passes it two types of fragments: fragment attributes and the tag body, which is implemented as a fragment. Recall from the discussion of fragment attributes that fragments are evaluated by the tag handler as opposed to the Web container. Within a tag file, you use the `jsp:invoke` element to evaluate a fragment attribute and the `jsp:doBody` element to evaluate a tag file body.

The result of evaluating either type of fragment is sent to the response or stored in an EL variable for later manipulation. To store the result of evaluating a fragment to an EL variable, you specify the `var` or `varReader` attributes. If `var` is specified, the container stores the result in an EL variable of type `String` with the name specified by `var`. If `varReader` is specified, the container stores the result in an EL variable of type `java.io.Reader` with the name specified by `varReader`. The `Reader` object can then be passed to a custom tag for further processing. A translation error occurs if both `var` and `varReader` are specified.

An optional `scope` attribute indicates the scope of the resulting variable. The possible values are `page` (default), `request`, `session`, or `application`. A translation error occurs if this attribute appears without specifying the `var` or `varReader` attribute.

Examples

Simple Attributes

The Duke's Bookstore `shipDate` tag, defined in `shipDate.tag`, is a custom tag with a simple attribute. The tag generates the date of a book order according to the type of shipping requested.

```
<%@ taglib prefix="sc" tagdir="/WEB-INF/tags" %>
<h3><fmt:message key="ThankYou"/> ${param.cardname}.</h3><br>
<fmt:message key="With"/>
<em><fmt:message key="${param.shipping}"/></em>,
<fmt:message key="ShipDateLC"/>
<sc:shipDate shipping="${param.shipping}" />
```

The tag determines the number of days until shipment from the `shipping` attribute passed to it by the page `bookreceipt.jsp`. From the days, the tag computes the ship date. It then formats the ship date.

```
<%@ attribute name="shipping" required="true" %>

<jsp:useBean id="now" class="java.util.Date" />
<jsp:useBean id="shipDate" class="java.util.Date" />
<c:choose>
  <c:when test="${shipping == 'QuickShip'}">
    <c:set var="days" value="2" />
  </c:when>
  <c:when test="${shipping == 'NormalShip'}">
    <c:set var="days" value="5" />
  </c:when>
  <c:when test="${shipping == 'SaverShip'}">
    <c:set var="days" value="7" />
  </c:when>
</c:choose>
<jsp:setProperty name="shipDate" property="time"
  value="${now.time + 86400000 * days}" />
<fmt:formatDate value="${shipDate}" type="date"
  dateStyle="full"/>.<br><br>
```

Simple and Fragment Attributes and Variables

The Duke's Bookstore catalog tag, defined in catalog.tag, is a custom tag with simple and fragment attributes and variables. The tag renders the catalog of a book database as an HTML table. The tag file declares that it sets variables named price and salePrice via variable directives. The fragment normalPrice uses the variable price and the fragment onSale uses the variables price and salePrice. Before the tag invokes the fragment attributes with the jsp:invoke element, the Web container passes values for the variables back to the calling page.

```
<%@ attribute name="bookDB" required="true"
  type="database.BookDB" %>
<%@ attribute name="color" required="true" %>
<%@ attribute name="normalPrice" fragment="true" %>
<%@ attribute name="onSale" fragment="true" %>

<%@ variable name-given="price" %>
<%@ variable name-given="salePrice" %>

<center>
<table>
<c:forEach var="book" begin="0" items="${bookDB.books}">
  <tr>
  <c:set var="bookId" value="${book.bookId}" />
  <td bgcolor="${color}">
     <c:url var="url" value="/bookdetails" >
        <c:param name="bookId" value="${bookId}" />
     </c:url>
     <a href="${url}"><
        strong>${book.title} </strong></a></td>
  <td bgcolor="${color}" rowspan=2>
  <c:set var="salePrice" value="${book.price * .85}" />
  <c:set var="price" value="${book.price}" />
  <c:choose>
     <c:when test="${book.onSale}" >
        <jsp:invoke fragment="onSale" />
     </c:when>
     <c:otherwise>
        <jsp:invoke fragment="normalPrice"/>
     </c:otherwise>
  </c:choose>

   </td>
```

```
...
</table>
</center>
```

The page `bookcatalog.jsp` invokes the `catalog` tag with simple attributes `bookDB`, which contains catalog data, and `color`, which customizes the coloring of the table rows. The formatting of the book price is determined by two fragment attributes—`normalPrice` and `onSale`—that are conditionally invoked by the tag according to data retrieved from the book database.

```
<sc:catalog bookDB ="${bookDB}" color="#cccccc">
  <jsp:attribute name="normalPrice">
    <fmt:formatNumber value="${price}" type="currency"/>
  </jsp:attribute>
  <jsp:attribute name="onSale">
    <strike>
    <fmt:formatNumber value="${price}" type="currency"/>
    </strike><br/>
    <font color="red">
    <fmt:formatNumber value="${salePrice}" type="currency"/>
    </font>
  </jsp:attribute>
</sc:catalog>
```

The screen produced by `bookcatalog.jsp` is shown in Figure 18–2. You can compare it to the version in Figure 16–2.

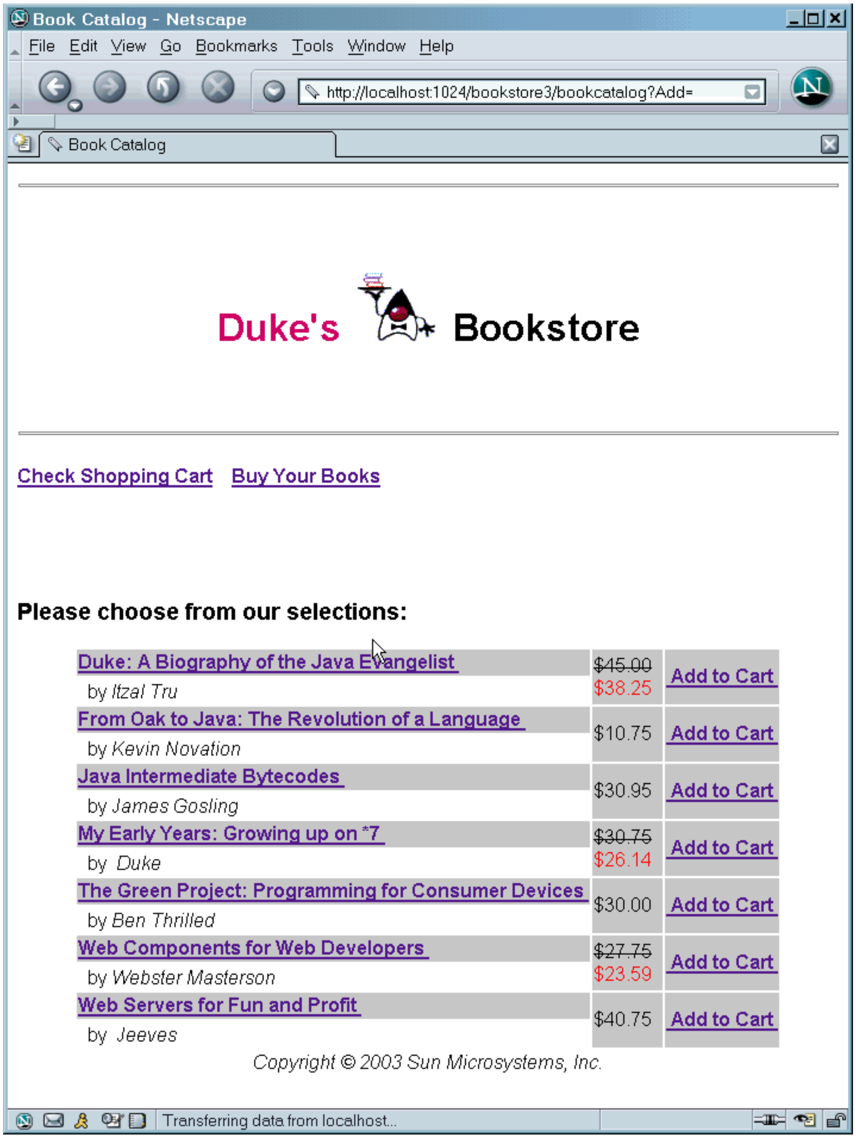


Figure 18–2 Book Catalog

Dynamic Attributes

The following code implements the tag discussed in Dynamic Attributes (page 711). An arbitrary number of attributes whose values are colors are stored in a `Map` named by the `dynamic-attributes` attribute of the `tag` directive. The JSTL `forEach` tag is used to iterate through the `Map` and the attribute keys and colored attribute values are printed in a bulleted list.

```
<%@ tag dynamic-attributes="colorMap"%>
<ul>
<c:forEach var="color" begin="0" items="${colorMap}">
  <li>${color.key} =
     <font color="${color.value}">${color.value}</font><li>
</c:forEach>
</ul>
```

Tag Library Descriptors

If you want to redistribute your tag files or implement your custom tags with tag handlers written in Java, you need to declare the tags in a tag library descriptor (TLD). A *tag library descriptor* (TLD) is an XML document that 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.

Tag library descriptor file names must have the extension `.tld` and must be packaged in the `/WEB-INF/` directory or subdirectory of the WAR file or in the `/META-INF/` directory or subdirectory of a tag library packaged in a JAR. If a tag is implemented as a tag file and is packaged in `/WEB-INF/tags/`or a subdirectory, a TLD will be automatically generated by the Web container, though you can provide one if you wish.

A TLD must begin with a root `taglib` element that specifies the schema and required JSP version:

```
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-
jsptaglibrary_2_0.xsd"
  version="2.0">
```

Table 18–6 lists the subelements of the `taglib` element:

Table 18–6 `taglib` Subelements

Element	Description
`description`	(optional) A string describing the use of the tag library.
`display-name`	(optional) Name intended to be displayed by tools.
`icon`	(optional) Icon that can be used by tools.
`tlib-version`	The tag library's version.
`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.
`validator`	See validator Element (page 731).
`listener`	See listener Element (page 731).
`tag-file | tag`	Declares the tag files or tags defined in the tag library. See Declaring Tag Files (page 731) and Declaring Tag Handlers (page 734). A tag library is considered invalid if a `tag-file` element has a `name` subelement with the same content as a `name` subelement in a `tag` element.
`function`	Zero or more EL functions (see Functions, page 662) defined in the tag library.
`tag-extension`	(optional) Extensions that provide extra information about the tag library for tools.

This section describes the top-level elements of TLDs. Subsequent sections describe how to declare tags defined in tag files, how to declare tags defined in tag handlers, and how to declare tag attributes and variables.

validator Element

This element defines an optional tag library validator that can be used to validate the conformance of any JSP page importing this tag library to its requirements. Table 18–7 lists the subelements of the `validator` element:

Table 18–7 validator Subelements

Element	Description
`validator-class`	The class implementing `javax.servlet.jsp.tagext.TagLibraryValidator`
`init-param`	(optional) Initialization parameters.

listener Element

A tag library can specify some classes that are event listeners (see Handling Servlet Life Cycle Events, page 609). 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.

Declaring Tag Files

Although not required for tag files, providing a TLD allows you to share the tag across more than one tag library and lets you import the tag library using a URI instead of the `tagdir` attribute.

tag-file TLD Element

A tag file is declared in the TLD with a `tag-file` element, whose subelements are listed in Table 18–8:

Table 18–8 `tag-file` Subelements

Element	Description
`description`	(optional) A description of the tag.
`display-name`	(optional) Name intended to be displayed by tools.
`icon`	(optional) Icon that can be used by tools.
`name`	The unique tag name.
`path`	Where to find the tag file implementing this tag, relative to the root of the Web application or the root of the JAR file for a tag library packaged in a JAR. This must begin with `/WEB-INF/tags/` if the tag file resides in the WAR, or `/META-INF/tags/` if the tag file resides in a JAR.
`example`	(optional) Informal description of an example use of the tag.
`tag-extension`	(optional) Extensions that provide extra information about the tag for tools.

Unpackaged Tag Files

Tag files placed in a subdirectory of `/WEB-INF/tags/` do not require a TLD file and don't have to be packaged. Thus, to create reusable JSP code, you simply create a new tag file and place the code inside of it.

The Web container generates an implicit tag library for each directory under and including `/WEB-INF/tags/`. There are no special relationships between subdirectories—they are allowed simply for organizational purposes. For example, the following Web application contains three tag libraries:

```
/WEB-INF/tags/
/WEB-INF/tags/a.tag
/WEB-INF/tags/b.tag
/WEB-INF/tags/foo/
/WEB-INF/tags/foo/c.tag
/WEB-INF/tags/bar/baz/
/WEB-INF/tags/bar/baz/d.tag
```

The implicit TLD for each library has the following values:

- `tlib-version` for the tag library. Defaults to 1.0.
- `short-name` is derived from the directory name. If the directory is `/WEB-INF/tags/`, the short name is simply `tags`. Otherwise, the full directory path (relative to the Web application) is taken, minus the `/WEB-INF/tags/` prefix. Then, all / characters are replaced with `-`, which yields the short name. Note that short names are not guaranteed to be unique.
- A `tag-file` element is considered to exist for each tag file, with the following sub-elements:
  - The `name` for each is the filename of the tag file, without the `.tag` extension.
  - The `path` for each is the path of the tag file, relative to the root of the Web application.

So, for the previous example, the implicit TLD for the `/WEB-INF/tags/bar/baz/` directory would be:

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <short-name>bar-baz</short-name>
  <tag-file>
    <name>d</name>
    <path>/WEB-INF/tags/bar/baz/d.tag</path>
  </tag-file>
</taglib>
```

Despite the existence of an implicit tag library, a TLD in the Web application can still create additional tags from the same tag files. To accomplish this, you add a `tag-file` element with a `path` that points to the tag file.

Packaged Tag Files

Tag files can be packaged in the `/META-INF/tags/` directory in a JAR file installed in the `/WEB-INF/lib/` directory of the Web application. Tags placed here are typically part of a reusable library of tags that can be easily used in any Web application.

Tag files bundled in a JAR require a tag library descriptor. Tag files that appear in a JAR but are not defined in a TLD are ignored by the Web container.

When used in a JAR file, the `path` subelement of the `tag-file` element specifies the full path of the tag file from the root of the JAR. Therefore, it must always begin with `/META-INF/tags/`.

Tag files can also be compiled into Java classes and bundled as a tag library. This is useful when you wish to distribute a binary version of the tag library without the original source. If you choose this form of packaging you must use a tool that produces portable JSP code that uses only standard APIs.

Declaring Tag Handlers

When tags are implemented with tag handlers written in Java, each tag in the library must be declared in the TLD with the `tag` element. The `tag` element contains the tag name, the class of its tag handler, information on the tag's attributes, and information on the variables created by the tag (see Tags That Define Variables, page 713).

Each attribute declaration contains an indication of whether the attribute is required, whether its value can be determined by request-time expressions, the type of the attribute, and whether the attribute is a fragment. Variable information can be given directly in the TLD or through a tag extra info class. Table 18–9 lists the subelements of the `tag` element:

Table 18–9 `tag` Subelements

Element	Description
`description`	(optional) A description of the tag.
`display-name`	(optional) name intended to be displayed by tools.
`icon`	(optional) Icon that can be used by tools.

Table 18–9 `tag` Subelements (Continued)

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 Declaring Tag Variables for Tag Handlers (page 737).
`body-content`	The body content type. See body-content Element (page 735).
`variable`	(optional) Declares an EL variable exposed by the tag to the calling page. See Declaring Tag Variables for Tag Handlers (page 737).
`attribute`	Declares attributes of the custom tag. See Declaring Tag Attributes for Tag Handlers (page 736).
`dynamic-attributes`	Whether the tag supports additional attributes with dynamic names. Defaults to `false`. If true, the tag handler class must implement the `javax.servlet.jsp.tagext.DynamicAttributes` interface.
`example`	(optional) Informal description of an example use of the tag.
`tag-extension`	(optional) Extensions that provide extra information about the tag for tools.

body-content Element

You specify the type of body that is valid for this tag with the `body-content` element. This element is used by the Web container to validate that a tag invocation has the correct body syntax and by page composition tools to assist the page author in providing a valid tag body. There are four possible values:

- `tagdependent`—The body of the tag is interpreted by the tag implementation itself, and is most likely in a different language, for example, embedded SQL statements.
- `JSP`—The body of the tag contains nested JSP syntax.
- `empty`—The body must be empty.
- `scriptless`—The body accepts only template text, EL expressions, and custom tags. No scripting elements are allowed.

Declaring Tag Attributes for Tag Handlers

For each tag attribute, you must specify whether the attribute is required, whether the value can be determined by an expression, optionally, the type of the attribute in an `attribute` element, and whether the attribute is a fragment. 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.For static values, the type is always `java.lang.String`. An attribute is specified in a TLD in an `attribute` element. Table 18–10 lists the subelements of the `attribute` element.

Table 18–10 `attribute` Subelements

Element	Description
`description`	(optional) A description of the attribute.
`name`	The unique name of the attribute being declared. A translation error results if more than one `attribute` element appears in the same tag with the same `name`.
`required`	(optional) Whether the attribute is required. The default is `false`.
`rtexprvalue`	(optional) Whether the attribute's value may be dynamically calculated at runtime by an EL expression. The default is `false`.
`type`	(optional) The runtime type of the attribute's value. Defaults to `java.lang.String` if not specified.
`fragment`	(optional) Whether this attribute is a fragment to be evaluated by the tag handler (true) or a normal attribute to be evaluated by the container prior to being passed to the tag handler. If this attribute is true: You do not specify the `rtexprvalue` attribute. The container fixes the `rtexprvalue` attribute at `true`. You do not specify the `type` attribute. The container fixes the `type` attribute at `javax.servlet.jsp.tagext.JspFragment`. Defaults to `false`.

If a tag attribute is not required, a tag handler should provide a default value.

The `tag` element for a tag that outputs its body if a test evaluates to true declares that the `test` attribute is required and that its value can be set by a runtime expression.

```
<tag>
  <name>present</name>
     <tag-class>condpkg.IfSimpleTag</tag-class>
  <body-content>scriptless</body-content>
  ...
  <attribute>
    <name>test</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  ...
</tag>
```

Declaring Tag Variables for Tag Handlers

The example described in Tags That Define Variables (page 713) defines an EL variable `departmentName`:

```
<tlt:iterator var="departmentName" type="java.lang.String"
     group="${myorg.departmentNames}">
  <tr>
    <td><a href="list.jsp?deptName=${departmentName}">
      ${departmentName}</a></td>
  </tr>
</tlt:iterator>
```

When the JSP page containing this tag is translated, the Web container generates code to synchronize the variable with the object referenced by the variable. To generate the code, the Web container requires certain information about the 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 (see TagExtraInfo Class, page 746). Using the `vari-`

able element is simpler, but less dynamic. With the variable element, the only aspect of the variable that you can specify at runtime is its name (via the name-from-attribute element). If you provide this information in a tag extra info class, you can also specify the type of the variable at runtime.

Table 18–11 lists the subelements of the variable element.

Table 18–11 variable Subelements

Element	Description
description	(optional) A description of the variable.
name-given | name-from-attribute	Defines an EL variable to be used in the page invoking this tag. Either name-given or name-from-attribute must be specified. If name-given is specified, the value is the name of the variable. If name-from-attribute is specified, the value is the name of an attribute whose (translation-time) value at of the start of the tag invocation will give the name of the variable. Translation errors arise in the following circumstances: 1. Specifying neither name-given or name-from-attribute or both. 2. If two variable elements have the same name-given.
variable-class	(optional) The fully qualified name of the class of the object. java.lang.String is the default.
declare	(optional) Whether the object is declared or not. True is the default.A translation error results if both declare and fragment are specified.
scope	(optional) The scope of the variable defined. Can be either AT_BEGIN, AT_END, or NESTED (see Table 18–12). Defaults to NESTED.

Table 18–12 Variable Availability

Value	Availability
NESTED	Between the start tag and the end tag.
AT_BEGIN	From the start tag until the scope of any enclosing tag. If there's no enclosing tag, then to the end of the page.

Table 18–12 Variable Availability (Continued)

Value	**Availability**
`AT_END`	After the end tag until the scope of any enclosing tag. If there's no enclosing tag, then to the end of the page.

You could define the following `variable` element for the `tlt:iterator` tag:

```
<tag>
  <variable>
    <name-given>var</name-given>
    <variable-class>java.lang.String</variable-class>
    <declare>true</declare>
    <scope>NESTED</scope>
  </variable>
</tag>
```

Programming Simple Tag Handlers

The classes and interfaces used to implement simple tag handlers are contained in the `javax.servlet.jsp.tagext` package. Simple tag handlers implement the `SimpleTag` interface. Interfaces can be used to take an existing Java object and make it a tag handler. For most newly created handlers, you would use the `SimpleTagSupport` classes as a base class.

The heart of a simple tag handler is a single method—`doTag`—which gets invoked when the end element of the tag is encountered. Note that the default implementation of the `doTag` method of `SimpleTagSupport` does nothing.

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 JSP context object (`javax.servlet.jsp.JspContext`). `JspContext` provides access to implicit objects. `PageContext` extends `JspContext` with servlet-specific behavior. A tag handler can retrieve all the other implicit objects (request, session, and application) accessible from a JSP page through these objects. If the tag is nested, a tag handler also has access to the handler (called the *parent*) associated with the enclosing tag.

Packaging Tag Handlers

Tag handlers 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 in the `WEB-INF/lib/` directory of the Web application.

How Is a Simple Tag Handler Invoked?

The `SimpleTag` interface defines the basic protocol between a simple tag handler and a JSP page's servlet. The JSP page's servlet invokes the `setJspContext`, `setParent`, and attribute setting methods before calling `doStartTag`.

```
ATag t = new ATag();
t.setJSPContext(...);
t.setParent(...);
t.setAttribute1(value1);
t.setAttribute2(value2);
...
t.setJspBody(new JspFragment(...))
t.doTag();
```

The following sections describe the methods that you need to develop for each type of tag introduced in Types of Tags (page 709).

Basic Tags

The handler for a basic tag without a body must implement the `doTag` method of the `SimpleTag` interface. The `doTag` method is invoked when the start tag is encountered.

The basic tag discussed in the first section,

```
<tt:basic />
```

would be implemented by the following tag handler:

```
public HelloWorldSimpleTag extends SimpleTagSupport {
  public void doTag() throws JspException, IOException {
    getJspContext().getOut().write("Hello, world.");
  }
}
```

Tags with Attributes

Defining Attributes in a Tag Handler

For each tag attribute, you must define a set method in the tag handler that conforms to the JavaBeans architecture conventions. For example, the tag handler for the JSTL `c:if` tag,

```
<c:if test="${Clear}">
```

contains the following method:

```
public void setTest(boolean test) {
  this.test = test;
}
```

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 `validate` method of a class derived from `TagExtraInfo`. This class is also used to provide information about variables defined by the tag (see TagExtraInfo Class, page 746).

The `validate` 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 `validate` method checks that the value of `attr1` is a valid Boolean value. Note that since the value of `attr1` can be computed at runtime, `validate` must check whether the tag user has chosen to provide a runtime value.

```
public class TwaTEI extends TagExtraInfo {
  public ValidationMessage[] validate(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 null;
      else
        return new ValidationMessage(data.getId(),
          "Invalid boolean value.");
    }
    else
      return null;
  }
}
```

Dynamic Attributes

Tag handlers that support dynamic attributes must declare that they do so in the `tag` element of the TLD (see Declaring Tag Handlers, page 734). In addition, your tag handler must implement the `setDynamicAttribute` method of the `DynamicAttributes` interface. For each attribute specified in the tag invocation that does not have a corresponding `attribute` element in the TLD, the Web container calls `setDynamicAttribute`, passing in the namespace of the attribute (or null if in the default namespace), the name of the attribute, and the value of the attribute. You must implement the `setDynamicAttribute` method to remember the names and values of the dynamic attributes so that they can be used later on when `doTag` is executed. If the `setDynamicAttribute` method an exception, the `doTag` method is not invoked for the tag, and the exception must be treated in the same manner as if it came from an attribute setter method.

The following implementation of `setDynamicAttribute` saves the attribute names and values in lists. Then, in the `doTag` method, the names and values are echoed to the response in an HTML list.

```
private ArrayList keys = new ArrayList();
private ArrayList values = new ArrayList();

public void setDynamicAttribute(String uri,
  String localName, Object value ) throws JspException {
```

```
        keys.add( localName );
        values.add( value );
    }

    public void doTag() throws JspException, IOException {
      JspWriter out = getJspContext().getOut();
      for( int i = 0; i < keys.size(); i++ ) {
          String key = (String)keys.get( i );
          Object value = values.get( i );
          out.println( "<li>" + key + " = " + value + "</li>" );
      }
    }
```

Tags with Bodies

A tag handler for a tag with a body is implemented differently depending on whether or not the tag handler needs to manipulate the body. A tag handler manipulates the body when it reads or modifies the contents of the body.

Tag Handler Does Not Manipulate the Body

If a tag handler needs to simply evaluate the body, it gets the body with the `getJspBody` method of `SimpleTag` and then evaluates the body with the `invoke` method.

The following tag handler accepts a `test` parameter and evaluates the body of the tag if the test evaluates to true. The body of the tag is encapsulated in a JSP fragment. If the test is true, the handler retrieves the fragment with the `getJspBody` method. The `invoke` method directs all output to a supplied writer or to the `JspWriter` returned by the `getOut` method of the `JspContext` associated with the tag handler if the writer is `null`.

```
public class IfSimpleTag extends SimpleTagSupport {
  private boolean test;
  public void setTest(boolean test) {
    this.test = test;
  }
  public void doTag() throws JspException, IOException {
    if(test){
      getJspBody().invoke(null);
    }
  }
}
```

Tag Handler Manipulates the Body

If the tag handler needs to manipulate the body, the tag handler must capture the body in a StringWriter. The invoke method directs all output to a supplied writer. Then the modified body is written to the JspWriter returned by the getOut method of the JspContext. Thus, a tag that converts its body to upper case could be written as follows:

```
public class SimpleWriter extends SimpleTagSupport {
  public void doTag() throws JspException, IOException {
    StringWriter sw = new StringWriter();
    jspBody.invoke(sw);
    jspContext().
      getOut().println(sw.toString().toUpperCase());
  }
}
```

Tags That Define Variables

Similar communication mechanisms exist for communication between JSP page and tag handlers as for JSP pages and tag files.

To emulate IN parameters, use tag attributes. A tag attribute is communicated between the calling page and the tag handler when the tag is invoked. No further communication occurs between the calling page and tag handler.

To emulate OUT or nested parameters, use variables with availability AT_BEGIN, AT_END, or NESTED. The variable is not initialized by the calling page, but set by the tag handler.

For AT_BEGIN availability, the variable is available in the calling page from the start tag until the scope of any enclosing tag. If there's no enclosing tag, then the variable is available to the end of the page. For AT_END availability, the variable is available in the calling page after the end tag until the scope of any enclosing tag. If there's no enclosing tag, then the variable is available to the end of the page. For nested parameters, the variable is available in the calling page between the start tag and the end tag.

When you develop a tag handler you are responsible for creating and setting the object referenced by the variable into a context accessible from the page. You do this by using the JspContext().setAttribute(name, value) or JspContext.setAttribute(name, value, scope) method. You retrieve the page context with the getJspContext method of SimpleTag.

Typically, an attribute passed to the custom tag specifies the name of the variable and the value of the variable is dependent on another attribute. For example, the `iterator` tag retrieves the name of the variable from the `var` attribute and determines the value of the variable from a computation performed on the `group` attribute.

```
public void doTag() throws JspException, IOException {
  if (iterator == null)
    return;
  while (iterator.hasNext()) {
    getJspContext().setAttribute(var, iterator.next());
    getJspBody().invoke(null);
  }
}
public void setVar(String var) {
  this.var = var;
}
public void setGroup(Collection group) {
  this.group = group;
  if(group.size() > 0)
    iterator = group.iterator();
}
```

The scope that an variable can have is summarized in Table 18–13. The scope constrains the accessibility and lifetime of the object.

Table 18–13 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 in the same Web application	The life of the application

TagExtraInfo Class

In Declaring Tag Variables for Tag Handlers (page 737) we discussed how to provide information about tag variable in the tag library descriptor. Here we describe another approach: defining a tag extra info class. You define a tag extra info class by extending the class `javax.servlet.jsp.tagext.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 of type `javax.servlet.jsp.tagext.TagData` 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 an EL variable's name and class.

The following example demonstrates how to provide information about the variable created by the `iterator` tag in a tag extra info class. Since the name (`var`) and class (`type`) of the 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 variable `var` to be used only within the tag body, the scope of the object is set to be `NESTED`.

```
package iterator;
public class IteratorTei 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("var"),
      type,
      true,
      VariableInfo.NESTED)
    };
  }
}
```

The fully qualified name of the tag extra info class defined for an EL variable must be declared in the TLD in the `tei-class` subelement of the `tag` element. Thus, the `tei-class` element for `IteratorTei` would be as follows:

```
<tei-class>
  iterator.IteratorTei
</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 `SimpleTagSupport.findAncestorWithClass(from, class)` or the `SimpleTagSupport.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 `connectionId` has been set. If the `connectionId` 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 SimpleTagSupport {
  public int doTag() throws JspException {
    String cid = getConnectionId();
    Connection connection;
    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 cid="con01" ... >
  ...
</tt:connection>
<tt:query id="balances" connectionId="con01">
  SELECT account, balance FROM acct_table
    where customer_number = ?
  <tt:param value="${requestScope.custNumber}" />
</tt:query>

<tt:connection ... >
  <tt:query cid="balances">
    SELECT account, balance FROM acct_table
    where customer_number = ?
    <tt:param value="${requestScope.custNumber}" />
  </tt:query>
</tt:connection>
```

The TLD for the tag handler indicates that the `connectionId` attribute is optional with the following declaration:

```
<tag>
  ...
  <attribute>
    <name>connectionId</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 `iterator` tag retrieves objects from a collection stored in a JavaBeans component and assigns them to an EL variable. This tag is a very simplified example of the an iterator tag. Web applications requiring such functionality should use the JSTL `forEach` tag, which is discussed in Iterator Tags (page 688). The body of the tag retrieves information from the variable. While elements remain in the collection, the `iterator` tag causes the body to be reevaluated.

JSP Page

The `index.jsp` page invokes the `iterator` tag to iterate through a collection of department names. Each item in the collection is assigned to the `departmentName` variable.

```
<%@ taglib uri="/tlt" prefix="tlt" %>
<html>
  <head>
  <title>Departments</title>
  </head>
  <body bgcolor="white">
  <jsp:useBean id="myorg" class="myorg.Organization"/>
  <table border=2 cellspacing=3 cellpadding=3>
    <tr>
      <td><b>Departments</b></td>
    </tr>
  <tlt:iterator var="departmentName" type="java.lang.String"
      group="${myorg.departmentNames}">
    <tr>
      <td><a href="list.jsp?deptName=${departmentName}">
        ${departmentName}</a></td>
    </tr>
  </tlt:iterator>
  </table>
  </body>
</html>
```

Tag Handler

The tag handler passes the current element of the group back to the page in an EL variable called `var`, which is accessed using the expression language in the calling page. After the variable is set, the body is evaluated with the `invoke` method.

```
public void doTag() throws JspException, IOException {
  if (iterator == null)
    return;
  while (iterator.hasNext()) {
    getJspContext().setAttribute(var, iterator.next());
    getJspBody().invoke(null);
  }
}
public void setVar(String var) {
  this.var = var;
}
public void setGroup(Collection group) {
  this.group = group;
  if(group.size() > 0)
    iterator = group.iterator();
}
```

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 Pages

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" prefix="tt" %>
<%@ page errorPage="/template/errorinclude.jsp" %>
<%@ include file="/template/screendefinitions.jsp" %>
<html>
<head>
<title>
<tt:insert definition="bookstore" parameter="title"/>
</title>
</head>
<body  bgcolor="#FFFFFF">
  <tt:insert definition="bookstore" parameter="banner"/>
<tt:insert definition="bookstore" parameter="body"/>
<center><em>Copyright &copy; 2002 Sun Microsystems, Inc. </
em></center>
</body>
</html>
```

`screendefinitions.jsp` creates a screen definition based on a request attribute `selectedScreen`:

```
<tt:definition name="bookstore"
screen="${requestScope
  ['javax.servlet.forward.servlet_path']}">
  <tt:screen id="/bookstore">
  <tt:parameter name="title" value="Duke's Bookstore"
    direct="true"/>
  <tt:parameter name="banner" value="/template/banner.jsp"
    direct="false"/>
  <tt:parameter name="body" value="/bookstore.jsp"
    direct="false"/>
  </tt:screen>
  <tt:screen id="/bookcatalog">
  <tt:parameter name="title" direct="true">
      <jsp:attribute name="value" >
        <fmt:message key="TitleBookCatalog"/>
      </jsp:attribute>
    </tt:parameter>
    <tt:parameter name="banner" value="/template/banner.jsp"
    direct="false"/>
    <tt:parameter name="body" value="/bookcatalog.jsp"
    direct="false"/>
  </tt:screen>
  ...
</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. Then `Dispatcher` performs business logic based on the request URL, which updates model objects. Finally, the servlet dispatches the request to `template.jsp`:

```
public class Dispatcher extends HttpServlet {
  public void doGet(HttpServletRequest request,
    HttpServletResponse response) {
    String bookId = null;
    BookDetails book = null;
    String clear = null;
    BookDBAO bookDBAO =
      (BookDBAO)getServletContext().
        getAttribute("bookDBAO");
```

```
HttpSession session = request.getSession();
String selectedScreen = request.getServletPath();
ShoppingCart cart = (ShoppingCart)session.
   getAttribute("cart");
if (cart == null) {
   cart = new ShoppingCart();
   session.setAttribute("cart", cart);
}
request.setAttribute("selectedScreen",
   request.getServletPath());
if (selectedScreen.equals("/bookcatalog")) {
   bookId = request.getParameter("Add");
   if (!bookId.equals("")) {
      try {
         book = bookDBAO.getBookDetails(bookId);
         if ( book.getOnSale() ) {
            double sale = book.getPrice() * .85;
            Float salePrice = new Float(sale);
            book.setPrice(salePrice.floatValue());
         }
         cart.add(bookId, book);
      } catch (BookNotFoundException ex) {
         // not possible
      }
   }
} else if (selectedScreen.equals("/bookshowcart")) {
   bookId =request.getParameter("Remove");
   if (bookId != null) {
      cart.remove(bookId);
   }
   clear = request.getParameter("Clear");
   if (clear != null && clear.equals("clear")) {
      cart.clear();
   }
} else if (selectedScreen.equals("/bookreceipt")) {
// Update the inventory
   try {
      bookDBAO.buyBooks(cart);
   } catch (OrderException ex) {
      request.setAttribute("selectedScreen",
         "/bookOrderError");
   }
}
try {
   request.
      getRequestDispatcher(
      "/template/template.jsp").
      forward(request, response);
```

```
      } catch(Exception ex) {
         ex.printStackTrace();
      }
   }

   public void doPost(HttpServletRequest request,
      HttpServletResponse response) {
      request.setAttribute("selectedScreen",
         request.getServletPath());
      try {
         request.
            getRequestDispatcher(
            "/template/template.jsp").
            forward(request, response);
      } catch(Exception ex) {
         ex.printStackTrace();
      }
   }
}
```

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 private objects. `DefinitionTag` creates a public object named `bookstore` that is used by `InsertTag`.

In `doTag`, `DefinitionTag` creates a private 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. These parameters are loaded when the body of the definition tag, which contains nested `screen` and `parameter` tags, is invoked. `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 a public `Definition` object.

```
public int doTag() {
   try {
      screens = new HashMap();
      getJspBody().invoke(null);
      Definition definition = new Definition();
      PageContext context = (PageContext)getJspContext();
      ArrayList params = (ArrayList) screens.get(screenId);
      Iterator ir = null;
      if (params != null) {
         ir = params.iterator();
```

```
            while (ir.hasNext())
               definition.setParam((Parameter)ir.next());
         // put the definition in the page context
         context.setAttribute(definitionName, definition,
            context.APPLICATION_SCOPE);
         }
      }
```

The table of screen definitions is filled in by `ScreenTag` and `ParameterTag` from text provided as attributes to these tags. Table 18–14 shows the contents of the screen definitions hash table for the Duke's Bookstore application.

Table 18–14 Screen Definitions

Screen Id	Title	Banner	Body
`/bookstore`	`Duke's Bookstore`	`/banner.jsp`	`/bookstore.jsp`
`/bookcatalog`	`Book Catalog`	`/banner.jsp`	`/bookcatalog.jsp`
`/bookdetails`	`Book Description`	`/banner.jsp`	`/bookdetails.jsp`
`/bookshowcart`	`Shopping Cart`	`/banner.jsp`	`/bookshowcart.jsp`
`/bookcashier`	`Cashier`	`/banner.jsp`	`/bookcashier.jsp`
`/bookreceipt`	`Receipt`	`/banner.jsp`	`/bookreceipt.jsp`

If the URL passed in the request is `/bookstore`, the `Definition` contains the items from the first row of Table 18–14:

Table 18–15 Definition for URL `/bookstore`

Title	Banner	Body
`Duke's Bookstore`	`/banner.jsp`	`/bookstore.jsp`

The parameters for the URL `/bookstore` are shown in Table 18–16. The parameters specify 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 18–16 Parameters for the URL `/bookstore`

Parameter Name	Parameter Value	isDirect
`title`	`Duke's Bookstore`	`true`
`banner`	`/banner.jsp`	`false`
`body`	`/bookstore.jsp`	`false`

`InsertTag` inserts parameters of the screen definition into the response. In the `doTag` method, it retrieves the definition object from the page context and then 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 void doTag() throws JspTagException {
  Definition definition = null;
  Parameter parameter = null;
  boolean directInclude = false;
  PageContext context = (PageContext)getJspContext();

    // get the definition from the page context
  definition = (Definition)context.getAttribute(
    definitionName, context.APPLICATION_SCOPE);
    // get the parameter
  if (parameterName != null && definition != null)
    parameter = (Parameter)
      definition.getParam(parameterName);

  if (parameter != null)
    directInclude = parameter.isDirect();

  try {
    // if parameter is direct, print to out
    if (directInclude && parameter  != null)
      context.getOut().print(parameter.getValue());
    // if parameter is indirect,
         include results of dispatching to page
    else {
```

```
            if ((parameter != null) &&
               (parameter.getValue() !=  null))
            context.include(parameter.getValue());
         }
      } catch (Exception ex) {
            throw new JspTagException(ex.getMessage());
      }
   }
```

19

Scripting in JSP Pages

JSP scripting elements allow you to use Java programming language statements in your JSP pages. Scripting element are typically used to create and access objects, define methods, and manage the flow of control. Many tasks that require the use of scripts can be eliminated by using custom tag libraries, in particular the JSP Standard Tag Library. 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. Nevertheless, there may be some circumstances that require its use.

There are three ways to create and use objects in scripting elements:

- 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 612) are created and used in *scriptlets* and *expressions*.

This chapter briefly describes the syntax and usage of JSP scripting elements.

The Example JSP Pages

This chapter illustrates JSP scripting elements using a version of the `hello2` example introduced in Chapter 4—`webclient`—that accesses a Web service. To build, package, deploy, and run the `webclient` example:

1. Build and deploy the JAX-RPC Web service `MyHelloService` described in Creating a Web Service with JAX-RPC (page 459).
2. In a terminal window, go to `<INSTALL>/jwstutorial13/examples/jaxrpc/webclient/`.
3. Run `ant build`. This target will spawn any necessary compilations and copy files to the `<INSTALL>/jwstutorial13/examples/jaxrpc/webclient/build/` directory.
4. Start Tomcat.
5. Run `ant install`. The `install` target notifies Tomcat that the new context is available.
6. Open your browser to `http://localhost:8080/webclient/greeting`

Note: The example assumes that the Java WSDP runs on the default port, 8080. If you have changed the port, you must update the port number in the file `<INSTALL>/jwstutorial13/examples/jaxrpc/webclient/response.jsp` before building and running the examples.

Using Scripting

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 the `language` attribute of the `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 `import` attribute of the `page` directive:

```
<%@ page import="fully_qualified_classname, packagename.*" %>
```

The `webclient` JSP page `response.jsp` imports the classes needed to access the JAX-RPC stub class and the Web service client classes with the following `page` directive:

```
<%@ page import="javax.xml.rpc.Stub,webclient.*" %>
```

Disabling Scripting

By default, scripting in JSP pages is valid. Since scripting can make pages difficult to maintain, some JSP page authors or page authoring groups may want to follow a methodology where scripting elements are not allowed.

You can invalidate scripting for a group of JSP pages by setting the `scripting-invalid` element of a JSP property group to `true`. For information on how to define a group of JSP pages, see Setting Properties for Groups of JSP Pages (page 677). When scripting is invalid, scriptlets, scripting expressions, and declarations will produce a translation error if present in any of the pages in the group. Table 19–1 summarizes the scripting settings and their meanings:

Table 19–1 Scripting Settings

JSP Configuration	Scripting Encountered
unspecified	Valid
`false`	Valid
`true`	Translation Error

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.

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.

For example, an older version of the Duke's Bookstore application retrieved the object that accesses the bookstore database from the context and stored a reference to the object in the variable `bookDBAO` in the `jspInit` method.The variable definition and the initialization and finalization methods `jspInit` and `jspDestroy` were defined in a declaration:

```
<%!
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 was removed from service, the `jspDestroy` method released the `BookDBAO` variable.

```
<%!
public void jspDestroy() {
  bookDBAO = null;
}
%>
```

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.

In the Web service version of the `hello2` application, `greeting.jsp` contains a scriptlet to retrieve the request parameter named `username` and test whether it is empty. If the `if` statement evaluates to true, the response page is included. Since the `if` statement opens a block, the HTML markup would be followed by a scriptlet that closes the block.

```
<%
  String username = request.getParameter("username");
  if ( username != null && username.length() > 0 ) {
%>
  <%@include file="response.jsp" %>
<%
  }
%>
```

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.

In the Web service version of the `hello2` application, `response.jsp` contains the following scriptlet which creates a JAX-RPC stub, sets the endpoint on the stub, and then invokes the sayHello method on the stub, passing the user name retrieved from a request parameter:

```
<%
     String resp = null;
     try {
        Stub stub = (Stub)(new
           MyHelloService_Impl().getHelloIFPort());
        stub._setProperty(
           javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,
           "http://localhost:8080/hello-jaxrpc/hello");
        HelloIF hello = (HelloIF)stub;
        resp =
           hello.sayHello(request.getParameter("username"));
     } catch (Exception ex) {
           resp = ex.toString();
     }
%>
```

A scripting expression is then used to insert the value of resp into the output stream:

```
<h2><font color="black"><%= resp %>!</font></h2>
```

Programming Tags That Accept Scripting Elements

Tag that accept scripting elements in attribute values or the body cannot be programmed as simple tags; they must be implemented as classic tags. The following sections describe the TLD elements and JSP tag extension API specific to classic tag handlers. All other TLD elements are the same as for simple tags.

TLD Elements

You specify the character of a classic tag's body content using the body-content element:

```
<body-content>empty | JSP | tagdependent</body-content>
```

You must declare the body content of tags that do not have a body as empty. For tags that have a body, there are two options. Body content containing custom and core tags, scripting elements, and HTML text is categorized as JSP. All other types of body content—for example, SQL statements passed to the query tag—would be labeled tagdependent.

Tag Handlers

Classic tag handlers are written with the Java language and implement either the Tag, IterationTag, 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.

The classes and interfaces used to implement classic tag handlers are contained in the javax.servlet.jsp.tagext package. Classic tag handlers implement either the Tag, IterationTag, or BodyTag interface. Interfaces can be used to take an existing Java object and make it a tag handler. For newly created classic tag 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 element 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 element of a custom tag is encountered, the handler's doEndTag method is invoked for all but simple tags. Additional methods are invoked in between when a tag handler needs to manipulate the body of the tag. For further information, see Tags with Bodies (page 765). In order to provide a tag handler implementation, you must implement the methods, summarized in Table 19–2, that are invoked at various stages of processing the tag.

Table 19–2 Tag Handler Methods

Tag Type	Interface	Methods
Basic	Tag	doStartTag, doEndTag
Attributes	Tag	doStartTag, doEndTag, setAttribute1,...,N, release
Body	Tag	doStartTag, doEndTag, release

Table 19–2 Tag Handler Methods (Continued)

Tag Type	Interface	Methods
Body, iterative evaluation	`IterationTag`	`doStartTag, doAfterBody, doEndTag, release`
Body, manipulation	`BodyTag`	`doStartTag, doEndTag, release, doInitBody, doAfterBody`

A tag handler has access to an API that allows it to communicate with the JSP page. The entry points to the API are two objects: the JSP context (`javax.servlet.jsp.JspContext`) for simple tag handlers and the page context (`javax.servlet.jsp.PageContext`) for classic tag handlers. `JspContext` provides access to implicit objects. `PageContext` extends `JspContext` with HTTP-specific behavior. A tag handler can retrieve all the other implicit objects (request, session, and application) accessible from a JSP page through these objects. In addition, 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.

How Is a Classic 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_AGAIN we
// iterate body evaluation
...
t.doAfterBody();
t.doEndTag();
out = pageContext.popBody();
t.release();
```

Tags with Bodies

A tag handler for a tag with a body is implemented differently depending on whether or not the tag handler needs to manipulate the body. A tag handler manipulates the body when it reads or modifies the contents of the body.

Tag Handler Does Not Manipulate the Body

If the tag handler does not need to manipulate the body, the tag handler should implement the `Tag` interface. If the tag handler implements the `Tag` interface and 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. The tag handler should return `EVAL_BODY_AGAIN` `doAfterBody` method if it determines that the body needs to be evaluated again.

Tag Handler Manipulates the Body

If the tag handler needs to manipulate the body, the tag handler must implement `BodyTag` (or be derived from `BodyTagSupport`).

When a tag handler implements the `BodyTag` interface, it must implement the `doInitBody` and the `doAfterBody` methods. These methods manipulate 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. `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_AGAIN`; otherwise, `doAfterBody` should return `SKIP_BODY`.

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;
      }
   }
```

release Method
A tag handler should reset its state and release any private resources in the `release` method.

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 `connectionId` has been set. 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 {
  public int doStartTag() throws JspException {
    String cid = getConnectionId();
    Connection connection;
    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 cid="con01" ... >
  ...
</tt:connection>
<tt:query id="balances" connectionId="con01">
  SELECT account, balance FROM acct_table
    where customer_number = ?
  <tt:param value="${requestScope.custNumber}" />
</tt:query>

<tt:connection ... >
  <tt:query cid="balances">
    SELECT account, balance FROM acct_table
    where customer_number = ?
    <tt:param value="${requestScope.custNumber}" />
  </tt:query>
</tt:connection>
```

The TLD for the tag handler indicates that the `connectionId` attribute is optional with the following declaration:

```
<tag>
  ...
  <attribute>
    <name>connectionId</name>
    <required>false</required>
  </attribute>
</tag>
```

Tags That Define Variables

The mechanisms for defining EL variables in classic tags are similar to those described in Chapter 18. You must declare the variable in a `variable` element of the TLD or in a tag extra info class. You use `PageContext().setAttribute(name, value)` or `PageContext.setAttribute(name, value, scope)` methods in the tag handler to create or update an association between a name accessible in the page context and the object that is the value of the variable. For classic tag handlers, Table 19–3 illustrates how the availability of a variable affects when you may want to set or update the variable's value.

Table 19–3 Scripting Variable Availability

Value	Availability	In Methods
`NESTED`	Between the start tag and the end tag	`doStartTag`, `doInitBody`, and `doAfterBody`.
`AT_BEGIN`	From the start tag until the end of the page	`doStartTag`, `doInitBody`, `doAfterBody`, and `doEndTag`.
`AT_END`	After the end tag until the end of the page	`doEndTag`

An EL variable defined by a custom tag can also be accessed in a scripting expression. For example, the Web service described in the previous section could be encapsulated in a custom tag that returns the response in an EL variable named by the `var` attribute and then `var` could be accessed in a scripting expression as follows:

```
<ws:hello var="response"
     name="<%=request.getParameter("username")%>" />
<h2><font color="black"><%= response %>!</font></h2>
```

Remember that in situations where scripting is not allowed

- In a tag body where the `body-content` is declared as `scriptless`
- In a page where scripting is specified to be invalid

you wouldn't be able to access the EL variable in a scriptlet or expression. Instead, you would have to use the JSP expression language to access the variable.

20

JavaServer Faces Technology

JAVASERVER Faces technology is a server-side user interface framework for Java technology-based Web applications.

The main components of JavaServer Faces technology are:

- An API for: representing UI components and managing their state; handling events, server side validation, and data conversion; defining page navigation; supporting internationalization and accessibility; and providing extensibility for all of these features.
- A JavaServer Pages (JSP) custom tag library for expressing UI components within a JSP page.

This well-defined programming model and UI component tag library significantly ease the burden of building and maintaining Web applications with server-side UIs. With minimal effort, you can:

- Wire client-generated events to server-side application code
- Map UI components on a page to server-side data
- Construct a UI with reusable and extensible components.
- Save and restore UI state beyond the life of server requests

As shown in Figure 20–1, the user interface you create with JavaServer Faces technology (represented by `myUI` in the graphic) runs on the server and renders back to the client.

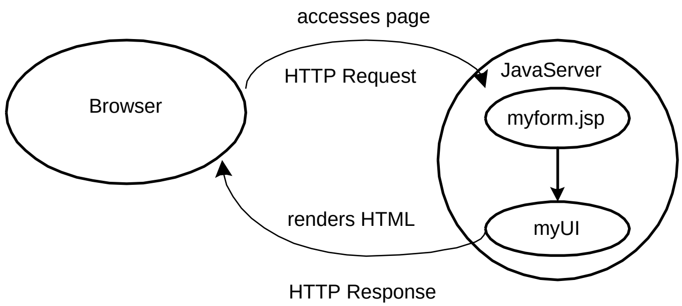


Figure 20–1 The UI Runs on the Server

The JSP page, `myform.jsp`, expresses the user interface components with custom tags defined by JavaServer Faces technology. The UI for the Web application (represented by `myUI` in the figure) manages the objects referenced by the JSP page. These objects include:

- The component objects that map to the tags on the JSP page
- The event listeners, validators, and converters that are registered on the components
- The objects that encapsulate the data and application-specific functionality of the components

JavaServer Faces Technology Benefits

One of the greatest advantages of JavaServer Faces technology is that it offers a clean separation between behavior and presentation. Web applications built with JSP technology partially achieve this separation. However, a JSP application cannot map HTTP requests to component-specific event handling or manage UI elements as stateful objects on the server. JavaServer Faces technology allows you to build Web applications that implement finer-grained separation of behavior and presentation traditionally offered by client-side UI architectures.

The separation of logic from presentation also allows each member of a Web application development team to focus on their piece of the development process, and provides a simple programming model to link the pieces together. For example, Page Authors with no programming expertise can use JavaServer Faces technology UI component tags to link to application code from within a Web page without writing any scripts.

Another important goal of JavaServer Faces technology is to leverage familiar UI-component and Web-tier concepts without limiting you to a particular scripting technology or markup language. While JavaServer Faces technology includes a JSP custom tag library for representing components on a JSP page, the JavaServer Faces technology APIs are layered directly on top of the Servlet API. This layering of APIs enables several important application use-cases such as: using another presentation technology besides JSP pages, creating your own custom components directly from the component classes, and generating output for different client devices.

Most importantly, JavaServer Faces technology provides a rich architecture for managing component state, processing component data, validating user input, and handling events.

What is a JavaServer Faces Application?

For the most part, JavaServer Faces applications are just like any other Java Web application. They run in a Java Servlet container, and they typically contain:

- JavaBeans components containing application-specific functionality and data
- Event listeners
- Pages, such as JSP pages
- Server-side helper classes, such as database-access beans

In addition to these items, a JavaServer Faces application also has:

- A custom tag library for rendering UI components on a page
- A custom tag library for representing event handlers, validators, and other actions.
- UI components represented as stateful objects on the server
- *backing bean*s, which define properties and functions for UI components
- Validators, converters, and event handlers
- An application configuration file for configuring application resources

A typical JavaServer Faces application that is using JSP pages that render HTML must include a custom tag library that defines the tags representing UI components and a custom tag library for representing other core actions, such as validators and event handlers. Both of these tag libraries are provided by the JavaServer Faces implementation.

The component tag library eliminates the need to hard-code UI components in HTML or another markup language, resulting in completely reusable components. And, the core tag library makes it easy to register events, validators, and other actions on the components.

The component tag library can be the `html_basic` tag library included with the JavaServer Faces implementation, or you can define your own tag library that renders custom components or renders output other than HTML.

Another important advantage of JavaServer Faces applications is that the UI components on the page are represented as stateful objects on the server. This allows the application to manipulate the component state and wire client-generated events to server-side code.

Finally, JavaServer Faces technology allows you to convert and validate data on individual components and report any errors before the server-side data is updated.

This chapter provides more detail on each of these features.

Framework Roles

Because of the division of labor enabled by the JavaServer Faces technology design, JavaServer Faces application development and maintenance can proceed quickly and easily. The members of a typical development team are those listed below. In many teams, individual developers play more than one of these roles, however, it is still useful to consider JavaServer Faces technology from a variety of perspectives based on primary responsibility.

- **Page Authors**, who use a markup language, like HTML, to author pages for Web applications. When using the JavaServer Faces technology framework, page authors will most likely use the tag library exclusively.
- **Application Developers**, who program the objects, the event handlers, the validators, and the page navigation. Application developers can also provide the extra helper classes.
- **Component Writers**, who have user-interface programming experience and prefer to create custom components using a programming language. These people can create their own components directly from the compo-

nent classes, or they can extend the standard components provided by JavaServer Faces technology.

- **Tools Vendors**, who provide tools that leverage JavaServer Faces technology to make building server-side user interfaces even easier.

The primary users of JavaServer Faces technology will be page authors and application developers. This tutorial is written with these two customers in mind. The next section walks through a simple application, explaining which piece of the application the page author and the application developer develops.

Chapter 22 covers the responsibilities of a component writer.

A Simple JavaServer Faces Application

This section describes the process of developing a simple JavaServer Faces application. You'll see what features a typical JavaServer Faces application contains, and what part each role has in developing the application.

Steps in the Development Process

Developing a simple JavaServer Faces application requires performing these tasks:

- Create the pages using the UI component and core tags
- Define page navigation in the application configuration file
- Develop the backing beans
- Add managed bean declarations to the application configuration file

These tasks can be done simultaneously or in any order. However, the people performing the tasks will need to communicate during the development process. For example, the page author needs to know the names of the objects in order to access them from the page.

The example used in this section is the `guessNumber` application, located in the *<JSF_HOME>*`/samples` directory. It asks you to guess a number between 0 and 10, inclusive. The second page tells you if you guessed correctly. The example also checks the validity of your input. Figure 20–2 shows what the first page looks like.

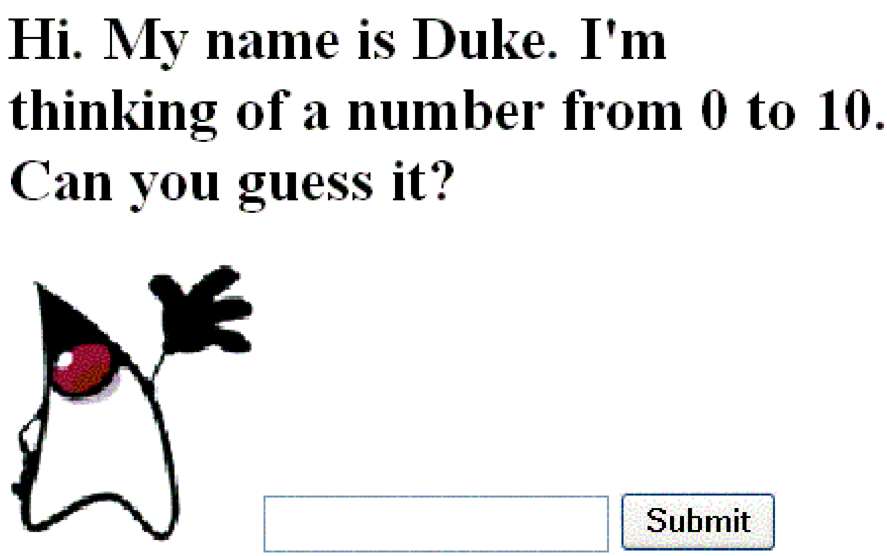


Figure 20–2 The greeting.jsp page of the guessNumber application

To deploy and execute this example, follow the instructions in Running the Examples on the Java WSDP Container (page 803).

Creating the Pages

Creating the pages is the page author's responsibility. This task involves laying out UI components on the pages, mapping the components to beans, and adding other core tags.

Here is the `greeting.jsp` page from the `guessNumber` application:

```
<HTML>
<HEAD> <title>Hello</title> </HEAD>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
```

```
<body bgcolor="white">
<f:view>
  <h:form id="helloForm" >
    <h2>Hi. My name is Duke.  I'm thinking of a number from
    <h:output_text value="#{UserNumberBean.minimum}"/> to
    <h:output_text value="#{UserNumberBean.maximum}"/>.
    Can you guess it?</h2>
    <h:graphic_image id="waveImg" url="/wave.med.gif" />
    <h:input_text id="userNo"
      value="#{UserNumberBean.userNumber}"
      validator="#{UserNumberBean.validate}"/>
    <h:command_button id="submit" action="success"
      value="Submit" />
    <p>
    <h:messages style="color: red;
      font-family: 'New Century Schoolbook', serif;
      font-style: oblique;
      text-decoration: overline" for="userNo"/>
  </h:form>
</f:view>
</HTML>
```

This page demonstrates a few important features that you will use in most of your JavaServer Faces applications. These features are described in the following sub-sections.

User Interface Component Model (page 790) includes a table that lists all of the component tags included with JavaServer Faces technology. Using the JavaServer Faces Tag Libraries (page 813) discusses the tags in more detail.

The form Tag

The `form` tag represents an input form, which allows the user to input some data and submit it to the server, usually by clicking a button. The tags representing the components that comprise the form are nested in the `form` tag. These tags are `h:input_text`, `h:command_button`, and `h:message`.

The input_text Tag

The `input_text` tag represents a text field component, into which the user enters a number. The instance of this tag included in `greeting.jsp` has three attributes: `id`, `value`, and `validator`.

The optional `id` attribute corresponds to the ID of the component object represented by this tag. If you don't include an `id` attribute, the JavaServer Faces implementation will generate one for you. See Using the JavaServer Faces Tag Libraries (page 813) for more information.

The `value` attribute binds the `userNo` component value to the bean property, `UserNumberBean.userNumber`, which holds the data entered into the text field. A page author can also bind a component instance to a property using the tag's `component` attribute.

The `validator` attribute refers to a method of the `UserNumberBean` that validates the input, making sure that it is an `Integer` in the range specified by `UserNumberBean`.

See Backing Bean Management (page 799) for more information on creating beans, binding to bean properties referencing bean methods, and configuring beans.

The command_button Tag

The `command_button` tag represents the button used to submit the data entered in the text field. The `action` attribute specifies an outcome that helps the navigation mechanism decide which page to open next. Define Page Navigation (page 778) discusses this further.

The messages Tag

The `messages` tag will display an error message if the data entered in the field does not comply with the rules specified by the validator. The error message displays wherever you place the `messages` tag on the page. The style attribute allows you to specify the formatting style for the message text. The `for` attribute refers to the component whose value failed validation, in this case the `userNo` component represented by the `input_text` tag in the `greeting.jsp` page.

Define Page Navigation

Defining page navigation involves determining which page to go to after the user clicks a button or a hyperlink. The JavaServer Faces navigation model is explained in Navigation Model (page 798). Navigating Between Pages (page 894) explains how to define the navigation rules for an entire application.

Navigation for the application is defined in the application configuration file, using a powerful rule-based system.

Here are the navigation rules defined for the guessNumber example:

```
<navigation-rule>
  <from-view-id>/greeting.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/response.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
<navigation-rule>
  <from-view-id>/response.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/greeting.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Each navigation-rule defines how to get from one page (specified in the from-view-id element) to the other pages of the application. The navigation-rule elements can contain any number of navigation-case elements, each of which defines the page to open next (defined by to-view-id) based on a logical outcome (defined by from-outcome).

The outcome can be defined by the action attribute of the UICommand component that submits the form, as it is in the guessNumber example:

```
<h:command_button id="submit" action="success"
  value="Submit" />
```

The outcome can also come from the return value of a method in a backing bean. This method performs some processing to determine the outcome. One example is that a method can check if the password the user entered on the page matches the one on file. If it does, the method could return "success"; otherwise, it might return "failure". An outcome of "failure" might result in the logon page being reloaded. An outcome of "success" might result in the page displaying the user's credit card activity opening. If you want the outcome to be returned by a method on a bean, you must refer to the method using the *JavaServer Faces expression language* (JSF EL) syntax with the action attribute, for example:

```
<h:command_button id="submit"
  action="#{userNumberBean.getOrderStatus}" value="Sumbit" />
```

To learn more about how navigation works and how to define navigation rules, see Navigation Model (page 798) and Navigating Between Pages (page 894).

Develop the Beans

Developing beans is one responsibility of the application developer. The page author and the application developer—if they are two different people—will need to work in tandem to make sure that the component tags refer to the proper UI component properties, ensure that the properties have the acceptable types, and take care of other such details.

A typical JavaServer Faces application couples a backing bean with each page in the application. The backing bean defines properties and methods that are associated with the UI components used on the page. Each backing bean property is bound to either a component instance or its value.

A backing bean can also define a set of methods that perform functions for the component, such as validating the component's data, handling events that the component fires, and performing processing associated with navigation when the component is activated.

The page author binds a component's value to a bean property by referring to the property with the component tag's `value` attribute. The page author binds a component instance to a bean property by referring to the property with the component tag's `binding` attribute.

Here is the `UserNumberBean` backing-bean property that maps to the data for the `userNo` component:

```
Integer userNumber = null;
...
public void setUserNumber(Integer user_number) {
  userNumber = user_number;
}
public Integer getUserNumber() {
  return userNumber;
}
public String getResponse() {
  if(userNumber != null &&
     userNumber.compareTo(randomInt) == 0) {
       return "Yay! You got it!";
```

```
        } else {
            return "Sorry, "+userNumber+" is incorrect.";
        }
    }
```

As you can see, this bean property is just like any other bean property: It has a set of accessor methods and a private data field. This means that you can reference beans you've already written from your JavaServer Faces pages.

A property can be any of the basic primitive and reference types, depending on what kind of component it references. This includes any of the `Number`, `java.math.BigDecimal` and `java.math.BigInteger` types. JavaServer Faces technology will automatically convert the data to the type specified by the bean property. See Writing Component Properties (page 838) for information on which types are accepted by which component tags.

You can also apply a converter to a component to convert the components value to a type not supported by the component. See Performing Data Conversions (page 878) for more information on applying a converter to a component.

In addition to binding components and their values to backing bean properties using component tag attributes, the page author can also refer to a backing bean from a component tag. For example, the `validator` attribute of the `input-text` tag in the `greeting.jsp` page refers to the `validate` method of the `UserNumberBean` backing bean. The `validate` method validates the user input and queues error messages onto the `userNo` component if the input is invalid. See Backing Bean Management (page 799) for more information on referencing methods from a component tag.

In this case, you could also use the standard `LongRangeValidator` instead of providing your own validation in the backing-file bean. See Performing Validation (page 864) for more information.

Adding Managed Bean Declarations

After developing the backing beans to be used in the application, you need to add declarations for them in the application configuration file so that the JavaServer Faces implementation can automatically create a new instance of the bean whenever it is needed.

The task of adding managed bean declarations to the application configuration file can be done by any member of the development team. Here is a managed bean declaration for UserNumberBean:

```
<managed-bean>
  <managed-bean-name>UserNumberBean</managed-bean-name>
  <managed-bean-class>
    guessNumber.UserNumberBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>minimum</property-name>
    <property-class>java.lang.Long</property-class>
    <value>0</value>
  </managed-property>
  <managed-property>
    <property-name>maximum</property-name>
    <property-class>java.lang.Long</property-class>
    <value>10</value>
  </managed-property>
</managed-bean>
```

One output_text tag on the greeting.jsp page binds its component to the minimum property of UserNumberBean. The other output_text tag binds its component to the maximum property of UserNumberBean. Here are these tags again:

```
<h:output_text value="#{UserNumberBean.minimum}"/>
<h:output_text value="#{UserNumberBean.maximum}"/>
```

As shown in the tags, the part of the expression before the "." matches the name defined by the managed-bean-name element corresponding to the proper managed-bean declaration from the application configuration file. The part of the expression after the "." matches the name defined by the property-name element corresponding to the proper managed-bean declaration.

Notice that the managed-property tags configure the minimum and maximum properties with values. These values are set when the bean is initialized, which happens when it is first referenced from a page.

Also notice that the configuration file does not configure the userNumber property. This is because the userNumber property is not initialized with a value when the bean is initialized.

The JavaServer Faces implementation processes this file on application startup time and initializes the `UserNumberBean` and stores it in session scope if no instance exists. The bean is then available for all pages in the application. For more information, see Backing Bean Management (page 799) and Application Configuration (page 801).

The Lifecycle of a JavaServer Faces Page

The lifecycle of a JavaServer Faces page is similar to that of a JSP page: The client makes an HTTP request for the page, and the server responds with the page translated to HTML. However, because of the extra features that JavaServer Faces technology offers, the lifecycle provides some additional services to process a page.

A JavaServer Faces page is represented by a tree of UI components, called a *view*. When a client makes a request for the page, the lifecycle starts. During the lifecycle, the JavaServer Faces implementation must build the view while considering state saved from the previous postback. When the client performs a postback of the page, the JavaServer Faces implementation must perform several tasks, such as validate the data input of components in the view and convert input data to types specified on the server side. The JavaServer Faces implementation performs all of these tasks as a series of steps in the lifecycle.

Which steps in the lifecycle are executed depends on whether or not the request originated from a JavaServer Faces application and whether or not the response is generated with the rendering phase of the JavaServer Faces lifecycle. This section first explains the different lifecycle scenarios. It then explains each of these lifecycle phases using the `guessNumber` example.

Request Processing Lifecycle Scenarios

A JavaServer Faces application supports two different kinds of responses and two different kinds of requests:

- Faces Response: A servlet response that was created by the execution of the Render Response (page 789) phase of the request processing lifecycle.
- Non-Faces Response: A servlet response that was not created by the execution of the *Render Response* phase. An example is a JSP page that does not incorporate JavaServer Faces components.
- Faces Request: A servlet request that was sent from a previously generated Faces Response. An example is a form submit from a JavaServer Faces user interface component, where the request URI identifies the JavaServer Faces component tree to use for processing the request.
- Non-Faces Request: A servlet request that was sent to an application component, such as a servlet or JSP page, rather than directed to a JavaServer Faces component tree.

These different requests and responses result in three possible lifecycle scenarios that can exist for a JavaServer Faces application:

- **Scenario 1: Non-Faces Request Generates Faces Response**

  An example of this scenario is when clicking a hyperlink on an HTML page opens a page containing JavaServer Faces components. To render a Faces Response from a Non-Faces Request, an application must provide a mapping to the `FacesServlet` in the URL to the page containing JavaServer Faces components. The `FacesServlet` accepts incoming requests and passes them to the lifecycle implementation for processing. Section Identifying the Servlet for Lifecycle Processing (page 808) describes how to provide a mapping to the `FacesServlet`. When generating a Faces Response, the application must create a new view, store it in the `FacesContext`, acquire object references needed by the view, and call `FacesContext.renderResponse`, which forces immediate rendering of the view by skipping to the Render Response (page 789) phase.

- **Scenario 2: Faces Request Generates Non-Faces Response**

  Sometimes a JavaServer Faces application might need to redirect to a different Web application resource or generate a response that does not contain any JavaServer Faces components. In these situations, the developer must skip the rendering phase (Render Response, page 789) by calling `FacesContext.responseComplete`. The `FacesContext` contains all of the information associated with a particular Faces Request. This method can be invoked during the Apply Request Values (page 787), Process Validations (page 787), or Update Model Values (page 788) phases.

- **Scenario 3: Faces Request Generates Faces Response**

  This is the most common scenario for the lifecycle of a JavaServer Faces application. It is also the scenario represented by the standard request processing lifecycle described in the next section. This scenario involves JavaServer Faces components submitting a request to a JavaServer Faces application utilizing the `FacesServlet`. Because the request has been handled by the JavaServer Faces implementation, no additional steps are required by the application to generate the response. All listeners, validators and converters will automatically be invoked during the appropriate phase of the standard lifecycle, which the next section describes.

Standard Request Processing Lifecycle

The standard request processing lifecycle represents scenario 3, described in the previous section. Most users of JavaServer Faces technology won't need to concern themselves with the request processing lifecycle. However, knowing that JavaServer Faces technology properly performs the processing of a page, a developer of JavaServer Faces applications doesn't need to worry about rendering problems associated with other UI framework technologies. One example involves state changes on individual components. If the selection of a component such as a checkbox effects the appearance of another component on the page, JavaServer Faces technology will handle this event properly and will not allow the page to be rendered without reflecting this change.

Figure 20–3 illustrates the steps in the JavaServer Faces request-response lifecycle.

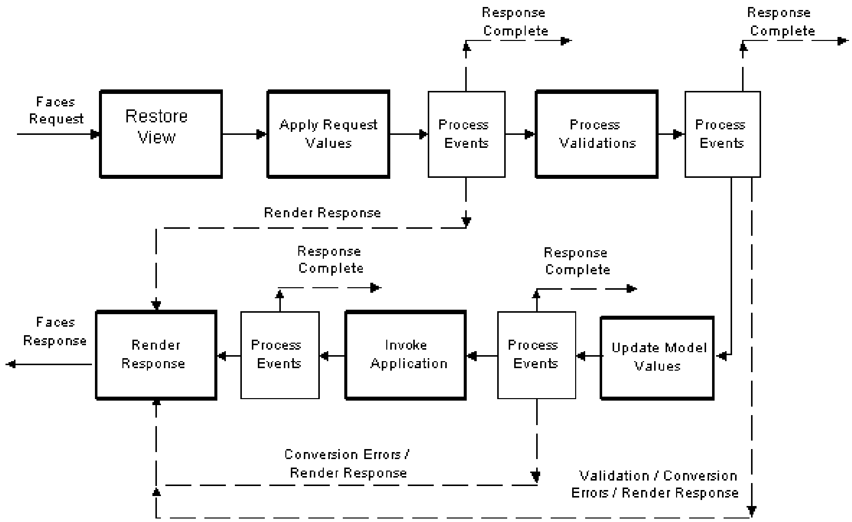


Figure 20–3 JavaServer Faces Request-Response Lifecycle

Restore View

When a request for a JavaServer Faces page is made, such as when clicking on a link or a button, the JavaServer Faces implementation begins the *Restore View* phase.

During this phase, the JavaServer Faces implementation builds the view of the JavaServer Faces page, wires up event handlers and validators, and saves the view in the `FacesContext`. The `FacesContext` instance contains all of the information needed to process a single request. All of the application's component tags, event handlers, converters, and navigators have access to the `FacesContext` instance.

If the page is being requested for the first time, the JavaServer Faces implementation creates an empty view during this phase. The empty view will be populated when the JSP page is processed during the lifecycle.

If the page has already been requested, a view corresponding to this page already exists. During this phase, the JavaServer Faces implementation restores the view with the state information saved on the client or the server.

The view for the `greeting.jsp` page of the `guessNumber` example would have the UIView component at the root of the tree, with `helloForm` as its child and the rest of the JavaServer Faces UI components as children of `helloForm`.

Apply Request Values

Once the component tree is restored, each component in the tree extracts its new value from the request parameters with its `decode` method. The value is then stored locally on the component. If the conversion of the value fails, an error message associated with the component is generated and queued on the `FacesContext`. This message will be displayed during the Render Response phase, along with any validation errors resulting from the Process Validations phase.

In the case of the `userNumber` component on the `greeting.jsp` page, the value is whatever the user entered in the field. Since the object property bound to the component has an `Integer` type, the JavaServer Faces implementation converts the value from a `String` to an `Integer`.

If any `decode` methods or event listeners called `renderResponse` on the current `FacesContext`, the JavaServer Faces implementation skips to the Render Response phase.

If events have been queued during this phase, the JavaServer Faces implementation broadcasts the events to interested listeners. See Implementing an Event Listener (page 888) for more information on how to specify which lifecycle processing phase the listener will process events.

At this point, if the application needs to redirect to a different Web application resource or generate a response that does not contain any JavaServer Faces components, it can call `FacesContext.responseComplete`.

At this point, the components are set to their new values, and messages and events have been queued.

Process Validations

During this phase, the JavaServer Faces implementation processes all validations registered on the components in the tree. It examines the component attributes that specify the rules for the validation and compares these rules to the local value stored for the component.

If the local value is invalid, the JavaServer Faces implementation adds an error message to the `FacesContext` and the lifecycle advances directly to the Render

Response phase so that the page is rendered again with the error messages displayed. If there were conversion errors from Apply Request Values, the messages for these errors are also displayed.

If any validate methods or event listeners called `renderResponse` on the current `FacesContext`, the JavaServer Faces implementation skips to the Render Response phase.

At this point, if the application needs to redirect to a different Web application resource or generate a response that does not contain any JavaServer Faces components, it can call `FacesContext.responseComplete`.

If events have been queued during this phase, the JavaServer Faces implementation broadcasts them to interested listeners. See Implementing an Event Listener (page 888) for more information on how to specify in which lifecycle processing phase a listener will process events.

In the `greeting.jsp` page, the JavaServer Faces implementation processes the validator on the `userNumber input_text` tag. It verifies that the data the user entered in the text field is an integer from the range 0 to 10. If the data is invalid, or conversion errors occurred during the Apply Request Values phase, processing jumps to the Render Response phase, during which the `greeting.jsp` page is rendered again with the validation and conversion error messages displayed in the component associated with the `messages` tag.

Update Model Values

Once the JavaServer Faces implementation determines that the data is valid, it can walk the component tree and set the corresponding server side object properties to the components' local values. The JavaServer Faces implementation will only update the bean properties pointed at by an input component's value attribute. If the local data cannot be converted to the types specified by the bean properties, the lifecycle advances directly to Render Response so that the page is re-rendered with errors displayed, similar to what happens with validation errors.

If any `updateModels` methods or any listeners called `renderResponse` on the current `FacesContext`, the JavaServer Faces implementation skips to the Render Response phase.

At this point, if the application needs to redirect to a different Web application resource or generate a response that does not contain any JavaServer Faces components, it can call `FacesContext.responseComplete`.

If events have been queued during this phase, the JavaServer Faces implementation broadcasts them to interested listeners. See Implementing an Event Listener (page 888) for more information on how to specify in which lifecycle processing phase a listener will process events.

At this stage, the `userNumber` property of the `UserNumberBean` is set to the local value of the `userNumber` component.

Invoke Application

During this phase, the JavaServer Faces implementation handles any application-level events, such as submitting a form or linking to another page.

At this point, if the application needs to redirect to a different Web application resource or generate a response that does not contain any JavaServer Faces components, it can call `FacesContext.responseComplete`.

If the view being processed was reconstructed from state information from a previous request and a component has fired an event, these events are broadcast to interested listeners.

The `greeting.jsp` page from the `guessNumber` example has one application-level event associated with the `UICommand` component. When processing this event, a default `ActionListener` implementation retrieves the outcome, "success", from the component's `action` attribute. The listener passes the outcome to the default `NavigationHandler`. The `NavigationHandler` matches the outcome to the proper navigation rule defined in the application's application configuration file to determine what page needs to be displayed next. See Navigating Between Pages (page 894) for more information on managing page navigation. The JavaServer Faces implementation then sets the response view to that of the new page. Finally, the JavaServer Faces implementation transfers control to the Render Response phase.

Render Response

During the Render Response phase, the JavaServer Faces implementation invokes the components' encoding functionality and renders the components from the component tree saved in the `FacesContext`.

If errors were encountered during the Apply Request Values phase, Process Validations phase, or Update Model Values phase, the original page is rendered during this phase. If the pages contain `messages` tags, any queued error messages are displayed on the page.

New components can be added to the view if the application includes custom renderers, which define how to render a component. After the content of the view is rendered, the state of the response is saved so that subsequent requests can access it and it is available to the Restore View phase. The Restore View phase accesses the tree during a subsequent request.

User Interface Component Model

JavaServer Faces UI components are configurable, reusable elements that compose the user interfaces of JavaServer Faces applications. A component can be simple, like a button, or compound, like a table, which can be composed of multiple components.

JavaServer Faces technology provides a rich, flexible component architecture that includes:

- A set of `UIComponent` classes for specifying the state and behavior of UI components
- A rendering model that defines how to render the components in different ways.
- An event and listener model that defines how to handle component events
- A conversion model that defines how to plug in data converters onto a component
- A validation model that defines how to register validators onto a component

This section briefly describes each of these pieces of the component architecture.

The User-Interface Component Classes

JavaServer Faces technology provides a set of UI component classes, which specify all of the UI component functionality, such as holding component state, maintaining a reference to objects, and driving event-handling and rendering for a set of standard components.

These classes are completely extensible, allowing component writers to create their own custom components. See Creating Custom UI Components (page 905) for an example of a custom image map component.

All JavaServer Faces UI component classes extend from `UIComponentBase`, which defines the default state and behavior of a `UIComponent`. The set of UI component classes included in this release of JavaServer Faces technology are:

- `UIColumn`: Represents a single column of data in a `UIData` component
- `UICommand`: Represents a control that fires actions when activated.
- `UIData`: Represents a data binding to a collection of data represented by a `DataModel` instance.
- `UIForm`: Encapsulates a group of controls that submit data to the application. This component is analogous to the form tag in HTML.
- `UIGraphic`: Displays an image.
- `UIMessage`: Displays a localized message
- `UIMessages`: Displays a set of localized messages
- `UIInput`: Takes data input from a user. This class is a subclass of `UIOutput`.
- `UIOutput`: Displays data output on a page.
- `UIPanel`: Manages the layout of its child components.
- `UIParameter`: Represents substitution parameters.
- `UISelectItem`: Represents a single item in a set of items.
- `UISelectItems`: Represents an entire set of items.
- `UISelectBoolean`: Allows a user to set a `boolean` value on a control by selecting or de-selecting it. This class is a subclass of `UIInput`.
- `UISelectMany`: Allows a user to select multiple items from a group of items. This class is a subclass of `UIInput`.
- `UISelectOne`: Allows a user to select one item out of a group of items.This class is a subclass of `UIInput`.
- `UIViewRoot`: Represents the root of the component tree

Most page authors and application developers will not have to use these classes directly. They will instead include the components on a page by using the component's corresponding tag. Most of these component tags can be rendered in different ways. For example, a `UICommand` can be rendered as a button or a hyperlink.

The next section explains how the rendering model works and how page authors choose how to render the components by selecting the appropriate tag.

The Component Rendering Model

The JavaServer Faces component architecture is designed such that the functionality of the components is defined by the component classes, whereas the component rendering can be defined by a separate renderer. This design has several benefits including:

- Component writers can define the behavior of a component once, but create multiple renderers, each of which defines a different way to render the component to the same client or to different clients.
- Page authors and application developers can change the appearance of a component on the page by selecting the tag that represents the appropriate component/renderer combination.

A *render kit* defines how component classes map to component tags appropriate for a particular client. The JavaServer Faces implementation includes a standard `RenderKit` class for rendering to an HTML client.

For every UI component that a render kit supports, the render kit defines a set of `Renderer` objects. Each `Renderer` defines a different way to render the particular component to the output defined by the render kit. For example, a `UISelectOne` component has three different renderers. One of them renders the component as a set of radio buttons. Another renders the component as a combo box. The third one renders the component as a list box.

Each JSP custom tag defined in the standard HTML `RenderKit` class is composed of the component functionality, defined in the `UIComponent` class, and the rendering attributes, defined by the `Renderer`. For example, the two tags in Table 20–1 both represent a `UICommand` component, rendered in two different ways:

Table 20–1 `UICommand` Tags

Tag	Rendered as
`command_button`	 **Figure 20–4** Login Button

Table 20–1 `UICommand` Tags (Continued)

Tag	Rendered as
`command_link`	hyperlink **Figure 20–5** A Hyperlink

The command part of the tags corresponds to the `UICommand` class, specifying the functionality, which is to fire an action. The button and hyperlink parts of the tags each correspond to a separate `Renderer`, which defines how the component appears on the page.

The JavaServer Faces reference implementation provides a custom tag library for rendering components in HTML. It supports all of the component tags listed in Table 20–2. To learn how to use the tags in an example, see Using the JavaServer Faces Tag Libraries (page 813).

Table 20–2 The Component Tags

Tag	Functions	Rendered as	Appearance
`column`	Represents a column of data in a `UIData` component	A column of data in an HTML table	A column in a table
`command_button`	Submits a form to the application.	An HTML `<input type=`*type*`>` element, where the *type* value can be `submit`, `reset`, or `image`	A button
`command_link`	Links to another page or location on a page.	An HTML `<a href>` element	A Hyperlink
`data_table`	Represents a data wrapper	An HTML `<table>` element	A table that can be updated dynamically

Table 20–2 The Component Tags (Continued)

Tag	Functions	Rendered as	Appearance
`form`	Represents an input form. The inner tags of the form receive the data that will be submitted with the form.	An HTML `<form>` element	No appearance
`graphic_image`	Displays an image.	An HTML `<img>` element	An image
`input_hidden`	Allows a page author to include a hidden variable in a page.	An HTML `<input type=hidden>` element	No appearance
`input_secret`	Allows a user to input a string without the actual string appearing in the field.	An HTML `<input type=password>` element	A text field, which displays a row of characters instead of the actual string entered
`input_text`	Allows a user to input a string.	An HTML `<input type=text>` element	A text field
`input_textarea`	Allows a user to enter a multi-line string.	An HTML `<textarea>` element	A multi-row text field
`message`	Displays a localized message	An HTML `<span>` tag if styles are used	A text string
`messages`	Displays localized messages	A set of HTML `<span>` tags if styles are used	A text string
`output_label`	Displays a nested component as a label for a specified input field.	An HTML `<label>` element	plain text
`output_link`	Links to another page or location on a page without generating an `ActionEvent`	An HTML `<a>` element	A hyperlink

Table 20–2 The Component Tags (Continued)

Tag	Functions	Rendered as	Appearance
`output_message`	Displays a localized message.	plain text	plain text
`output_text`	Displays a line of text.	plain text	plain text
`panel_grid`	Displays a table.	An HTML `<table>` element with `<tr>` and `<td>` elements	A table
`panel_group`	Groups a set of components under one parent.		A row in a table
`selectboolean_checkbox`	Allows a user to change the value of a boolean choice.	An HTML `<input type=checkbox>` element.	A checkbox
`selectitem`	Represents one item in a list of items in a `UISelectOne` component.	An HTML `<option>` element	No appearance
`selectitems`	Represents a list of items in a `UISelectOne` component.	A list of HTML `<option>` elements	No appearance
`selectmany_checkboxlist`	Displays a set of checkboxes, from which the user can select multiple values.	A set of HTML `<input>` elements of type checkbox	A set of checkboxes
`selectmany_listbox`	Allows a user to select multiple items from a set of items, all displayed at once.	A set of HTML `<select>` elements	A list box
`selectmany_menu`	Allows a user to select multiple items from a set of items.	A set of HTML `<select>` elements	A scrollable combo box

Table 20–2 The Component Tags (Continued)

Tag	Functions	Rendered as	Appearance
`selectone_listbox`	Allows a user to select one item from a set of items, all displayed at once.	A set of HTML `<select>` elements	A list box
`selectone_menu`	Allows a user to select one item from a set of items.	An HTML `<select>` element	A scrollable combo box
`selectone_radio`	Allows a user to select one item from a set of items.	An HTML `<input type=radio>` element	A set of radio buttons

Conversion Model

A JavaServer Faces application can optionally associate a component with server-side object data. This object is a JavaBeans component, such as a backing bean. An application gets and sets the object data for a component by calling the appropriate object properties for that component.

When a component is bound to an object, the application has two views of the component's data: the model view and the presentation view, which represents the data in a manner that can be viewed and modified by the user.

A JavaServer Faces application must ensure that the component's data can be converted between the model view and the presentation view. This conversion is usually performed automatically.

In some situations, you might want to specify a particular converter for a component's data. To facilitate this, JavaServer Faces technology includes a set of standard `Converter` implementations and also allows you to create your own custom `Converter` implementations. If you register the `Converter` implementation on a component, the `Converter` implementation converts the component's data between the two views. See Performing Data Conversions (page 878) for more details on the converter model, how to use the standard converters, and how to create and use your own custom converter.

Event and Listener Model

One goal of the JavaServer Faces specification is to leverage existing models and paradigms so that developers can quickly become familiar with using JavaServer Faces in their web applications. In this spirit, the JavaServer Faces event and listener model leverages the JavaBeans architecture event model design, which is familiar to GUI developers and Web Application Developers.

Like the JavaBeans component architecture, JavaServer Faces technology defines `Listener` and `Event` classes that an application can use to handle events generated by UI components. An `Event` object identifies the component that generated the event and stores information about the event. To be notified of an event, an application must provide an implementation of the `Listener` class and register it on the component that generates the event. When the user activates a component, such as by clicking a button, an event is fired. This causes the JavaServer Faces implementation to invoke the listener method that processes the event.

JavaServer Faces supports three kinds of events: value-change events, action events, and data-model events.

A *value-change* event occurs when the user changes a component value. An example is selecting a checkbox, which results in the component's value changing to true. The component types that generate these types of events are the `UIInput`, `UISelectOne`, `UISelectMany`, and `UISelectBoolean` components. Value-change events are only fired if no validation errors were detected.

An *action event* occurs when the user clicks a button or a hyperlink. The `UICommand` component generates this event.

A *data model event* occurs when a new row of a `UIData` component is selected.

For more information on handling these different kinds of events, see Handling Events (page 887).

Validation Model

JavaServer Faces technology supports a mechanism for validating a component's local data during the Process Validations (page 787) phase, before object data is updated.

Like the conversion model, the validation model defines a set of standard classes for performing common data validation checks. The `jsf_core` tag library also defines a set of tags that correspond to the standard `Validator` implementations.

Most of the tags have a set of attributes for configuring the validator's properties, such as the minimum and maximum allowable values for the component's data. The page author registers the validator on a component by nesting the validator's tag within the component's tag.

Also like the conversion model, the validation model allows you to create your own `Validator` implementation and corresponding tag to perform custom validation. You can also provide a method in a backing-file bean that performs validation for a component. The component refers to this method through its `validator` attribute. See Performing Validation (page 864) for more information on the standard `Validator` implementations, how to create custom `Validator` implementation and validator tags, and how to reference validation code from the component tag's `validator` attribute.

Navigation Model

Virtually all web applications are made up of a set of pages. One of the primary concerns of a web application developer is managing the navigation between these pages.

The JavaServer Faces navigation model makes it easy to define page navigation and to handle any additional processing needed to choose the sequence in which pages are loaded. In many cases, no code is required to define navigation. Instead, navigation can be defined completely in the application configuration file (see section Application Configuration, page 801) using a small set of XML elements. The only situation in which you need to provide some code is if additional processing is required to determine which page to access next.

To load the next page in a web application, the user usually clicks a button. As explained in Define Page Navigation (page 778), a button click generates an action event. The JavaServer Faces implementation provides a default action event listener to handle this event. This listener determines the outcome of the action, such as success or failure. This outcome can be defined as a string property of the component that generated the event or as the result of extra processing performed in a bean property associated with the component. After the outcome is determined, the listener passes it to the `NavigationHandler` instance associated with the application. Based on which outcome is returned, the `Navigation-`

Handler selects the appropriate page by consulting the application configuration file.

For more information on how to perform page navigation, see section Navigating Between Pages (page 894).

Backing Bean Management

Another critical function of web applications is proper management of resources. This includes separating the definition of UI component objects from objects that perform application-specific processing and hold data. It also includes storing and managing these object instances in the proper scope.

A typical JavaServer Faces application includes one or more backing beans, which are server-side objects associated with UI components used in the page. A backing bean defines UI component properties, each of which is bound to either a component's value or a component instance. A backing bean can also define methods that perform functions associated with a component, which include validation, event handling, and navigation processing.

A page author uses the *JavaServer Faces expression language* (JSF EL) to bind a component's value or its instance to a backing bean property. The expression must be enclosed in the curly brackets of "#{}". A page author also uses the JSF EL to refer to the backing-bean methods that perform processing for the component.

For example, consider the input_text tag from the greeting.jsp page of the guessNumber application:

```
<h:input_text id="userNo"
  value="#{UserNumberBean.userNumber}"
  validator="#{UserNumberBean.validate}" />
```

This tag binds the userNo component's value to the UserNumberBean.userNumber backing-bean property. It also refers to the UserNumberBean.validate method, which performs validation of the component's local value, which is whatever the user enters into the field corresponding to this tag.

The property bound to the component's value must be of a type supported by the component. For example, the userNumber property returns an Integer, which is one of the types that a UIInput component supports, as shown in Develop the Beans (page 780).

In addition to the `validator` attribute, tags representing a `UIInput` can also use a `valueChangeListener` attribute to refer to a method that responds to `ValueChangeEvents`, which a `UIInput` component can fire.

A tag representing a component that implements `ActionSource` can refer to backing-bean methods using `actionListener` and `action` attributes. The `actionListener` attribute refers to a method that handles an `ActionEvent`. The `action` attribute refers to a method that performs some processing associated with navigation and returns a logical outcome, which the navigation system uses to determine which page to display next.

A tag can also bind a component instance to a backing-bean property. It does this by referencing the property from the `binding` attribute:

```
<input_text binding="#{UserNumberBean.userNoComponent}" />
```

The property referenced from the `binding` attribute must accept and return the same component type as the component instance to which it's bound. Here is an example property that can be bound to the component represented by the preceding example `input-text` tag:

```
UIInput userNoComponent = null;
...
public void setUserNoComponent(UIInput userNoComponent) {
  this.userNoComponent = userNoComponent;
}
public UIInput getUserNoComponent() {
  return userNoComponent;
}
```

When a component instance is bound to a backing-bean property, the property holds the component's local value. Conversely, when a component's value is bound to a backing-bean property, the property holds its model value, which is updated with the local value during the Update Model Values phase of the lifecycle.

Binding a component instance to a bean property has these advantages:

- The backing bean can programmatically modify component attributes.
- The backing bean can instantiate components rather than letting the page author do so.

Binding a component's value to a bean property has these advantages:

- The page author has more control over the component attributes.
- The backing-bean has no dependencies on the JavaServer Faces API, allowing for greater separation of the presentation layer from the model layer.
- The JavaServer Faces implementation can perform conversions on the data based on the type of the bean property without the developer needing to apply a converter.

Backing beans are created and stored with the application using the *Managed Bean Creation Facility*, which is configured in the application configuration file, as shown in Adding Managed Bean Declarations (page 781). When the application starts up, it processes this file, making the beans available to the application and instantiating them when the component tags reference them.

For more information on configuring beans using the Managed Bean Creation Facility, see Configuring Beans (page 849). For more information on writing the beans and their properties, see Writing Component Properties (page 838). For more information on binding component instances or data to properties, see Binding Component Values and Instances to External Data Sources (page 858). For information on referencing backing-bean methods from component tags, see Referencing a Backing Bean Method (page 863).

Application Configuration

Previous sections of this chapter have discussed the various resources available to a JavaServer Faces application. These include: converters, validators, components, beans, actions, navigation handlers, and others.

JavaServer Faces technology provides a portable configuration resource format (an XML document) for configuring resources required at application startup time. This tutorial explains in separate sections how to configure resources in the XML document. See Setting Up The Application Configuration File (page 811) for information on requirements for setting up the application configuration file. See Configuring Beans (page 849) for an explanation of how to use the XML file to instantiate beans. See Navigating Between Pages (page 894) for information on how to define page navigation in the XML file. See Performing Validation (page 864) and Performing Data Conversions (page 878) for how to register custom validators and converters. See Register the

Component (page 934) and Register the Renderer with a Render Kit (page 933) for information on how to register components and renderers to an application.

To access resources registered with the application, you use the `Application` class, which is automatically created for each application. The `Application` class acts as a centralized factory for resources that are defined in the XML file.

When an application starts up, the JavaServer Faces implementation creates a single instance of the `Application` class and configures it with the information you configure in the application configuration file.

When you need to access the `Application` instance, the easiest way to retrieve it is to call the `getApplication` method of the `FacesContext` instance.

About the Examples

The JavaServer Faces 1.0 Beta release includes four complete, working examples, which are located in the *<JSF_HOME>*`/samples` directory of your JavaServer Faces technology download. Table 20–3 lists the examples.

This tutorial uses these examples to explain JavaServer Faces technology. It also uses the Coffee Break (See Chapter 25) application to document some features.

Table 20–3 Examples

Example	Location	Function
cardemo	*<JSF_HOME>*`/samples/cardemo`	A car store application
guessNumber	*<JSF_HOME>*`/samples/guessNumber`	Duke asks you to guess a number
non-jsp	*<JSF_HOME>*`/samples/non-jsp`	Refers to a backing-file bean method that performs navigation processing for the component and returns a logical outcome
components	*<JSF_HOME>*`/samples/components`	Showcases tabbed-panes, tree-control, and repeater custom components

Running the Examples on the Java WSDP Container

The Java WSDP includes an older version of JavaServer Faces technology. In order to run JavaServer Faces applications based on the latest version of Jav-Server Faces technology with the Java WSDP, you must remove the older version from the Java WSDP installation and replace it with the latest version. You do this by following the steps detailed in the section entitled Installing JavaServer Faces Technology in Java WSDP in the Installation Instructions, which are located in the `<JSF_HOME>/docs` directory of the JavaServer Faces download.

The JavaServer Faces technology download includes pre-built WAR files of each of the examples. These WAR files are located in the respective example directories, all under the `<JSF_HOME>/samples` directory.

Once you have the latest JavaServer Faces technology in the Java WSDP, follow the directions in the section entitled, Running the Examples in Java WSDP in the same Installation Instructions, which details how to run the example using the pre-built WAR file.

Building and Running the Sample Applications Manually

It is also possible to build each of the sample applications yourself. Before doing so, you need to set the environment variables, as described in About the Examples (page xxv). After setting environment variables, follow the instructions in the section, Building and Running the Examples in the Installation Instructions, located in `<JSF_HOME>/docs`.

The cardemo Example

This chapter primarily uses the `cardemo` example to illustrate the basic concepts of JavaServer Faces technology. This example emulates on online car dealership, with features such as price updating, car option packaging, a custom converter, a custom validator, and an image map custom component. Table 20–4 lists all of the files used in this example, except for the image and properties files.

Table 20–4 Example Files

File	Function
`chooseLocale.jsp`	The first page that allows you to select a locale
`storeFront.jsp`	Shows the cars available
`carDetail.jsp`	Allows you to choose the options for a particular car
`confirmChoices.jsp`	Shows the options currently chosen for a particular car
`customerInfo.jsp`	Allows you to enter your personal information so that you can order the car
`finish.jsp`	The final page that thanks you for ordering the car
`CarBean.java`	Encapsulates a car model, including pricing and package choices
`CarCustomizer.java`	Customizes a CarBean for a set of options in a package
`CarStore.java`	The main bean for the application. Maintains a map of CarBean nstances, keyed by model name, and a Map CarCustomizer instances, keyed by package name.
`CustomerBean.java`	Represents the model for the customer information
`FirstNam-eChanged.java`	Handles the event of entering a value in the First Name field on the customerInfo page
`FormatValida-tor.java`	Defines a custom `Validator`
`MessageFactory.java`	Retrieves localized messages from the application

The `cardemo` also uses a set of beans, custom components, renderers, and tags, as shown in Table 20–5. These files are located in the `samples/components` directory of your JavaServer Faces technology download.

Table 20–5 Beans, Custom Components, Renderers, and Tags Used by `cardemo`

File	Function
`AreaComponent`	The class that defines the component corresponding to the `area` custom tag
`AreaRenderer`	This `Renderer` performs the delegated rendering for the `UIArea` component
`AreaTag`	The tag handler that implements the `area` custom tag
`ImageArea`	The bean that stores the shape and coordinates of the hot spots
`MapComponent`	The class that defines the component orresponding to the `map` custom tag
`MapTag`	The tag handler that implements the `map` custom tag

Basic Requirements of a JavaServer Faces Application

In addition to developing your application, as described in Steps in the Development Process (page 775), you need to satisfy some other requirements of JavaServer Faces applications, including configuring your application, packaging all of the necessary files, packaging the files properly, and configuring resources. This section describes how to perform all of these administrative tasks.

JavaServer Faces applications must be compliant with the Java Servlet specification, version 2.3 (or later) and the JavaServer Pages specification, version 1.2 (or later). All JavaServer applications are packaged in a WAR file, which must conform to specific requirements in order to execute across different JavaServer

Faces implementations. At a minimum, a WAR file for a JavaServer Faces application must contain:

- A Web application deployment descriptor, called `web.xml`, to configure resources required by a Web application.
- A specific set of JAR files containing essential classes.
- A set of application classes, JavaServer Faces pages, and other required resources, such as image files.
- An application configuration file, which defines application resources

The WAR file typically has this directory structure:

```
index.html
JSP pages
WEB-INF/
   web.xml
   faces-config.xml
   tag library descriptors (optional)
   classes/
      class files
   lib/
      JAR files
```

The `web.xml`, the set of JAR files, and the set of application files must be contained in the `WEB-INF` directory of the WAR file. Usually, you will want to use the `Ant` build tool to compile the classes, build the necessary files into the WAR, and deploy the WAR file. The `Ant` tool is included in the Java WSDP. You configure how the `Ant` build tool builds your WAR file with a `build.xml` file. Each example in the download has its own build file, to which you can refer when creating your own build file.

Another requirement is that all requests to a JavaServer Faces application that reference previously saved JavaServer Faces components must go through the `FacesServlet`. The `FacesServlet` manages the request processing lifecycle for Web applications and initializes the resources required by the JavaServer Faces implementation. To make sure your JavaServer Faces application complies with this requirement, see the section, Identifying the Servlet for Lifecycle Processing (page 808).

Writing the web.xml File

The `web.xml` file is located at the top level of the `WEB-INF` directory. See Configuring Web Applications (page 85) to find out what a standard `web.xml` file should contain.

The `web.xml` file for a JavaServer Faces application must specify certain configurations, which include:

- The servlet used to process JavaServer Faces requests
- The servlet mapping for the processing servlet

The following XML markup defines the required configurations specific to JavaServer Faces technology for the `cardemo` application:

```
<web-app>
  ...
  <context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>client</param-value>
  </context-param>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>
      javax.faces.webapp.FacesServlet
    </servlet-class>
    <load-on-startup> 1 </load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>
</web-app>
```

Specifying Where State is Saved

When implementing the state-holder methods in a component class (described in Saving and Restoring State, page 929), specify in your web.xml file where you want the state to be saved, either client or server. You do this with a `context-param` element that has a nested `param-name` element that specifies `javax.faces.STATE_SAVING_METHOD`.

The `context-param` element from the example `web.xml` file above specifies that state must be saved in the client. If state is saved on the client, the state of the entire view is rendered to a hidden field on the page. If you want to save the state on the server, you need to specify "server" in the param-value element nested in the context-param element. The JavaServer Faces implementation saves the state on the server by default.

Identifying the Servlet for Lifecycle Processing

The servlet element identifies the FacesServlet, which processes the lifecycle of the application. The load-on-startup element has a value of true, which indicates that the FacesServlet should be loaded when the application starts up.

The servlet-mapping element lists each servlet name defined in the servlet element and gives the URL path to the servlet. Tomcat will map the path to the servlet when a request for the servlet is received.

Before a JavaServer Faces application can launch the first JSP page, the Web container must invoke the FacesServlet in order for the application lifecycle process to start. The application lifecycle is described in the section, The Lifecycle of a JavaServer Faces Page (page 783).

To make sure that the FacesServlet is invoked, you need to include the path to the FacesServlet in the URL to the first JSP page. You define the path in the url-pattern element nested inside the servlet-mapping element of the web.xml file.

The mapping in the example web.xml file above uses extension mapping to identify a JSP page as having JavaServer Faces content. If a request comes to the server for a JSP page with a .jsf extension, the container will send the request to the FacesServlet, which will expect a corresponding JSP page of the same name to exist containing the content. For example, if the request URL is http://localhost/cardemo/storeFront.jsf, the FacesServlet will map to the storeFront.jsp page.

You can also specify a prefix mapping instead of an extension mapping. For example, you can replace the *.jsf in the web.xml file above with /faces/. If you do this, however, the user of your application must enter the /faces/prefix in the URL by doing one of two things:

- Include an HTML page in your application that has the URL to the first JSP page, and include the path to the FacesServlet:

  ```
  <a href="faces/First.jsp">
  ```

- Include the path to the FacesServlet in the URL to the first page when you enter it in your browser:

  ```
  http://localhost:8080/myApp/faces/First.jsp
  ```

The second method allows you to start your application from the first JSP page, rather than starting it from an HTML page. However, the second method requires your user to identify the first JSP page. When you use the first method, the user only has to enter:

```
http://localhost:8080/myApp
```

Including the Required JAR Files

JavaServer Faces applications require several JAR files to run properly. If you are not running the application on the Java WSDP, which already has these JAR files, the WAR file for your JavaServer Faces application must include the following set of JAR files in the `WEB-INF/lib` directory:

- `jsf-api.jar` (contains the `javax.faces.*` API classes)
- `jsf-impl.jar` (contains the implementation classes of the JavaServer Faces RI)
- `jstl.jar` (required to use JSTL tags and referenced by JavaServer Faces reference implementation classes)
- `standard.jar` (required to use JSTL tags and referenced by JavaServer Faces reference implementation classes)
- `commons-beanutils.jar` (utilities for defining and accessing JavaBeans component properties)
- `commons-digester.jar` (for processing XML documents)
- `commons-collections.jar` (extensions of the Java 2 SDK Collections Framework)
- `commons-logging.jar` (a general purpose, flexible logging facility to allow developers to instrument their code with logging statements)

Including the Classes, Pages, and Other Resources

All application classes and properties files should be copied into the `WEB-INF/classes` directory of the WAR file during the build process. JavaServer Faces pages should be at the top level of the WAR file. The `web.xml`, `faces-config.xml`, and extra TLD files should be in the `WEB-INF` directory. Other resources, such as images can be at the top level or in a separate directory of the WAR file.

The `build` target of the example build file copies all of these files to a temporary `build` directory. This directory contains an exact image of the binary distribution for your JavaServer Faces application:

```
<target name="build" depends="prepare"
  description="Compile Java files and copy static files." >
  <javac srcdir="src"
       destdir="${build}/${example}/WEB-INF/classes">
    <include name="**/*.java" />
    <classpath refid="classpath"/>
  </javac>
  <copy todir="${build}/${example}/WEB-INF">
    <fileset dir="web/WEB-INF"    >
      <include name="web.xml" />
      <include name="*.tld" />
      <include name="*.xml" />
    </fileset>
  </copy>
  <copy todir="${build}/${example}/">
    <fileset dir="web">
      <include name="*.html" />
      <include name="*.gif" />
      <include name="*.jpg" />
      <include name="*.jsp" />
      <include name="*.xml" />
      <include name="*.css" />
    </fileset>
  <copy todir=
    "${build}/${example}/WEB-INF/classes/${example}">
    <fileset dir="web" >
      <include name="*properties"/>
    </fileset>
  </copy>
  <jar jarfile=
    "${example}.war" basedir="${build}/${example}"/>
  <copy todir="${build}" file="${example}.war" />
  <delete file="${example}.war" />
</target>
```

When writing a build file for your Web application, you can follow the build files included with each example.

Setting Up The Application Configuration File

The application configuration file is an XML file, usually named `faces-config.xml`, whose purpose is to configure resources for an application. These resources include: navigation rules, converters, validators, render kits, and others. For a complete description of the application configuration file, see Application Configuration (page 801). This section explains the basic requirements of the application configuration file.

The application configuration file must be valid against the DTD located at `http://java.sun.com/dtd/web-facesconfig_1_0.dtd`. In addition, each file must include in this order:

- The XML version number:
  ```
  <?xml version="1.0"?>
  ```
- This DOCTYPE declaration at the top of the file:
  ```
  <!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0/
  /EN"
  "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
  ```
- A `faces-config` tag enclosing all of the other declarations:
  ```
  <faces-config>
  ...
  </faces-config>
  ```

You can have more than one application configuration file, and there are three ways that you can make these files available to the application. The JavaServer Faces implementation finds the file or files by looking for:

- A resource named `/META-INF/faces-config.xml` in any of the JAR files in the Web application's `/WEB-INF/lib` directory. If a resource with this name exists, it is loaded as a configuration resource. This method is practical for a packaged library containing some components and renderers. The `demo-components.jar`, located in `<`*`JWSDP_HOME`*`>/jsf/samples` uses this method.
- A context init parameter, `javax.faces.application.CONFIG_FILES` that specifies one or more (comma-delimited) paths to multiple configuration files for your Web application. This method will most likely be used for enterprise-scale applications that delegate the responsibility for maintaining the file for each portion of a big application to separate groups.

- A resource named `faces-config.xml` in the `/WEB-INF/` directory of your application if you don't specify a context init parameter. This is the way most simple applications will make their configuration files available.

21

Using JavaServer Faces Technology

THIS section shows you how to get started using JavaServer Faces technology in a Web application by demonstrating simple JavaServer Faces features using working examples.

Using the JavaServer Faces Tag Libraries

JavaServer Faces technology provides two tag libraries: `html_basic` and `jsf_core`. The `html_basic` tag library defines tags for representing common HTML user interface components. The tags in `jsf_core` are independent of any rendering technology and can therefore be used with any render kit. Using these tag libraries is similar to using any other custom tag library. This section assumes that you are familiar with the basics of Custom Tags in JSP Pages (page 705).

Requirements For Using the Tags on The Page

To use any of the JavaServer Faces tags, you need to include these `taglib` directives at the top of each page containing the tags defined by these tag libraries:

```
<%@ taglib uri="http://java.sun.com/jsf/html/" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core/" prefix="f" %>
```

The `uri` attribute value uniquely identifies the tag library. The `prefix` attribute value is used to distinguish tags belonging to the tag library. For example, the `form` tag must be referenced in the page with the `h` prefix, like this:

```
<h:form ...>
```

Make sure that you also package the `html_basic` and `jsf_core` TLDs with your application. See Basic Requirements of a JavaServer Faces Application (page 805) for details.

A JavaServer Faces page is represented by a tree of components. At the root of the tree is the `UIViewRoot` component. The `view` tag represents this component on the page. As such, all component tags on the page must be enclosed in the `view` tag, which is defined in the `jsf_core` library:

```
<f:view>
  ... other faces tags, possibly mixed with other content ...
</f:view>
```

You can enclose other content within the `view` tag, including HTML and other JSP tags, but all JavaServer Faces tags must be enclosed within the `view` tag.

The `view` tag has an optional `locale` attribute. If this attribute is present, its value overrides the `Locale` stored in the `UIViewRoot`. This value is specified as a `String` and must be of the form:

```
:language:[{-,_}:country:[{-,_}:variant]
```

The :language:, :country: and :variant: parts of the expression are as specified in java.util.Locale.

If you want to include a JavaServer Faces page within another JavaServer Faces page or another JSP page, you must enclose the entire nested page in an `f:sub-`

`view` tag. You can add the `subview` tag on the parent page and nest a `jsp:include` inside it to include the page, like this:

```
<f:subview>
  jsp:include page="theNestedPage.jsp"
<f:subview>
```

You can also include `subview` inside the nested page, but it must enclose all of the JavaServer Faces components on the nested page. The `cardemo` application has a page called `optionsPanel`, which is nested inside `carDetail`.

Using the Core Tags

The tags defined by the `jsf_core` TLD are used to perform core actions that are independent of a particular render kit. The `jsf_core` tags are listed in Table 21–1.

Table 21–1 The `jsf-core` Tags

	Tags	**Functions**
Event Handling Tags	`action_listener`	Registers an action listener on a parent component
	`valuechange_listener`	Registers a value-change listener on a parent component
Attribute Configuration Tag	`attribute`	Adds configurable attributes to a parent components
Data Conversion Tags	`converter`	Registers an arbitrary converter on the parent component
	`convert_datetime`	Registers a DateTime converter instance on the parent component
	`convert_number`	Registers a Number converter instance on the parent component
Facet Tag	`facet`	Signifies a nested component that has a special relationship to its enclosing tag

Table 21–1 The `jsf-core` Tags (Continued)

	Tags	Functions
Parameter Substitution Tag	`parameter`	Substitutes parameters into a `MessageFormat` instance and to add query string name/value pairs to a URL.
Tags for Representing Items in a List	`selectitem`	Represents one item in a list of items in a `UISelectOne` or `UISelectMany` component
	`selectitems`	Represents a set of items in a `UISelectOne` or `UISelectMany` component
Container Tag	`subview`	Contains all JavaServer Faces tags in a page that is included in another JavaServer Faces page.
Validator Tags	`validate_doublerange`	Registers a `DoubleRangeValidator` on a component
	`validate_length`	Registers a `LengthValidator` on a component
	`validate_longrange`	Registers a `LongRangeValidator` on a component
	`validator`	Registers a custom `Validator` on a component
Output Tag	`verbatim`	Generates a `UIOutput` component that gets its content from the body of this tag.
Container For Form Tags	`view`	Encloses all JavaServer Faces tags on the page.

These tags are used in conjunction with component tags and are therefore explained in other sections of this tutorial. Table 21–2 lists which sections explain how to use which `jsf-core` tags.

Table 21–2 Where the `jsf-core` Tags are Explained

Tags	**Where Explained**
Event-Handling Tags	Handling Events (page 889)
Data Conversion Tags	Performing Data Conversions (page 880)
`facet`	The UIPanel Component (page 829)
`parameter`	Using the output_message Tag (page 828)
`selectitem` and `selectitems`	The UISelectMany Component (page 833) and The UISelectOne Component (page 835)
`subview`	Requirements For Using the Tags on The Page (page 814)
`verbatim`	Using the output_link Tag (page 828)
`view`	Requirements For Using the Tags on The Page (page 814)
Validator Tags	Performing Validation (page 866)

Using the HTML Tags

The tags defined by `html_basic` represent HTML form controls and other basic HTML elements. These controls display data or accept data from the user. This data is collected as part of a form and is submitted to the server, usually when the user clicks a button. This section explains how to use each of the component tags shown in Table 20–2, and is organized according to the `UIComponent` classes from which the tags are derived.

This section does not explain every tag attribute, only the most commonly-used ones. Please refer to `html_basic.tld` file in the *<JSF_HOME>*`/lib` directory of your download for a complete list of tags and their attributes.

In general, most of the component tags have these attributes in common:

- `id`: uniquely identifies the component
- `value`: identifies an external data source mapped to the component's value
- `binding`: identifies a bean property mapped to the component instance

The `id` attribute is not required for a component tag except in these situations:

- Another component or a server-side class must refer to the component
- The component tag is impacted by a JSTL conditional or iterator tag (for more information, see Flow Control Tags (page 686)).

If you don't include an `id` attribute, the JavaServer Faces implementation automatically generates a component ID.

`UIOutput` and subclasses of `UIOutput` have `value` and `binding` attributes, which are always optional, except in the case of `SelectItems`, which requires a `value` attribute. Using the `value` and `binding` attributes is explained more in Binding Component Values and Instances to External Data Sources (page 860).

For each of the components discussed in this section, Working With Backing Beans (page 839) explains how to write a bean property bound to a particular UI component or its value.

The UIForm Component

A `UIForm` component is an input form with child components representing data that is either presented to the user or submitted with the form. The `form` tag encloses all of the controls that display or collect data from the user. Here is an example:

```
<h:form>
... other faces tags and other content...
</h:form>
```

The `form` tag can also include HTML markup to layout the controls on the page. The `form` tag itself does not perform any layout; its purpose is to collect data and to declare attributes that can be used by other components in the form.

The UIColumn Component

The `UIColumn` component represents a column of data in a `UIData` component. While the `UIData` component is iterating over the rows of data, it processes the

UIColumn for each row. UIColumn has no renderer associated with it and is represented on the page with a column tag. Here is an example column tag from the orderForm page of the CoffeeBreak example:

```
<h:data_table ...
  value="#{CoffeeBreakBean.cart.items}"
  var="sci"
  ...
  <h:column>
    <f:facet name="header">
      <h:output_text  value="Coffee"/>
    </f:facet>
    <h:output_text id="coffeeName"
      value="#{sci.item.coffeeName}"/>
  </h:column>
  ...
</h:data_table>
```

In this example, the UIData component iterates through a list of beans (CoffeeBreakBean.cart.items). While doing that, it first renders a column header with the name "Coffee", then renders a cell in the column. Each cell contains a coffee name for each bean in the list of beans.

The UICommand Component

The UICommand component performs an action when it is activated. The most common example of such a component is the button. This release supports Button and Link as UICommand component renderers.

In addition to the tag attributes listed in Using the HTML Tags (page 817), the command_button and command_link tags can use these attributes:

- action, which is either a logical outcome String or a JSF EL expression that points to a bean method that returns a logical outcome String. In either case, the logical outcome String is used by the navigation system to determine what page to access when the UICommand component is activated
- actionListener, which is a JSF EL expression that points to a bean method that processes an ActionEvent fired by the UICommand component

See Navigating Between Pages (page 896) for more information on using the action attribute.

See Writing a Method to Handle an ActionEvent (page 849) for details on using the `actionListener` attribute.

Using the command_button Tag

Most pages in the `cardemo` example use the `command_button` tag. When a user clicks the button, the data from the current page is processed, and the next page is opened. Here is a `command_button` tag from `carDetail.jsp`:

```
<h:command_button action="#{carstore.buyCurrentCar}"
  value="#{bundle.buy}" />
```

Clicking the button will cause the `confirmChoices.jsp` page to open. This page allows a user to fill in his name and shipping information.

The `value` attribute references the localized message for the button's label. The `bundle` part of the expression refers to the `ResourceBundle` that contains a set of localized messages. The `buy` part of the expression is the key that corresponds to the message that is displayed on the button. For more information on localizing JavaServer Faces applications, see Performing Localization (page 903).

The `action` attribute references a method on the `CarStore` backing bean that performs some processing and returns an outcome. The outcome is passed to the default `NavigationHandler`, which matches the outcome against a set of navigation rules defined in the application configuration file.

See Navigating Between Pages (page 896) for information on how to use the `action` attribute.

Using the command_link Tag

The `command_link` tag represents an HTML hyperlink and is rendered as an HTML <a> element. The `command_link` tag is used to submit an `ActionEvent` to the application. See Handling Events (page 889) for more information on action events.

A `command_link` tag must have an `output_text` tag nested inside it. This tag represents the text the user clicks to generate the event. The following tag is from the `chooseLocale` page from the `cardemo` application.

```
<h:command_link id="NAmerica" action="storeFront"
  actionListener="#{carstore.chooseLocaleFromLink}">
  <h:output_text value="#{bundle.english}" />
</h:command_link>
```

This tag will render the following HTML:

```
<a href="#"
  onmousedown="document.forms[1]['_id4:NAmerica'].
    value='_id4:NAmerica';
      document.forms[1].submit()">English</a>
    <input type="hidden" name="_id4:NAmerica" /></td>
```

Note: Notice that the `command_link` tag will render JavaScript. If you use this tag, make sure your browser is JavaScript-enabled.

The UIData Component

The `UIData` component supports data binding to a collection of data objects. The `Table` renderer displays the data as an HTML table. The `UIColumn` component represents a column of data within the `UIData` component. The `UIData` component does the work of iterating over each record in the data source and displaying a row in the table for it. Here is a portion of the `data_table` tag used by the `orderForm` page of the Coffee Break example:

```
<h:data_table id="table" columnClasses="list-column-
center,list-column-right, list-column-center, list-column-
right" headerClass="list-header" rowClasses="list-row"
styleClass="list-background"
  value="#{CoffeeBreakBean.cart.items}" var="sci">
  <f:facet name="header">
    <h:output_text  value="#{CBMessages.OrderForm}"/>
  </f:facet>
  <h:column>
    <f:facet name="header">
      <h:output_text  value="Coffee"/>
    </f:facet>
    <h:output_text id="coffeeName"
      value="#{sci.item.coffeeName}"/>
  </h:column>
  <h:column>
    <f:facet name="header">
      <h:output_text  value="Price"/>
    </f:facet>
    <h:output_text id="retailPricePerPound"
      value="#{sci.item.retailPricePerPound}">
      <f:convert_number type="number"/>
    </h:output_text>
  </h:column>
```

```
      <h:column>
         <f:facet name="header">
            <h:output_text  value="Quantity"/>
         </f:facet>
         <h:input_text id="quantity" required="true" size="3"
            value="#{sci.pounds}" >
            <f:convert_number type="number"/>
         </h:input_text>
      </h:column>
      <h:column>
         <f:facet name="header">
            <h:output_text value="Total" />
         </f:facet>
         <h:output_text value="#{sci.price}">
            <f:convert_number type="number"/>
         </h:output_text>
         <f:facet name="footer">
            <h:output_text  value="#{CoffeeBreakBean.cart.total}"/>
         </f:facet>
      </h:column>
   </h:data_table>
```

This `data_table` tag renders the table shown in Figure 21–1.

OrderForm			
Coffee	**Price**	**Quantity**	**Total**
Wake Up Call	7.43	0.0	0.00
French Roast	6.75	0.0	0.00
Kona	8.78	0.0	0.00
Mocca	5.40	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
			0.00

Figure 21–1 Table on orderForm page

This tag displays a list of coffees, their prices per pound, and a set of input fields into which the user can enter the number of pounds of each coffee he wants to buy, and a total price. The first `facet` tag inside of `data_table` renders a header for the entire table. The `facet` tags inside of each column render the headings for each column.

This example is a classic use case for a `UIData` component because the number of coffees is not known to the application developer or the page author at the time the application is developed. The `UIData` component can dynamically adjust the number of rows of the table to accommodate the underlying data.

The `value` attribute of `data_table` references the data to be included in the table. This data can take the form of:

- A list of beans
- An array of beans
- A single bean
- A `javax.faces.model.DataModel`
- A `java.sql.ResultSet`
- A `javax.servlet.jsp.jstl.sql.ResultSet`
- A `javax.sql.RowSet`

All data sources for `UIData` components have a `DataModel` wrapper. Unless you explicitly construct a `DataModel`, the JavaServer Faces implementation will create a `DataModel` wrapper around data of any of the other acceptable types. See Writing Component Properties (page 840) for more information on how to write properties for use with a `UIData` component.

The `var` attribute specifies a name that the components within the `data_table` tag use as an alias to the data referenced in the `value` attribute of `data_table`.

The `UIData` component also has the ability to display only a subset of the underlying data. This is not shown in the example above. To display a subset of the data, you use the optional `first` and `rows` attributes.

The `first` attribute specifies the first row to be displayed. The `rows` attribute specifies the number of rows—starting with the first row—to be displayed. By default, both `first` and `rows` are set to zero, which causes all of the rows of the underlying data to display. As an example, if you wanted to display records 2 through 10 of the underlying data, you would set `first` to 2 and `rows` to 9. When you display a subset of the data in your pages, you might want to consider including a link or button that causes subsequent rows to display when clicked.

The `data_table` tag also has a set of optional attributes for adding styles to the table:

- `columnClasses`: Defines styles for all of the columns
- `footerClass`: Defines styles for the footer
- `headerClass`: Defines styles for the header
- `rowClasses`: Defines styles for the rows
- `styleClass`: Defines styles for the entire table

Each of these attributes can specify more than one style. If `columnClasses` or `rowClasses` specifies more than one style, the styles are applied to the columns or rows in the order that the styles are listed in the attribute. In other words, if `columnClasses` specifies styles `list-column-center` and `list-column-right`, and there are two columns in the table, the first column will have style `list-column-center`, and the second column will have style `list-column-right`.

If the `style` attribute specifies more styles than there are columns or rows, the remaining styles will be assigned to columns or rows starting from the first row. Likewise, if the `style` attribute specifies less styles than there are columns or rows, the remaining columns or rows will be assigned styles starting from the first style.

The UIGraphic Component

The `UIGraphic` component displays an image. The `carstore` application has many examples of `graphic_image` tags. Here is the `graphic_image` tag used with the image map on `chooseLocale.jsp`:

```
<h:graphic_image id="mapImage" url="/images/world.jpg"
  alt="#{bundle.chooseLocale}" usemap="#worldMap" />
```

The `url` attribute specifies the path to the image. It also corresponds to the local value of the `UIGraphic` component so that the URL can be retrieved with the `currentValue` method or indirectly from a backing bean. The URL of the example tag begins with a "/", which adds the relative context path of the Web application to the beginning of the path to the image.

The `alt` attribute specifies the alternative text displayed when the user mouses over the image. In this example, the `alt` attribute refers to a localized message. See Performing Localization (page 903) for details on how to localize your JavaServer Faces application.

The `usemap` attribute refers to the image map defined by the custom `MapComponent` component on the same page. See Chapter 22 for more information on the image map.

The UIInput and UIOutput Components

The UIInput component displays a value to a user and allows the user to modify this data. The most common example is a text field. The UIOutput component displays data that cannot be modified. The most common example is a label.

Both UIInput and UIOutput components can be rendered in several different ways. Table 21–3 lists the renderers of UIInput and UIOutput. Recall from The Component Rendering Model (page 792) that the tags are composed of the component and the renderer. For example, the input_text tag refers to a UIInput component that is rendered with the Text renderer

Table 21–3 UIInput and UIOutput Renderers

Component	Renderer	Tag	Function
UIInput	Hidden	input_hidden	Allows a page author to include a hidden variable in a page
	Secret	input_secret	Accepts one line of text with no spaces and displays it as a set of asterisks as it is typed
	Text	input_text	Accepts a text string of one line
	TextArea	input_textarea	Accepts multiple lines of text
UIOutput	Label	output_label	Displays a nested component as a label for a specified input field
	Link	output_link	Display an <a href > tag that links to another page without generating an ActionEvent
	Message	output_message	Displays a localized message
	Text	output_text	Displays a text string of one line

You can apply the DateTime and Number converters to convert data associated with the input_text, output_text, input_hidden, and input_secret tags. See Performing Data Conversions (page 880) for more information on using these renderers as converters.

The rest of this section explains how to use selected tags listed in the Table 21–3. These tags are: `output_label`, `output_link`, `output_message`, `input_secret`, `output_text`, and `input_text`.

The `input_hidden` and `input_textarea` tags are similar to the `input_text` tag. Refer to the `html_basic` TLD, located in the lib directory of your JavaServer Faces technology download, to see what attributes are supported for tags not documented here.

Using the output_text and input_text Tags

The `Text` renderer can render both `UIInput` and `UIOutput` components. The `input_text` tag displays and accepts a single-line string. The `output_text` tag displays a single-line string. This section shows you how to use the `input_text` tag. The `output_text` tag is written in a similar way.

The following attributes, supported by both `output_text` and `input_text`, are likely to be the most commonly used:

- `id`: Identifies the component associated with this tag
- `value`: Identifies the local value or an external data source, such as a backing bean property or localized message, bound to the component
- `converter`: Identifies a converter that will be used to convert the component's local data. See Performing Data Conversions (page 880) for more information on how to use this attribute.
- `validator`: Is a JSF EL expression pointing to a backing-bean method that performs validation on the component's data. See Reference the Validator Method from the Component Tag (page 879) for an example of using `validator`.
- `valueChangeListener`: Is a JSF EL expression that points to a backing-bean method that handles the event of entering a value in this component. See Referencing a Method That Handles a ValueChangeEvent (page 896) for an example of using `valueChangeListener`.

Here is an example of an `input_text` tag from the `customerInfo` page:

```
<h:input_text value="#{CustomerBean.firstName}" />
```

The `value` attribute refers to the `firstName` property on the `CustomerBean` bean. After the user submits the form, the value of the `firstName` property in `CustomerBean` will be set to the text entered in the field corresponding to this tag.

Using the output_label Tag

The `output_label` tag is used to attach a label to a specified input field for accessibility purposes. Here is an example of an `output_label` tag:

```
<h:output_label for="firstName">
  <h:output_text id="firstNameLabel" value="First Name"/>
</h:output_label>
...
<h:input_text id="firstName" />
```

The `for` attribute maps to the `id` of the input field to which the label is attached. The `output_text` tag nested inside the `output_label` tag represents the actual label component. The `value` attribute on the `output_text` tag indicates the label that is displayed next to the input field.

Using the output_link Tag

The `output_link` tag is used to render a hyperlink that, when clicked, loads another page but does not generate an `ActionEvent`. Here is an example of an `output_link` tag:

```
<h:output_link value="javadocs">
  <f:verbatim>Documentation for this demo</f:verbatim>
</h:output_link>
```

As shown in this example, the `output_link` tag requires a nested `verbatim` tag, which identifies the text the user clicks to get to the next page.

Using the output_message Tag

The `output_message` tag allows a page author to display concatenated messages as a `MessageFormat` pattern. Here is an example of an `output_message` tag from the `finish` page of the `cardemo` application:

```
<h:output_message  value="#{bundle.thanksLabel}">
  <f:parameter value="#{sessionScope.firstName}"/>
</h:output_message>
```

The `value` attribute specifies the `MessageFormat` pattern. The `parameter` tag specifies the substitution parameters for the message. The `value` for the parameter maps to the user's first name, located in session scope. When the message is displayed in the page, the first name in the session scope replaces the `{0}` in the message located at the `thanksLabel` key in the `bundle` resource bundle:

```
Thanks, {0}, for using carstore. Your car will ship soon.
```

If you have more than one parameter for one message, make sure you put the `parameter` tags in the proper order so that the data is inserted in the correct place in the message.

A page author can also hardcode the data to be substituted in the message using the `value` attribute on the `parameter` tag.

Using the input_secret Tag

The `input_secret` tag renders an `<input type="password">` HTML tag. When the user types a string in this field, a row of asterisks is displayed instead of the string the user types. Here is an example of an `input_secret` tag:

```
<h:input_secret redisplay="false"
  value="#{LoginBean.password}" />
```

In this example, the `redisplay` attribute is set to false. This will prevent the password from being displayed in a query string or in the source file of the resulting HTML page.

The UIPanel Component

A `UIPanel` component is used as a layout container for its children. When using the renderers from the HTML render kit, a `UIPanel` is rendered as an HTML table. This component differs from `UIData` in that `UIData` can dynamically add or delete rows to accommodate the underlying data source, whereas a `UIPanel` must have the number of rows predetermined. Table 21–4 lists all of the renderers and tags corresponding to the `UIPanel` component.

Table 21–4 UIPanel Renderers and Tags

Renderer	Tag	Renderer Attributes	Function
`Grid`	`panel_grid`	`columnClasses, columns, footerClass, headerClass, panelClass, rowClasses`	Displays a table
`Group`	`panel_group`		Groups a set of components under one parent

The `panel_grid` tag is used to represent entire tables. The `panel_group` tag is used to represent rows in the tables. To represent individual cells in the rows, the `output_text` tag is usually used, but any output component tag can be used to represent a cell.

This section shows you how to create tables with `panel_grid` and how to use the `panel_group` tag to generate rows for the tables.

The `panel_grid` tag has a set of attributes that specify CSS stylesheet classes: the `columnClasses`, `footerClass`, `headerClass`, `panelClass`, and `rowClasses`. These stylesheet attributes are not required.

The `panel_grid` tag also has a `columns` attribute. The `columns` attribute is required if you want your table to have more than one column because the `columns` attribute tells the renderer how to group the data in the table.

If a `headerClass` is specified, the `panel_grid` must have a header as its first child. Similarly, if a `footerClass` is specified, the `panel_grid` must have a footer as its last child.

The `cardemo` application includes one `panel_grid` tag on the `confirmChoices.jsp` page:

```
<h:panel_grid columns="2" footerClass="subtitle"
  headerClass="subtitlebig" styleClass="medium"
  columnClasses="subtitle,medium">
  <f:facet name="header">
    <h:output_text value="#{bundle.buyTitle}"/>
  </f:facet>
  <h:output_text value="#{bundle.Engine}" />
  <h:output_text
    value="#{carstore.currentModel.attributes.engine}"
  />
  <h:output_text value="#{bundle.Brakes}" />
  ...
  <f:facet name="footer">
    <h:panel_group>
      <h:output_text value="#{bundle.yourPriceLabel}"  />
      <h:output_text
        value="#{carstore.currentModel.currentPrice}" />
    </h:panel_group>
  </f:facet>
  ...
</h:panel_grid>
```

This `panel_grid` is rendered to a table that lists all of the options that the user chose on the previous page, `carDetail`. This `panel_grid` uses stylesheet classes to format the table. The CSS classes are defined in the `stylesheet.css` file in the `<JSF_HOME>/samples/cardemo/web` directory of your installation. The `subtitlebig` definition is:

```
.subtitlebig {
  font-family: Arial, Helvetica, sans-serif;
  font-size: 14px;
  color: #93B629;
  padding-top: 10;
  padding-bottom: 10;
}
```

Since the `panel_grid` tag specifies a `headerClass` and a `footerClass`, the `panel_grid` must contain a header and footer. Usually, a `facet` tag is used to represent headers and footers. This is because header and footer data is usually static.

A *facet* is used to represent a component that is independent of the parent-child relationship of the page's component tree. Since header and footer data is static, the elements representing headers and footers should not be updated like the rest of the components in the tree.

The example `panel_grid` uses a `facet` tag for the footer. Facets can have only one child, and so a `panel_group` tag is needed to group more than one element within a `facet`.

The `panel_group` tag is needed within the `footer` facet tag because the footer requires two cells of data, represented by the two `output_text` tags within the `panel_group` tag:

```
<f:facet name="footer">
  <h:panel_group>
    <h:output_text value="#{bundle.yourPriceLabel}"  />
    <h:output_text
      value="#{carstore.currentModel.currentPrice}" />
  </h:panel_group>
</f:facet>
```

A `panel_group` tag can also be used to encapsulate a nested tree of components so that the tree of components appears as a single component to the parent component.

In between the `header` and `footer` facet tags, are the `output_text` tags, each of which represents a cell of data in the table.

The data represented by the `output_text` tags is grouped into rows according to the value of the `columns` attribute of the `output_text` tag. The `columns` attribute in the example is set to "2". So from the list of `output_text` tags representing the table data, the data from the odd `output_text` tags is rendered in the first column and the data from the even `output_text` tags is rendered in the second column.

The UISelectBoolean Component

The `UISelectBoolean` class defines components that have a `boolean` value. The `selectboolean_checkbox` tag is the only tag that JavaServer Faces technology provides for representing boolean state. The `optionsPanel` page of the `cardemo` application has a set of `selectboolean_checkbox` tags. Here is the one representing the `cruisecontrol` component:

```
<h:selectboolean_checkbox  title="#{bundle.cruiseLabel}"
  binding=
    "#{carstore.currentModel.components.cruisecontrol}" >
</h:selectboolean_checkbox>
<h:output_text value="#{bundle.cruiseLabel}" />
```

The `binding` attribute refers to the backing-bean property that holds the component instance. Alternatively, the tag can map to a bean property that holds the component value, which in this case must be of type `boolean` or `Boolean`. This property must be referenced with the `value` attribute using the same JSF EL syntax that is shown in the `binding` attribute value in the example tag in this section. See section UISelectBoolean Properties (page 844) for information on how to write a backing-bean property that can be bound to a `UISelectBoolean` component.

The UISelectMany Component

The `UISelectMany` class defines components that allow the user to select zero or more values from a set of values. This component can be rendered as a set of checkboxes, a listbox, or a menu. This section explains the `selectmany_checkboxlist` and `selectmany_menu` tags. The `selectmany_listbox` tag is similar to the `selectmany_menu` tag, except `selectmany_listbox` does not have a `size` attribute since a listbox displays all items at once.

Using the selectmany_checkboxlist Tag

The `selectmany_checkboxlist` tag renders a set of checkboxes with each checkbox representing one value that can be selected. The `cardemo` does not have an example of a `selectmany_checkboxlist` tag, but this tag can be used to render the checkboxes on the `optionsPanel.jsp` page:

```
<h:selectmany_checkboxlist
  value="#{MyCarBean.currentOptions}">
  <h:selectitem itemLabel="Sunroof"
    value="#{MyCarBean.sunRoofSelected}">
  </h:selectitem>
  <h:selectitem itemLabel="Cruise Control"
```

```
      value=
         "#{MyCarBean.cruiseControlSelected}" >
    </h:selectitem>
  </h:selectmany_checkboxlist>
```

The `value` attribute identifies the backing-bean property, `currentOptions`, for the current set of options. This property holds the values of the currently selected items from the set of checkboxes.

The `selectmany_checkboxlist` tag must also contain a tag or set of tags representing the set of checkboxes. To represent a set of items, you use the `selectitems` tag. To represent each item individually, use a `selectitem` tag for each item. UISelectItem and UISelectItems (page 837) section explains these two tags in more detail.

Using the selectmany_menu Tag

The `selectmany_menu` tag represents a component that contains a list of items, from which a user can choose one or more items. The menu is also commonly known as a drop-down list or a combo box. The tag representing the entire list is the `selectmany_menu` tag. Here is an example of a `selectmany_menu` tag:

```
<h:selectmany_menu  id="fruitOptions"
  value="#{FruitOptionBean.chosenFruits}”>
  <h:selectitems
     value="#{FruitOptionBean.allFruits}"/>
</h:selectmany_menu>
```

The attributes of the `selectmany_menu` tag are the same as those of the `selectmany_checkboxlist` tag. Again, the `value` of the `selectmany_menu` tag maps to the property that holds the currently selected items' values. A `selectmany_menu` tag can also have a `size` attribute, whose value specifies how many items will display at one time in the menu. When the `size` attribute is set, the menu will render with a scrollbar for scrolling through the displayed items.

Like the `selectmany_checkboxlist` tag, the `selectmany_menu` tag must contain either a `selectitems` tag or a set of `selectitem` tags for representing the items in the list. The `value` attribute of the `selectitems` tag in the example maps to the property that holds all of the items in the menu. UISelectItem and UISelectItems (page 837) explains these two tags.

The UIMessage and UIMessages Components

The `UIMessage` and `UIMessages` components are used to display error messages. Here is an example from `guessNumber`:

```
<h:input_text id="userNo" value="#{UserNumberBean.userNumber}"
  validate="UserNumberBean.validate"/>
...
<h:messages
  style="color: red;
  font-family: 'New Century Schoolbook', serif;
  font-style: oblique;
  text-decoration: overline" id="errors1" for="userNo"/>
```

The `for` attribute refers to the ID of the component that generated the error messages. The messages tag will display the error messages wherever the messages tag appears in the page.

The `style` attribute allows you to specify the style of the text of the message. In the example in this section, the text will be red, New Century Schoolbook, serif font family, oblique style, and a line will appear over the text.

If you use the `message` tag instead of the `messages` tag, only the first applicable error message will display.

The UISelectOne Component

The `UISelectOne` class defines components that allow the user to select one value from a set of values. This component can be rendered as a listbox, a radio button, or a menu. This section explains the `selectone_radio` and `selectone_menu` tags. The `selectone_listbox` tag is similar to the `selectone_menu` tag, except `selectone_listbox` does not have a `size` attribute since a listbox displays all items at once. This section explains how to use the `selectone_radio` and `selectone_menu` tags.

Using the selectone_radio Tag

The selectone_radio tag renders a set of radio buttons, in which each radio button corresponds to one value that can be selected. Here is an example selectone_radio tag:

```
<h:selectone_radio  id="chooseAFruit"
  value="#{Produce.currentFruit}">
  <h:selectitems
    value="#{Produce.fruits}"/>
</h:selectone_radio>
```

The id attribute of the selectone_radio tag uniquely identifies the radio group. The id is only required if another component, backing bean, or listener must refer to this component; otherwise, the JavaServer Faces implementation will generate a component id for you.

The value attribute identifies the backing-bean property, currentFruit. This property holds the value of the currently selected item from the set of radio buttons. The currentFruit property can be any of the types supported by JavaServer Faces technology.

The selectone_radio tag must also contain a tag or set of tags representing the list of items contained in the radio group. To represent a set of tags, you use the selectitems tag. To represent each item individually, use a selectitem tag for each item. UISelectItem and UISelectItems (page 837) explains these two tags in more detail.

Using the selectone_menu Tag

The selectone_menu tag represents a component that contains a list of items, from which a user can choose one item. The menu is also commonly known as a drop-down list or a combo box. The tag representing the entire list is the selectone_menu tag. Here is an example selectone_menu tag:

```
<h:selectone_menu  id="chooseAFruit"
  value="#{Produce.currentFruit}">
  <h:selectitems
    value="#{Produce.fruits}"/>
</h:selectone_menu>
```

The attributes of the selectone_menu tag are the same as those of the selectone_radio tag. Again, the value of the selectone_menu tag maps to the property that holds the currently selected item's value. A selectone_menu tag can also have a size attribute, whose value specifies how many items will display at one time in the menu. When the `size` attribute is set, the menu will render with a scrollbar for scrolling through the displayed items.

Like the `selectone_radio` tag, the `selectone_menu` tag must contain either a `selectitems` tag or a set of `selectitem` tags for representing the items in the list. UISelectItem and UISelectItems (page 837) section explains these two tags.

UISelectItem and UISelectItems

`UISelectItem` and `UISelectItems` represent components that can be nested inside a `UISelectOne` or a `UISelectMany` component. The `UISelectItem` is associated with a `SelectItem` instance, which contains the value, label, and description of a single item in the `UISelectOne` or `UISelectMany` component. The `UISelectItems` instance represents a set of `SelectItem` instances, containing the values, labels, and descriptions of the entire list of items.

The `selectitem` tag represents a `SelectItem`. The `selectitems` tag represents a `UISelectItems` component. You can use either a set of `selectitem` tags or a single `selectitems` tag within your `selectone` or `selectmany` tags.

The advantages of using `selectitems` are

- You can represent the items using different data structures, including `Array`, `Map`, `List`, and `Collection`. The data structure is composed of `SelectItem` instances.
- You can dynamically generate a list of values at runtime.

The advantages of using `selectitem` are:

- The page author can define the items in the list from the page.
- You have less code to write in the bean for the `selectitem` properties.

For more information on writing component properties for the `UISelectItems` components, see Writing Component Properties (page 840). The rest of this section shows you how to use the `selectitems` and `selectitem` tags.

The selectitems Tag

Here is the `selectone_menu` tag from the section The UISelectOne Component (page 835):

```
<h:selectone_menu  id="chooseAFruit"
  value="#{Produce.currentFruit}">
  <h:selectitems
     value="#{Produce.fruits}"/>
</h:selectone_menu>
```

The `id` attribute of the `selectitems` tag refers to the `UISelectItems` component object.

The `value` attribute binds the `selectitems` tag to the `fruits` property of the `Produce` bean.

In the `Produce` bean, the `fruits` property has a type of `ArrayList`:

```
fruits = new ArrayList(fruitItems.length);
```

`UISelectItems` is a collection of `SelectItem` instances. You can see this by noting how the `fruits` `ArrayList` is populated:

```
for (i = 0; i < fruitItems.length; i++) {
  fruits.add(new SelectItem(fruitItems[i], fruitItems[i],
     fruitItems[i]));
}
```

The arguments to the `SelectItem` constructor are:

- An `Object` representing the value of the item
- A `String` representing the label that displays in the `UISelectOne` component on the page
- A `String` representing the description of the item

The section UISelectItems Properties (page 847) describes in more detail how to write a backing-bean property for a `UISelectItems` component

The selectitem Tag

Let's see how the `chooseAFruit` tag from the previous section would look if you used `selectitem` instead of `selectitems`:

```
<h:selectone_menu id="chooseAFruit"
  value="apple">
  <h:selectitem
    itemValue="apple" itemLabel="apple"/>
  <h:selectitem
    itemValue="banana" itemLabel="banana"/>
  <h:selectitem
    itemValue="pear" itemLabel="pear"/>
</h:selectone_menu>
```

The `itemValue` attribute represents the default value of the `SelectItem` instance. The `itemLabel` attribute represents the `String` that appears in the dropdown list component on the page.

You can also use the `value` attribute instead of the `itemValue` attribute to refer to a bean property that represents the item's value.

Working With Backing Beans

As explained in Backing Bean Management (page 799), a backing bean is typically associated with a JavaServer Faces page that includes a form. The backing bean defines properties that can hold the data or instances of the components on the page. It can also include special methods that perform certain functions associated with the components on the page. These functions include validation, event handling, and navigation processing.

Including the component properties and the special methods in the same bean eases the development process. This is because the methods that perform functions for the components can more readily access the component data if it's needed.

This section explains how to:

- Write component properties and backing-bean methods
- Configure component properties in the application configuration file
- Bind components and their values to backing-bean properties
- Reference backing-bean methods from component tags

Writing Component Properties

As explained in Backing Bean Management (page 799), there are two kinds of backing-bean properties: those that are bound to a component's value and those that are bound to a component instance.

The component tag binds the component's value to a property using its `value` attribute. The component tag binds the component instance to a property using its `binding` attribute. Using the attributes to bind components and their values to properties is discussed in Binding Component Values and Instances to External Data Sources (page 860).

In order to bind a component's value to a backing bean property, the type of the property must match the type of the component's value to which it is bound. In other words, if a backing bean property is bound to a `UISelectBoolean` component's value, the property should accept and return a `boolean` value or a `Boolean` wrapper `Object` instance.

In order to bind a component instance to a backing bean property, the type of the property must be the component itself. In other words, if a backing-bean property is bound to a `UISelectBoolean` instance, the property should accept and return a `UISelectBoolean`.

The rest of this section explains how to write properties that can be bound to component values and component instances for the component objects described in the Using the HTML Tags (page 817) section. Many of the example tags in that section come from the `carDetail` page of the `cardemo` application. This page displays the options for the chosen car.

The components of the `carDetail` page are bound to component properties of `CarBean`. This bean has a unique way of storing the component data. Rather than providing the usual bean property for a component's data, it has a mechanism for accessing the data and the component types from resource bundles.

While reading the data from the resource bundle, `CarBean` dynamically creates a `Map` of component objects, which get exposed to the page using JSF EL expressions. In addition, the `CarBean` wraps this component map with a read-only map that exposes only the values of the instances. The `binding` attribute of a component tag references the component in the read-only `Map`, as shown in this tag:

```
<h:selectboolean_checkbox  title="#{bundle.towPkgLabel}"
  alt="#{bundle.towPkgLabel}"
  binding="#{carstore.currentModel.components.towPackage}" >
</h:selectboolean_checkbox>
```

The component tag corresponding to the tow package checkbox uses a JSF EL expression to access its corresponding component, which is stored in the components `Map` in `CarBean` under the name "towPackage". Furthermore, when the page is submitted, the value of the checkbox is stored as the local value of the `UISelectBooleanCheckbox` instance.

This technique for storing component data and accessing it from the page might be a little complicated for most simple applications. Therefore, this section uses examples that don't generate UI components from a `ResourceBundle`.

Instead, this section describes how to write traditional bean properties that you can reference from the `value` and `binding` attributes of component tags using JSF EL expressions. For more information on the technique used in `CarBean`, please see the class comments for `CarBean`.

Writing Properties Bound to Component Values

To write a backing bean property bound to a component's value, you need to know the types that the component accepts.

Table 21–5 lists all the component classes described in Using the HTML Tags (page 817) and the acceptable types of their values

When referencing the property with the component tag's `value` attribute, be sure to use the proper syntax. For example, if a `value` attribute has a value of `"#{MyBean.currentOption}"`, the corresponding `String` property of `MyBean` should be:

```
int currentOption = null;
int getCurrentOption(){...}
void setCurrentOption(int option){...}
```

For more information on JavaBeans conventions, see JavaBeans Components (page 663).

Table 21–5 Acceptable Component Types

Component	Types
`UIInput`, `UIOutput`, `UISelectItem`, `UISelectOne`	Any of the standard converters

Table 21–5 Acceptable Component Types (Continued)

Component	Types
UIData	array of beans ,java.util.Collection of beans, single bean, java.sql.ResultSet, javax.servlet.jsp.jstl.sql.Result, javax.sql.RowSet
UISelectBoolean	boolean
UISelectItems	java.lang.String, Collection, Array, Map
UISelectMany	array or Collection. Elements of the array or Collection can be any of the standard types.

UIInput and UIOutput Properties

The firstName UIInput component of the customerInfo page of cardemo is mapped to an Integer property in UserNumber.java:

```
<h:input_text  id="firstName"
  value="#{customer.firstName}" required="true">
```

Here is the property mapped to the firstName component tag:

```
String firstName = null;
...
public void setFirstName(String first) {
  firstName = first;
}

public String getFirstName() {
  return firstName;
}
```

The values of the UIInput and UIOutput components can be converted to any of the standard types by either applying a Converter or by creating the bean property bound to the component with the desired type. See Performing Data Conversions (page 880) for more information on applying a Converter.

UIData Properties

UIData components must be bound to one of the types listed in Table 21–5. The UIData component from the orderForm page of the CoffeeBreak example is discussed in the section The UIData Component (page 821). Here is part of the tag from that section:

```
<h:data_table  id="table"
  ...
  value="#{CoffeeBreakBean.cart.items}"
  var="sci" >
```

The JSF EL expression points to the items ArrayList of the ShoppingCart bean instantiated in the CoffeeBreakBean. The items ArrayList is composed of ShoppingCartItem beans.

This code snippet from ShoppingCart.java shows the ArrayList being populated and the accessor method:

```
items = new ArrayList();
for(Iterator i = rpl.getItems().iterator(); i.hasNext(); ) {
  RetailPriceItem item = (RetailPriceItem) i.next();
  ShoppingCartItem sci = new ShoppingCartItem(item, new
    BigDecimal(0.0), new BigDecimal("0.00"));
  items.add(sci);
  numberOfItems++;
}...

public synchronized ArrayList getItems() {
  return items;
}
```

All the components contained in the UIData component are bound to the properties on the ShoppingCartItem bean that is bound to the entire UIData component. For example, here is the input_text tag that represents the quantity field:

```
<h:input_text id="quantity"
  required="true"
  size="3"
  value="#{sci.pounds}" >
```

The `UIInput` component corresponding to this tag is bound to the `pounds` property on the `ShoppingCartItem` bean:

```
public void setPounds(BigDecimal newPounds) {
  this.pounds=newPounds;
}
```

UISelectBoolean Properties

Properties that hold this component's data must be of `boolean` or `Boolean` type. Here is an example `selectboolean_checkbox` tag:

```
<h:selectboolean_checkbox
  title="fruits"
  value="#{Produce.aFruit}" >
</h:selectboolean_checkbox>
```

Here is an example property that can be bound to the component represented by the example tag:

```
  protected boolean aFruit = false;
     ...
  public void aFruit(boolean a_fruit) {
     aFruit = a_Fruit;
  }
  public boolean aFruit() {
     return aFruit;
  }
```

UISelectMany Properties

Since a `UISelectMany` component allows a user to select one or more items from a list of items, this component must map to a bean property of type `java.util.Collection` or `array`. This bean property represents the set of currently selected items from the list of available items.

Here is an example `selectmany_checkbox` tag:

```
<h:selectmany_checkbox
  value="#{Produce.currentFruits}">
  <h:selectitem itemLabel="Apple"
     value="#{currentFruits.appleSelected}">
  </h:selectitem>
```

```
    <h:selectitem itemLabel="Pear"
      value="#{currentFruits.pearSelected}" >
    </h:selectitem>
  </h:selectmany_checkboxlist>
```

Here is a bean property that maps to the `value` of this `selectmany_checkbox` example:

```
protected ArrayList currentFruits = null;

public Object[] getCurrentFruits() {
  return currentFruits.toArray();
}
public void setCurrentFruits(Object[] newFruits) {
  int len = 0;
  if (null == newFruits ||
    (len = newFruits.length) == 0) {
      return;
  }
  currentFruits.clear();
  currentFruits = new ArrayList(len);
  for (int i = 0; i < len; i++) {
    currentFruits.add(newFruits[i]);
  }
}
```

Note that the `setFruits(Object)` method must clear the `Collection` and rebuild it with the new set of values that the user selected.

As explained in the section The UISelectMany Component (page 833), the `UISelectItem` and `UISelectItems` components are used to represent all the values in a `UISelectMany` component. See UISelectItem Properties (page 846) and UISelectItems Properties (page 847) for information on how to write the bean properties for the `UISelectItem` and `UISelectItems` components.

UISelectOne Properties

`UISelectOne` properties accept the same types as `UIInput` and `UIOutput` properties. This is because a `UISelectOne` component represents the single selected item from a set of items. This item could be a `String`, `int`, `long`, or `double`.

Here is an example `selectone_radio` tag:

```
<h:selectone_radio  id="currentFruit"
  value="#{Produce.currentFruit}">
  <h:selectitems
     value="#{Produce.currentFruits}"/>
</h:selectone_radio>
```

Here is a property corresponding to this tag:

```
protected Object currentFruit = fruits[0];
...
public void setCurrentFruit(Object aFruit) {
  currentFruit = eng;
}

public Object getCurrentFruit() {
  return currentFruit;
}
```

Note that `currentFruit` is one of the objects in an array of objects, that represent the list of items in the `UISelectOne` component.

As explained in the The UISelectOne Component (page 835) section, the `UISelectItem` and `UISelectItems` components are used to represent all the values in a `UISelectOne` component. See UISelectItem Properties (page 846) and UISelectItems Properties (page 847) for information on how to write the backing bean properties for the `UISelectItem` and `UISelectItems` components.

UISelectItem Properties

A `UISelectItem` component represents one value in a set of values in a `UISelectMany` or `UISelectOne` component. A `UISelectItem` property must be mapped to a property of type `SelectItem`. A `SelectItem` object is composed of: an `Object` representing the value, and two `Strings` representing the label and description of the `SelectItem`.

Here is an example bean property for a `SelectItem` component:

```
SelectItem itemOne = null;

SelectItem getItemOne(){
  return SelectItem(String value, String label, String
     description);
}
```

```
void setItemOne(SelectItem item) {
  itemOne = item;
}
```

UISelectItems Properties

The `UISelectItems` properties are the most difficult to write and require the most code. `UISelectItems` components are typically children of `UISelectMany` and `UISelectOne` components. Each `UISelectItems` component is composed of a set of `SelectItem` instances. In your backing bean, you must define a set of `SelectItem` objects, set their values, and populate the `UISelectItems` object with the `SelectItem` objects. Here is an example code snippet that shows how to create a `SelectItems` property:

```
import javax.faces.component.SelectItem;
...
protected ArrayList fruitOption;
...
public Produce() {
  protected String Fruits[] = {
     "Apple", "Pear", "Loquat"
  };
  fruitOption = new ArrayList(Fruits.length);
  ...
  for (i = 0; i < Fruits.length; i++) {
     fruitOption.add(new SelectItem(Fruits[i],
                Fruits[i], Fruits[i]));
  }
}
...
public void setFruitOption(Collection aFruit) {
  fruitOption = new ArrayList(aFruit);
}
public Collection getFruitOption() {
  return fruitOption;
}
```

The code first initializes `fruitOption` as an `ArrayList`. The `for` loop creates a set of `SelectItem` objects with values, labels and descriptions for each of the fruit types. Finally, the code includes the obligatory `setFruitOption` and `get-FruitOption` accessor methods.

Writing Properties Bound to Component Instances

A property bound to a component instance returns and accepts the component itself rather than the component's type. For example, the `repeater` page in the `components` demo has a `selectboolean_checkbox` tag:

```
<h:selectboolean_checkbox
  id="checked"
  binding="#{RepeaterBean.checked}"/>
```

Its `binding` attribute refers to the `checked` property in the `RepeaterBean`:

```
private UISelectBoolean checked = null;
public UISelectBoolean getChecked() { return checked; }
public void setChecked(UISelectBoolean checked) {
  this.checked = checked;
}
```

This property is of type `UISelectBoolean`. The properties of other components are written in a similar way.

Writing Backing-Bean Methods

Methods of a backing bean perform application-specific functions for components on the page. These functions include performing validation on the component's value, handling action events, handling value-change events, and performing processing associated with navigation.

By using a backing bean to perform these functions, you don't need to implement a `Validator` to handle the validation or a `Listener` to handle events. Also, by using a backing bean instead of a `Validator` implementation to perform validation, you don't need to create a custom tag for the `Validator`.

This section describes the requirements for writing the backing bean methods.

Writing a Method to Perform Validation

A backing bean method that performs validation must accept a `FacesContext` and a `UIInput` component as parameters, just like the `validate` method of the `Validator` interface does. Only values of `UIInput` components or values of components that extend UIInput can be validated. A component refers to this method with its validate attribute.

Here is the backing bean method of the CheckoutFormBean in the Coffee Break example:

```
public void validateEmail(FacesContext context,
     UIInput toValidate) {
  String message = "";
  String email = (String) toValidate.getValue();
  if (email.indexOf('@') == -1) {
     toValidate.setValid(false);
     message = CoffeeBreakBean.loadErrorMessage(context,
        CoffeeBreakBean.CB_RESOURCE_BUNDLE_NAME,
           "EMailError");
     context.addMessage(toValidate.getClientId(context),
        new FacesMessage(message, ""));
  } else {
        toValidate.setValid(true);
  }
}
```

This method checks if the input to the component referencing this method contains the character, "@". See Implement the Validator Method (page 873) for more information on this method.

Writing a Method to Handle an ActionEvent

A backing bean method that handles an ActionEvent must be a public method that accepts an ActionEvent and returns void. This method is referenced with the component's actionListener attribute. Only components that implement ActionSource can refer to this method.

The following backing bean method from the CarStore bean of the cardemo application processes an ActionEvent:

```
public void chooseLocaleFromLink(ActionEvent event) {
  String current = event.getComponent().getId();
  FacesContext context = FacesContext.getCurrentInstance();
  context.getViewRoot().
     setLocale((Locale) locales.get(current));
  resetMaps();
}
```

See Handling an ActionEvent (page 894) for more information on this method.

Writing a Method to Handle Navigation

A backing bean method that handles navigation processing must be a public method that takes no parameters and returns a `String`, which is the logical outcome string that the navigation system uses to determine what page to display next. This method is referenced with the component's `action` attribute.

The following backing bean method from the `CarStore` bean performs processing in response to the `Buy` button being clicked:

```
public String buyCurrentCar() {
  getCurrentModel().getCurrentPrice();
  return "confirmChoices";
}
```

See Using an Action Method With a Navigation Rule (page 902) for more information on this method.

Writing a Method to Handle Value-Change Events

A backing bean that handles a `ValueChangeEvent` must be a public method that accepts a `ValueChangeEvent` and returns `void`. This method is referenced with the component's `valueChangeEvent` attribute.

Here is an example backing bean method that processes a `ValueChangeEvent`:

```
public void processValueChangeEvent(ValueChangeEvent event)
  throws AbortProcessingException {
  if (null != event.getNewValue()) {
    FacesContext.getCurrentInstance().
      getExternalContext().getSessionMap().
        put("firstName", event.getNewValue());
  }
}
```

This method is a modification of the `processValueChange(ValueChangeEvent)` method of the `FirstNameChanged` listener in the `cardemo` example. See Handling ValueChangeEvents (page 895) for more information on this method.

Configuring Beans

The way to instantiate backing beans used in a JavaServer Faces application and store them in scope is with the Managed Bean Creation facility. This facility is configured in the application configuration resource file (see section Application Configuration, page 801) using `managed-bean` XML elements to define each bean. This file is processed at application startup time, which means that the objects declared in it are available to the entire application before any of the pages are accessed.

With the Managed Bean Creation facility you can:

- Create backing beans in one centralized file that is available to the entire application, rather than conditionally instantiating backing beans throughout the application.
- Make changes to the backing bean without any additional code
- When a managed bean is created, customize the bean's property values directly from within the configuration file.
- Using `value` elements, set the property of one managed bean to be the result of evaluating another value binding expression.
- Create managed beans programmatically as well as from a JSP page. You'd do this by creating a `ValueBinding` for the value reference expression and then calling `getValue` on it.

This section shows you how to initialize backing beans using the Managed Bean Creation Facility. Working With Backing Beans (page 839) explains how to write a bean class. Binding Component Values and Instances to External Data Sources (page 860) explains how to reference a managed bean from the component tags.

Using the managed-bean Element

You create a backing bean using a `managed-bean` element, which represents an instance of a bean class that must exist in the application. At runtime, the JavaServer Faces implementation processes the `managed-bean` element and instantiates the bean as specified by the element configuration if no instance already exists.

Here is an example of instantiating a bean using the Managed Bean Creation Facility:

```
<managed-bean-name> CarBean </managed-bean-name>
  <managed-bean-class>
    cars.CarBean
  </managed-bean-class>
  <managed-bean-scope> application </managed-bean-scope>
  <managed-property>
    <property-name>sportsCar</property-name>
    <value>dukesRoadster</value>
  </managed-property>
  ...
</managed-bean-name>
```

The `managed-bean-name` element defines the key under which the bean will be stored in a scope. For a component to map to this bean, the component tag's `value` must match the `managed-bean-name` up to the first period. For example, consider this `value` expression that maps to the `sportsCar` property of `CarBean`:

```
value="#{CarBean.sportsCar}"
```

The part before the "." matches the `managed-bean-name` of `CarBean`. Using the HTML Tags (page 817) has more examples of using `value` to bind components to bean properties.

The `managed-bean-class` element defines the fully-qualified name of the JavaBeans component class used to instantiate the bean. It is the application developer's responsibility to ensure that the class complies with the configuration of the bean in the application configuration resources file. For example, the property definitions must match those configured for the bean.

The `managed-bean-scope` element defines the scope in which the bean will be stored. The four acceptable scopes are: `none`, `request`, `session` or `application`. If you define the bean with a none scope, the bean is instantiated anew each time it is referenced, and so it does not get saved in any scope. One reason to use a scope of none is when a managed bean references another `managed-bean`. The second bean should be in none scope if it is only supposed to be created when it is referenced. See Initializing Managed Bean Properties (page 857) for an example of initializing a managed-bean property.

The `managed-bean` element can contain zero or more `managed-property` elements, each corresponding to a property defined in the bean class. These elements are used to initialize the values of the bean properties. If you don't want a particular property initialized with a value when the bean is instantiated, do not include a `managed-property` definition for it in your application configuration file.

A `managed-bean` element also contains zero or more `map-entries` elements or `list-entries` elements. The `map-entries` element configures a bean or set of beans that are instances of `Map`. The `list-entries` element configures a bean or set of beans that are instances of `List`.

To map to a property defined by a `managed-property` element, the part of a component tag's `value` expression after the "." must match the `managed-property` element's `property-name` element. In the example above, the `sportsCar` property is initialized with the value `dukesRoadster`. The next section explains in more detail how to use the `managed-property` element.

Initializing Properties using the managed-property Element

A `managed-property` element must contain a `property-name` element, which must match the name of the corresponding property in the bean. A `managed-property` element must also contain one of a set of elements (listed in Table 21–6 on page 853) that defines the value of the property. This value must be of the same type as that defined for the property in the corresponding bean. Which element you use to define the value depends on the type of the property defined in the bean. Table 21–6 on page 853 lists all of the elements used to initialize a value.

Table 21–6 Subelements of managed-property Elements That Define Property Values

Element	ValueThat it Defines
`list-entries`	Defines the values in a list.
`map-entries`	Defines the values of a map.
`null-value`	Explicitly sets the property to null.
`value`	Defines a single value, such as a String or int, or a JSF EL expression

Using the managed-bean Element (page 851) includes an example of initializing `String` properties using the `value` subelement. You also use the `value` subelement to initialize primitive and other reference types. The rest of this section describes how to use the `value` subelement and other subelements to initialize properties of type `java.util.Map`, `array` and `Collection`, and initialization parameters.

Referencing an Initialization Parameter

Another powerful feature of the Managed Bean Creation Facility is the ability to reference implicit objects from a managed bean property.

Suppose that you have a page that accepts data from a customer, including the customer's address. Suppose also that most of your customers live in a particular zip code. You can make the zip code component render with this zip code by saving it in an implicit object and referencing it when the page is rendered.

You can save the zip code as an initial default value in the context `initParam` implicit object by setting the `context-param` element in your `web.xml` file:

```
<context-param>
  <param-name>defaultZipCode</param-name>
  <param-value>94018</param-name>
</context-param>
```

Next, you write a `managed-bean` declaration with a property that references the parameter:

```
<managed-bean>
  <managed-bean-name>customer</managed-bean-name>
     <managed-bean-class>CustomerBean</managed-bean-class>
     <managed-bean-scope>request</managed-bean-scope>
     <managed-property>
        <property-name>zipCode</property-name>
           <value-ref>initParam.defaultZipCode</value-ref>
        </managed-property>
        ...
</managed-bean>
```

To access the zip code at the time the page is rendered, refer to the property from the `zip` component tag's `value` attribute:

```
<h:input_text id=zip value="#{customer.zipCode}"
```

Retrieving values from other implicit objects are done in a similar way. See Table 21–9 for a list of implicit objects.

Initializing Map Properties

The `map-entries` element is used to initialize the values of a bean property with a type of `java.util.Map` if the `map-entries` element is used within a `managed-property` element. Here is the definition of `map-entries` from the `web-facesconfig_1_0.dtd` (located in the <*JSF_HOME*>`/lib` directory) that defines the application configuration file:

```
<!ELEMENT map-entries (key-class?, value-class?, map-entry*) >
```

As this definition shows, a `map-entries` element contains an optional `key-class` element, an optional `value-class` element and zero or more `map-entry` elements.

Here is the definition of `map-entry` from the DTD:

```
<!ELEMENT map-entry (key, (null-value|value )) >
```

According to this definition, each of the `map-entry` elements must contain a `key` element and either a `null-value` or `value` element. Here is an example that uses the `map-entries` element:

```
<managed-bean>
  ...
  <managed-property>
    <property-name>cars</property-name>
    <map-entries>
      <map-entry>
        <key>Jalopy</key>
        <value>50000.00</value>
      </map-entry>
      <map-entry>
        <key>Roadster</key>
        <value>
          sportsCars.roadster
        </value>
      </map-entry>
    </map-entries>
  </managed-property>
</managed-bean>
```

The map that is created from this `map-entries` tag contains two entries. By default, the keys and values are all converted to `java.lang.String`. If you want to specify a different type for the keys in the map, embed the `key-class` element just inside the `map-entries` element:

```
<map-entries>
  <key-class>java.math.BigDecimal</key-class>
  ...
</map-entries>
```

This declaration will convert all of the keys into `java.math.BigDecimal`. Of course, you need to make sure that the keys can be converted to the type that you specify. The key from the example in this section cannot be converted to a `java.math.BigDecimal` because it is a `String`.

If you also want to specify a different type for all of the values in the map, include the `value-class` element after the `key-class` element:

```
<map-entries>
  <key-class>int</key-class>
  <value-class>java.math.BigDecimal</value-class>
  ...
</map-entries>
```

Note that this tag only sets the type of all the `value` subelements.

The first `map-entry` in the example above includes a `value` subelement. The `value` subelement defines a single value, which will be converted to the type specified in the bean.

The second `map-entry` defines a `value` element, which references a property on another bean. Referencing another bean from within a bean property is useful for building a system out of fine-grained objects. For example, a request-scoped form-handling object might have a pointer to an application-scoped database mapping object. Together the two can perform a form handling task. Note that including a reference to another bean will initialize the bean if it does not exist already.

Instead of using a `map-entries` element, it is also possible to assign the entire map with a `value` element that specifies a map-typed expression.

Initializing Array and Collection Properties

The `values` element is used to initialize the values of an `array` or `Collection` property. Each individual value of the array or `Collection` is initialized using a `value` or `null-value` element. Here is an example:

```
<managed-bean>
  ...
  <managed-property>
    <property-name>cars</property-name>
    <values>
      <value-type>java.lang.Integer</value-type>
      <value>Jalopy</value>
      <value>#{myCarsBean.luxuryCar}</value>
      <null-value/>
    </values>
  </managed-property>
</managed-bean>
```

This example initializes an `array` or a `Collection`. The type of the corresponding property in the bean determines which data structure is created. The `values` element defines the list of values in the `array` or `Collection`. The `value` element specifies a single value in the `array` or `Collection`. The `value` element references a property in another bean. The `null-value` element will cause the property's set method to be called with an argument of `null`. A `null` property cannot be specified for a property whose data type is a Java primitive, such as `int`, or `boolean`.

Initializing Managed Bean Properties

Sometimes you might want to create a bean that also references other managed beans so that you can construct a graph or a tree of beans. For example, suppose that you want to create a bean representing a customer's information, including the mailing address and street address, each of which are also beans. The following `managed-bean` declarations create a `CustomerBean` instance that has two `AddressBean` properties, one representing the mailing address and the other representing the street address. This declaration results in a tree of beans with `CustomerBean` as its root and the two `AddressBean` objects as children.

```
<managed-bean>
  <managed-bean-name>customer</managed-bean-name>
  <managed-bean-class>
    com.mycompany.mybeans.CustomerBean
  </managed-bean-class>
  <managed-bean-scope> request </managed-bean-scope>
```

```
        <managed-property>
            <property-name>mailingAddress</property-name>
            <value>addressBean</value>
        </managed-property>
        <managed-property>
            <property-name>streetAddress</property-name>
            <value>addressBean</value>
        </managed-property>
        <managed-property>
            <property-name>customerType</property-name>
            <value>New</value>
        </managed-property>
    </managed-bean>
    <managed-bean>
        <managed-bean-name>addressBean</managed-bean-name>
        <managed-bean-class>
            com.mycompany.mybeans.AddressBean
        </managed-bean-class>
        <managed-bean-scope> none </managed-bean-scope>
        <managed-property>
            <property-name>street</property-name>
            <null-value/>
        <managed-property>
        ...
    </managed-bean>
```

The first `CustomerBean` declaration (with the `managed-bean-name` of customer) creates a `CustomerBean` in request scope. This bean has two properties, called `mailingAddress` and `streetAddress`. These properties use the `value` element to reference a bean, named `addressBean`.

The second managed bean declaration defines an `AddressBean`, but does not create it because its `managed-bean-scope` element defines a scope of `none`. Recall that a scope of `none` means that the bean is only created when something else references it. Since both the `mailingAddress` and `streetAddress` properties both reference `addressBean` using the `value` element, two instances of `AddressBean` are created when `CustomerBean` is created.

When you create an object that points to other objects, do not try to point to an object with a shorter life span because it might be impossible to recover that scope's resources when it goes away. A session-scoped object, for example, cannot point to a request-scoped object. And objects with `none` scope have no effective life span managed by the framework, so they can only point to other `none` scoped objects. Table 21–7 outlines all of the allowed connections:

Table 21–7 Allowable Connections Between Scoped Objects

An object of this scope	May point to a object of this scope
none	none
application	none, application
session	none, application, session
request	none, application, session, request

Cycles are not permitted in forming these connections to avoid issues involving order of initialization.

Initializing Maps and Lists

In addition to configuring `Map` and `List` properties, you can also configure a `Map` and a `List` directly so that you can reference them from a tag rather than referencing a property that wraps a `Map` or `List`.

To illustrate, lets re-configure the example from Initializing Map Properties (page 855):

```
<managed-bean>
  ...
  <map-entries>
    <map-entry>
      <key>Jalopy</key>
      <value>50000.00</value>
    </map-entry>
    <map-entry>
      <key>Roadster</key>
      <value>
        #{sportsCars.roadster}
      </value>
    </map-entry>
  </map-entries>
</managed-bean>
```

This configuration is the same as that in Initializing Map Properties (page 855), except there is no `managed-property` element used to wrap `map-entries`.

Binding Component Values and Instances to External Data Sources

As explained in Backing Bean Management (page 799), a component tag can bind a component value or a component instance to a bean property.

A component tag's `value` attribute uses the JSF EL syntax to bind a component's value to a bean property. A component tag's `binding` attribute uses the JSF EL syntax to bind a component instance to a bean property, as shown in these example tags:

```
<h:input_text id="carOne"
  value="#{CarBean.carName}" />
<h:input_text id="carTwo"
  binding="#{CarBean.carInstance}" />
```

The `value` attribute of the `carOne` tag uses a JSF EL expression to bind the value of the `carOne` component to the `carName` property of `CarBean`.

The `binding` attribute of the `carTwo` tag uses a JSF EL expression to bind the `carTwo` component instance to the `carInstance` property of `CarBean`.

The `value` attribute can also be used to specify a literal value or to map the component's data to any primitive (such as `int`), structure (such as an array), or collection (such as a list), independent of a JavaBeans component. Table 21–8 lists some example JSF EL expressions that you can use with the value attribute.

Table 21–8 Example JSF EL Expressions for the value Attribute

Value	Expression
A property initialized from a context init parameter	`initParam.quantity`
A bean property	`CarBean.engineOption`
Value in an array	`engines[3]`
Value in a collection	`CarPriceMap["jalopy"]`

Table 21–8 Example JSF EL Expressions for the value Attribute (Continued)

Value	Expression
Property of an object in an array of objects	`cars[3].carPrice`

In addition to referencing bean properties using `value` and `binding` attributes, you can also reference bean properties as well as methods and resource bundles from a custom component attribute by creating a `ValueBinding` for it. Please see Creating the Component Tag Handler (page 916) and Creating Custom Component Classes (page 922) for more information on JSF EL-enabling your component's attributes.

The next two sections explain in more detail how to use the `value` attribute to bind a component's value to a bean property or other external data sources and how to use the binding attribute to bind a component instance to a bean property

Binding a Component Value to a Property

To bind a component's value to a bean property, you must first specify the name of the bean or property as the value of the `value` attribute. As explained in Configuring Beans (page 851), the component tag's `value` expression must match the corresponding `message-bean-name` element up to the first "." in the expression. Likewise, the part of the `value` expression after the "." must match the name specified in the corresponding `property-name` element in the application configuration file. For example, consider this bean configuration:

```
<managed-bean>
  <managed-bean-name>CarBean</managed-bean-name>
  <managed-property>
    <property-name>carName</property-name>
    <value>Jalopy</value>
  </managed-property>
  ...
</managed-bean>
```

This example configures a bean called `CarBean`, which has a property called `carName` of type String.

To bind a component to this bean property, you refer to the property using a JSF EL expression from the `value` attribute of the component's tag:

```
<h:output_text value="#{CarBean.carName}" />
```

See Configuring Beans (page 851) for information on how to configure beans in the application configuration file.

Writing Component Properties (page 840) explains in more detail how to write the backing-bean properties for each of the component types.

Binding a Component Value to an Implicit Object

One external data source that a `value` attribute can refer to is a value mapped in an implicit object.

Suppose that you have a set of pages that all display a version number in a `UIOutput` component. You can save this number in an implicit object. This way, all of the pages can reference it, rather than each page needing to include it. To save `versionNo` as an initial default value in the context `initParam` implicit object set the `context-param` element in your `web.xml` file:

```
<context-param>
  <param-name>versionNo</param-name>
  <param-value>1.05</param-name>
</context-param>
```

To access the version number at the time the page is rendered, refer to the parameter from the `version` component tag's `value` attribute:

```
<h:output_text id=version value="#{initParam.versionNo}"
```

Storing values to and retrieving values from other implicit objects are done in a similar way. Table 21–9 lists the implicit objects that a value attribute can refer to.

Table 21–9 Implicit Objects

Implicit object	**What it is**
applicationScope	A `Map` of the application scope attribute values, keyed by attribute name.
cookie	A `Map` of the cookie values for the current request, keyed by cookie name.
facesContext	The `FacesContext` instance for the current request.
header	A `Map` of HTTP header values for the current request, keyed by header name.
headerValues	A `Map` of `String` arrays containing all of the header values for HTTP headers in the current request, keyed by header name.
initParam	A `Map` of the context initialization parameters for this web application.
param	A `Map` of the request parameters for this request, keyed by parameter name.
paramValues	A `Map` of `String` arrays containing all of the parameter values for request parameters in the current request, keyed by parameter name.
requestScope	A `Map` of the request attributes for this request, keyed by attribute name.
sessionScope	A `Map` of the session attributes for this request, keyed by attribute name.
tree	The root `UIComponent` in the current component tree stored in the `FacesRequest` for this request.

Binding a Component Instance to a Bean Property

A component instance can be bound to a bean property using a JSF EL expression with the `binding` attribute of the component's tag. Here is an example:

```
<h:input_text id="carOne" binding="#{CarBean.carName}"/>
```

This tag binds the `carOne` component to the `carName` property of `CarBean`. This bean property must return and accept a `UIInput` component.

By binding a component to a bean property, other methods of the bean can access the component instance more readily rather than having to acquire an `Application` instance and use it to get a `ValueBinding` for the component.

Here is a tag from the `carDetail` page of the cardemo application that uses a value reference expression with its `binding` attribute to bind a bean property to a component instance:

```
<h:selectboolean_checkbox  title="#{bundle.towPkgLabel}" |
  alt="#{bundle.towPkgLabel}"
  binding="#{carstore.currentModel.components.towPackage}" >
</h:selectboolean_checkbox>
```

The `CarStore` bean is the backing bean. The `currentModel` is the `CarBean` instance, which represents the chosen car. And `towPackage` corresponds to the component instance, which in this case is stored in a `Map`. The data access mechanism that `CarBean` uses is explained more in Writing Component Properties (page 840) as well as in the `CarBean` class comments.

For simplicity's sake, assume that the tag corresponding to the `towPackage` component referenced a simple bean property that returned the component instance:

```
<h:selectboolean_checkbox  title="#{bundle.towPkgLabel}" |
  alt="#{bundle.towPkgLabel}"
  binding="#{MyCarBean.SBComponent}" >
</h:selectboolean_checkbox>
```

Here is a bean property that can be bound to the `towPackage` component:

```
...
UISelectBoolean mySBComponent = null;
...
UISelectBoolean getMySBComponent{
  return mySBComponent;
```

```
}
void setMySBComponent(UISelectBoolean sbComponent){
  mySBComponent = sbComponent;
}
```

Your bean can also do the work of instantiating the components rather than letting the tag do it. In this case, when the tag is processed, the JSF EL expression is evaluated. If the bean referenced from the component attribute instantiates the component then the JavaServer Faces implementation will use that component instance and not create one from the tag.

Instantiating components in a bean is not covered in this tutorial, but will be covered in the next version. Please refer to the `CarBean` class of the cardemo application for more information on instantiating components in the bean.

Referencing a Backing Bean Method

A component tag has a set of attributes for referencing backing-bean methods that can perform certain functions for the component associated with the tag. These attributes are summarized in Table 21–10.

Table 21–10 Component Tag Attributes

attribute	function
`action`	Refers to a backing bean method that performs navigation processing for the component and returns a logical outcome `String`
`actionListener`	Refers to a backing bean method that handles `ActionEvents`
`validator`	Refers to a backing bean method that performs validation on the component's value
`valueChangeListener`	Refers to a backing bean method that handles `ValueChangeEvents`

Only components that implement `ActionSource` can use the `action` and `actionListener` attributes. Only `UIInput` components or components that extend `UIInput` can use the `validate` or `valueChangeListener` attributes.

The component tag refers to a backing bean method using a JSF EL expression as the value of one of these attributes. Here is an example tag that uses two JSF EL expressions:

```
<h:input text ...
  validator="#{CarBean.validateInput"
  valueChangeListener="#{CarBean.processValueChange}" />
```

The `validator` attribute refers to the `validateInput` method of `CarBean`. This method will validate this component's data.

The `valueChangeListener` attribute refers to the `processValueChange` method of `CarBean`. This method will process the event fired by the component corresponding to this tag.

The other attributes are used in a similar way. See Writing Backing-Bean Methods (page 848) for information on how to write the methods to which these attributes refer.

Performing Validation

JavaServer Faces technology provides a set of standard classes and associated tags that page authors and application developers can use to validate a component's data. Table 21–11 lists all of the standard validator classes and the tags that allow you to use the validators from the page.

Table 21–11 The Validator Classes

Validator Class	Tag	Function
`DoubleRangeValidator`	`validate_doublerange`	Checks if the local value of a component is within a certain range. The value must be floating-point or convertible to floating-point.
`LengthValidator`	`validate_length`	Checks if the length of a component's local value is within a certain range. The value must be a `java.lang.String`.

Table 21–11 The Validator Classes (Continued)

Validator Class	Tag	Function
LongRangeValidator	validate_longrange	Checks if the local value of a component is within a certain range. The value must be any numeric type or String that can be converted to a long.

All of these validator classes implement the `Validator` interface. Component writers and application developers can also implement this interface to define their own set of constraints for a component's value.

This section shows you how to display error messages resulting from validation failures, use the standard `Validator` implementations, and write your own custom validator by either implementing the `Validator` interface or implementing a backing-bean method.

Displaying Validation Error Messages

A page author can output error messages resulting from both standard and custom validation failures using the `messages` tag. Here is an example of a `messages` tag from the `greeting` page of the `guessNumber` application:

```
<h:messages style="color: red;
     font-family: 'New Century Schoolbook', serif;
     font-style: oblique;
     text-decoration: overline" id="errors1" for="userNo"/>
```

The `messages` tag causes validation error messages to be displayed wherever the tag is located on the page. The `for` attribute of the tag must match the id of the component whose data needs to be validated. This means that you must provide an ID for the component by specifying a value for the component tag's `id` attribute. If the `for` attribute is not specified, the errors resulting from all failed validations on the page will display wherever the tag is located on the page.

The `messages` tag also has attributes for specifying the appearance of different parts of the message text using Cascading Style Sheet (CSS) language. The `style` attribute specifies a style for the entire message. Please refer to the HTML component tag library located in the *<JSF_HOME>*/lib directory to see what the other attributes of the messages tag are.

The next two sections show examples of using the messages tag with the validation tags.

Requiring a Value

The zip input_text tag on the customerInfo page has a required attribute, which is set to true. Because of this, the JavaServer Faces implementation checks if the value of the component is null or is an empty String.

If your component must have a non-null value or a String value at least one character in length, you should add a required attribute to your component tag and set it to true.

If a component sets required to true, the existence of a value is checked first. If the value is null or a zero-length String, none of the other validators registered on the component are checked. If your component does not set required to true, any other validators on the component will be called, but they will have to handle the case of a null or zero-length String themselves.

Here is the zip input_text tag:

```
<h:input_text id="zip"
  value="#{CustomerBean.zip}" size="10" required="true">
  <cd:format_validator
    formatPatterns="99999|99999-9999|### ###"/>
</h:input_text>
<h:messages  for="zip" />
```

The zip component tag contains a custom validator tag. This custom validator is discussed in the section Creating a Custom Validator (page 870).

Because of the messages tag, an error will display on the page if the value is null or an empty String.

Using the Standard Validators

When using the standard Validator implementations, you don't need to write any code to perform validation. You simply nest the standard validator tag of your choice inside a tag that represents a component of type UIInput (or a subclass of UIInput) and provide the necessary constraints, if the tag requires it.

Validation can only be performed on `UIInput` components or components whose classes extend `UIInput` since these components accept values that can be validated.

This section shows you how to use the `DoubleRangeValidator` and the `LengthValidator`.

Using the DoubleRangeValidator

None of the examples included with JavaServer Faces use the `DoubleRangeValidator`, but you could easily add a component to the cardemo and register a `DoubleRangeValidator` on it. For example, you could create a `bidPrice` component that accepts a bid for a car. You could also register a `DoubleRangeValidator` on this component. This validator checks if the user enters a number that can be converted to a `double` and is at least 5,000.00. Here is the example `bidPrice` `input_text` tag:

```
<h:input_text id="bidPrice" maxlength="7"
  value="#{CustomerBean.bidPrice}" >
  <f:validate_doublerange minimum="5000.00"/>
</h:input_text>
<h:messages for="bidPrice"/>
```

The `validate_doublerange` tag also has a `maximum` attribute, with which you can set a maximum value. The `bidPrice` tag also uses the `maxlength` attribute to restrict the input to seven characters.

Using the LengthValidator

The `symbol` component tag on the `repeater` page (part of the components demo) has a `LengthValidator` registered on it:

```
<h:input_text id="symbol"
  required="true"
  size="6"
  value="#{customer.symbol}">
  <f:validate_length maximum="6" minimum="2"/>
</h:input_text>
```

The symbol field takes a stock symbol as input. The `LengthValidator` ensures that the symbol has a minimum of two characters and a maximum of six characters.

Creating a Custom Validator

If the standard validators don't perform the validation checking you need, you can easily create a custom validator for this purpose. Usually, you will want to display an error message when data fails validation. You need to store these error messages in a `ResourceBundle`. See Lesson: Isolating Locale-Specific Data in the Java Tutorial for more information on creating a `ResourceBundle`. After you create the `ResourceBundle`, these are the steps you take to create and use a custom validator:

1. Either implement a method or a `Validator` interface to perform the validation.
2. Register the error messages with the application or queue them onto the FacesContext.
3. Register the `Validator` implementation if you created one in step 1.
4. Create a custom tag or use the validator tag if you implemented a `Validator` interface.
5. Reference the validator from the component tag.

This section uses two different custom validator examples to illustrate the two different ways—mentioned in step 1 above—to implement a custom validator. One of the validators is used in the `cardemo` application. The other is used in the `CoffeeBreak` application.

The `cardemo` application uses a general-purpose custom validator that validates input data against a format pattern that is specified in the custom validator tag. This validator is used with the Credit Card Number field and the Zip code field. Here is the custom validator tag used with the Zip code field:

```
<cd:format_validator
  formatPatterns="99999|99999-9999|### ###" />
```

According to this validator, the data entered in the Zip code field must be either:

- A 5-digit number
- A 9-digit number, with a hyphen between the 5th and 6th digits
- A 6-character string, consisting of numbers or letters, with a space between the 3rd and 4th character

The CoffeeBreak application has a validator that makes sure the email address that a user enters contains an "@" symbol.

The rest of this section describe how these validators are implemented, how they work, and how to use them in a page.

Implement the Validator Interface

You can create a custom validator by implementing a backing bean method that performs the validation, or you can implement the `Validator` interface. This `Validator` implementation must contain a constructor, a set of accessor methods for any attributes on the tag, and a `validate` method, which overrides the `validate` method of the `Validator` interface.

The `FormatValidator` class of the `cardemo` application implements `Validator` and validates the data on the Credit Card Number field and the Zip code field. This class defines accessor methods for setting the attribute `formatPatterns`, which specifies the acceptable format patterns for input into the fields.

In addition to the constructor and the accessor methods, the class overrides `Validator.validate`, which validates the input and also accesses the custom error messages to be displayed when the `String` is invalid.

All custom `Validator` implementations must override the `validate` method, which takes the `FacesContext` and the component whose data needs to be validated. A validator can only validate data on a `UIInput` component or a component that extends `UIInput`. This method performs the actual validation of the data.

Here is the `validate` method from `FormatValidator`:

```
public void validate(FacesContext context, UIInput component) {
  boolean valid = false;
  String value = null;
  if ((context == null) || (component == null)) {
    throw new NullPointerException();
  }
  if (!(component instanceof UIOutput)) {
    return;
  }
  if ( formatPatternsList == null ) {
    component.setValid(true);
    return;
  }
  Object input = ((UIOutput)component).getValue();
  if(input != null) {
    value = input.toString();
    //validate the value against the list of valid patterns.
```

```
            Iterator patternIt = formatPatternsList.iterator();
            while (patternIt.hasNext()) {
               valid = isFormatValid(
                  ((String)patternIt.next()), value);
               if (valid) {
                  break;
               }
            }
            if ( valid ) {
               component.setValid(true);
            }  else {
               component.setValid(false);
               FacesMessage errMsg =
                  MessageFactory.getMessage(context,
                     FORMAT_INVALID_MESSAGE_ID,
                     (new Object[] {formatPatterns}));
               context.addMessage(component.getClientId(context)),
                  errMsg);
            }
         }
      }
```

This method gets the local value of the component and converts it to a `String`. It then iterates over the `formatPatternsList` list, which is the list of acceptable patterns as specified in the `formatPatterns` attribute of the `format_validator` tag.

While iterating over the list, this method checks the pattern of the local value against the patterns in the list. If the pattern of the local value does not match any pattern in the list, this method: marks the component's local value invalid by calling `component.setValid(false)`, generates an error message, and queues the error message to the `FacesContext` so that the message is displayed on the page during the Render Response phase.

The error messages are retrieved from the `Application` instance by `MessageFactory`. An application that creates its own custom messages must provide a class, like `MessageFactory`, that retrieves the messages from the `Application` instance. When creating your own application, you can simply copy the `MessageFactory` class from the `cardemo` application to your application.

The `getMessage` method of `MessageFactory` takes a `FacesContext`, a static `String` that represents the key into the `Properties` file, and the format pattern as an Object. The key corresponds to the static message ID in the FormatValidator class:

```
public static final String FORMAT_INVALID_MESSAGE_ID =
  "cardemo.Format_Invalid";
}
```

When the error message is displayed, the format pattern will be substituted for the {0} in the error message, which, in English, is:

```
Input must match one of the following patterns {0}
```

Implement the Validator Method

As explained in Writing Backing-Bean Methods (page 848), a backing bean can contain a method that performs validation on the data of a component to which it is bound. Rather than needing to implement a Validator interface, you can simply write the backing bean method to perform the validation.

The following method is the validateEmail method from the CheckoutFormBean of the Coffee Break application. It checks that the input entered into a UIInput component contains the character "@".

```
public void validateEmail(FacesContext context,
    UIInput toValidate) {
  String message = "";
  String email = (String) toValidate.getValue();
  if (email.indexOf('@') == -1) {
    toValidate.setValid(false);
    message = CoffeeBreakBean.loadErrorMessage(context,
      CoffeeBreakBean.CB_RESOURCE_BUNDLE_NAME,
        "EMailError");
    context.addMessage(toValidate.getClientId(context),
      new FacesMessage(message, ""));
  } else {
      toValidate.setValid(true);
  }
}
```

A method that handles validation must accept a FacesContext and a UIInput as parameters. The UIInput instance is the component whose value requires validation. The validateEmail method from the Coffee Break application first gets the local value of the component. It then checks if the @ character is contained in the value. If it isn't, the method sets the component's valid property to false.

The method then loads the error message and queues it onto the FacesContext, associating the message with the component ID.

Register the Error Messages

If you create custom error messages, you need to make them available at application startup time. You do this by either queuing the message on the FacesContext programmatically as described in the next section, or you register the messages with your application using the application configuration file.

Here is the part of the file that registers the error messages for the cardemo application:

```
<application>
  <message-bundle>carstore.bundles.Messages</message-bundle>
  <locale-config>
    <default-locale>en</default-locale>
    <supported-locale>de</supported-locale>
    <supported-locale>fr</supported-locale>
    <supported-locale>es</supported-locale>
  </locale-config>
</application>
```

This set of elements will cause your Application instance to be populated with the messages contained in the specified ResourceBundle.

The message-bundle element represents a set of localized messages. It must contain the fully-qualified path to the ResourceBundle containing the localized messages, in this case, carstore.bundles.Messages.

The locale-config element lists the default locale and the other supported locales. The default locale is specified with the default-locale element. Each supported locale is specified with a separate supported-locale element.

The supported-locale and default-locale tags accept the lower-case, two-character codes as defined by ISO-639. Make sure that your ResourceBundle actually contains the messages for the locales that you specify with these tags.

Queuing an Error Message

Instead of registering messages in the configuration file, you can queue the message on the `FacesContext` programmatically. The `validateEmail` method does this:

```
...
String message = "";
...
message = CoffeeBreakBean.loadErrorMessage(context,
  CoffeeBreakBean.CB_RESOURCE_BUNDLE_NAME,
    "EMailError");
context.addMessage(toValidate.getClientId(context),
  new FacesMessage(message, ""));
...
```

These lines also call the `loadErrorMessage` to get the message from the `ResourceBundle`. Here is the `loadErrorMessage` method from `CoffeeBreak-Bean`:

```
public static String loadErrorMessage(FacesContext context,
  String basename, String key) {
  if ( bundle == null ) {
    try {
      bundle = ResourceBundle.getBundle(basename,
        context.getViewRoot().getLocale());
    } catch (Exception e) {
      return null;
    }
  }
  return bundle.getString(key);
}
```

Register the Custom Validator

Just as the message resources need to be made available at application startup time, so does the custom validator. If you implemented the `Validator` interface, you need to register this custom validator in the application configuration file with the standard `validator` XML tag from the `jsf-core` tag library:

```
<validator>
  <description>
    Registers the concrete Validator implementation,
    cardemo.FormatValidator with the validator
    identifier, FormatValidator
```

```
    </description>
    <validator-id>FormatValidator</validator-id>
    <validator-class>cardemo.FormatValidator</validator-class>
    <attribute>
      <description>
        List of format patterns separated by '|'.
        The validator compares these patterns against
        the data entered in a component that has this
        validator registered on it.
      </description>
      <attribute-name>formatPatterns</attribute-name>
      <attribute-class>java.lang.String</attribute-class>
    </attribute>
  </validator>
```

The `validator-id` and `validator-class` are required subelements. The `validator-id` represents the identifier under which the `Validator` class should be registered. This ID is used by the tag class corresponding to the custom `validator` tag.

The `validator-class` element represents the fully-qualified class name of the `Validator` class.

The `attribute` element identifies an attribute associated with the `Validator`. It has required `attribute-name` and `attribute-class` subelements. The `attribute-name` element refers to the name of the attribute as it appears in the `validator` tag. The `attribute-class` element identifies the Java type of the value associated with the attribute.

Create a Custom Tag or Use the validator Tag

If you implemented a Validator interface rather than implementing a backing bean that performs the validation, you need to either:

- Specify which validator class to use with the `validator` tag. The `Validator` implementation defines its own properties
- Create a custom tag that provides attributes for configuring the properties of the validator from the page

If you want to configure the attributes in the `Validator` implementation rather than from the page, you need to nest a `f:validator` tag inside the tag of the component whose data needs to be validated and set the `validator` tag's `id` attribute to the ID of the validator that is defined in the application configuration file:

```
<h:input_text id="zip" value="#{CustomerBean.zip}"
      size="10" ... >
  <f:validator validatorId="FormatValidator" />
  ...
</h:input_text>
```

If you want to allow the page author to configure properties on the validator, you should create a custom validator tag. If you create a custom tag, you need to:

- Write a tag handler to create and register the `Validator` instance on the component
- Write a TLD to define the tag and its attributes
- Add the custom tag to the page.

Writing the Tag Handler

The tag handler associated with a custom validator tag must extend the `ValidatorTag` class. This class is the base class for all custom tag handlers that create `Validator` instances and register them on a UI component. The `FormatValidatorTag` is the class that registers the `FomatValidator` instance.

The `FormatValidator` tag handler class:

- Sets the ID of the `Validator` by calling `super.setValidatorId("FormatValidator")`.
- Provides a set of accessor methods for each attribute defined on the tag.
- Implements the `createValidator` method of the `ValidatorTag` class. This method creates an instance of the `Validator` and sets the range of values accepted by the validator.

Here is the `createValidator` method from `FormatValidatorTag`:

```
protected Validator createValidator() throws JspException {
  FormatValidator result = null;
  result = (FormatValidator) super.createValidator();
  Assert.assert_it(null != result);
  result.setFormatPatterns(formatPatterns);
  return result;
}
```

This method first calls `super.createValidator` to get a new `Validator` and casts it to `FormatValidator`.

Next, the tag handler sets the `Validator` instance's attribute values to those supplied as tag attributes in the page. The handler gets the attribute values from the page via the accessor methods that correspond to the attributes.

Writing the Tag Library Descriptor

To define a tag, you need to declare it in a tag library descriptor (TLD), which is an XML document that describes a tag library. A TLD contains information about a library and each tag contained in the library. See Tag Library Descriptors (page 729) for more information about TLDs.

The custom validator tag for the Credit Card Number and Zip Code fields is defined in the `cardemo.tld`, located in `<*JWSDP_HOME*>/jsf/samples/cardemo/web/WEB-INF` directory of your download bundle. It contains only one tag definition, for `format_validator`:

```
<tag>
  <name>format_validator</name>
  <description>
    Defines the format-validator tag, with the
    tag-handler class, cardemo.FormatValidatorTag.
    This tag must be nested inside a UI component
    tag. The value of the UI component whose tag
    encloses the format-validator tag is validated
    against the format patterns defined by the
    formatPatterns attribute.
  <tag-class>cardemo.FormatValidatorTag</tag-class>
  <attribute>
    <name>formatPatterns</name>
    <required>true</required>
    <rtexprvalue>false</rtexprvalue>
    <description>
      Defines the format patterns to use to
      validate the data of the UI component,
      whose tag encloses the format_validator tag.
    </description>
  </attribute>
</tag>
```

The `name` element defines the name of the tag as it must be used in the page. The `tag-class` element defines the tag handler class. The attribute elements define each of the tag's attributes.

Adding the Custom Tag to the Page

To use the custom validator in the JSP page, you need to declare the custom tag library that defines the custom tag corresponding to the custom component.

To declare the custom tag library, include a `taglib` directive at the top of each page that will contain the custom validator tags included in the tag library. Here is the `taglib` directive that declares the cardemo tag library:

```
<%@ taglib uri="/WEB-INF/cardemo.tld" prefix="cd" %>
```

The `uri` attribute value uniquely identifies the tag library. The `prefix` attribute value is used to distinguish tags belonging to the tag library. Here is the `format_validator` tag from the `zip` tag on `Customer.jsp`:

```
<cd:format_validator
     formatPatterns="99999|99999-9999|## ###" />
```

To register this validator on the `zip` component (corresponding to the Zip Code field) you need to nest the `format_validator` tag within the `zip` component tag:

```
<h:input_text id="zip" value="#{CustomerBean.zip}" size="10"
  required="true">
  <cd:format_validator
     formatPatterns=”99999|99999-9999|### ###" />
</h:input_text>
<h:messages for="zip" />
```

The `messages` tag following the `zip` `input_text` tag will cause the error messages to display next to the component on the page. The `for` attribute refers to the component whose value is being validated.

A page author can use the same custom validator for any similar component by simply nesting the custom validator tag within the component tag.

Reference the Validator Method from the Component Tag

If your validation code is implemented by a backing bean method, to use the method to validate a component’s data, you need to refer to the method from the component’s tag using the `validate` attribute. Here is the component tag corresponding to the email component from the `checkoutForm` page of the Coffee Break application:

```
<h:input_text id="email" value="#{checkoutFormBean.email}"
  size="25" maxlength="125"
  validator="#{checkoutFormBean.validateEmail}"/>
```

This tag references the `validateEmail` method described in Implement the Validator Method (page 873) using a JSF EL expression.

Performing Data Conversions

A typical Web application must deal with two different viewpoints of the underlying data being manipulated by the user interface:

- The model view, in which data is represented as native Java types, such as `java.util.Date` or `java.util.Number`.
- The presentation view, in which data is represented in a manner that can be read or modified by the user. For example, a `java.util.Date` might be represented as a text string in the format mm/dd/yy or as a set of three text strings.

The JavaServer Faces implementation automatically converts component data between these two views when the bean property associated with the component is of one of the supported types. For example, if a `UISelectBoolean` component is associated with a bean property of type `java.lang.Boolean`, the JavaServer Faces implementation will automatically convert the data from `String` to `Boolean`. In addition, some components must be bound to properties of a particular type. For example, a `UISelectBoolean` component must be bound to a property of type `boolean` or `java.lang.Boolean`.

Sometimes you might want to convert a component's data to a type other than a standard type, or you might want to convert the format of the data. To facilitate this, JavaServer Faces technology allows you to register a `Converter` implementation on `UIOutput` components and components whose classes subclass `UIOutput`.

The `Converter` converts the data between the two views. You can either use the standard converters supplied with the JavaServer Faces implementation or create your own custom `Converter`. This section describes how to use the standard `Converter` implementations and explains an example of a custom `Converter`.

Using the Standard Converters

The JavaServer Faces implementation provides a set of `Converter` implementations that you can use to convert your component data. These standard `Converter` implementations, located in the `javax.faces.convert` package are:

- `BigDecimalConverter`
- `BigIntegerConverter`
- `BooleanConverter`
- `ByteConverter`
- `CharacterConverter`
- `DateTimeConverter`
- `DoubleConverter`
- `FloatConverter`
- `IntegerConverter`
- `LongConverter`
- `NumberConverter`
- `ShortConverter`

Two of these standard converters (the `DateTimeConverter` and the `NumberConverter`) each have their own tags, which allow you to configure the format of the component data by configuring the tag attributes. Using the DateTime Converter (page 882) discusses using the `DateTimeConverter`. Using the Number Converter (page 884) discusses using the `NumberConverter`.

You can use the other `Converter` implementations in one of three ways: You can make sure that the component that uses the `Converter` is bound to a backing bean property of the same type as the `Converter`, or you can refer to the `Converter` by class using the component tag's `converter` attribute, or you can refer to the `Converter` by its `converterId` using the `f:converter` tag. The latter two will convert the component's local value. For example, if you want a component's data to be converted to an `Integer`, you can bind the component to a property similar to this:

```
Integer age = 0;
public Integer getAge(){ return age;}
public void setAge(Integer age) {this.age = age;}
```

Alternatively, if the component is not bound to a bean property, you can use the `converter` attribute on the component tag:

```
<h:input_text value="#{LoginBean.Age}"
  converter="javax.faces.convert.IntegerConverter />
```

The data corresponding to this tag will be converted to a `java.lang.Integer`. Notice that the `Integer` type is already a supported type of the `NumberConverter`. If you don't need to specify any formatting instructions with the `convert_number` tag attributes, and one of the other converters will suffice, you can simply reference that converter with the component tag's `converter` attribute.

Finally, you can nest an `f:converter` tag within the component tag and refer to the Converter's ID with the `f:converter` tag's `converterId` attribute. The ID must match the ID in the application configuration file. Here is an example:

```
<h:input_text value="#{LoginBean.Age}" />
  <f:converter converterId="Integer" />
</h:input_text>
```

Using the DateTime Converter

You can convert a component's data to a `java.util.Date` by nesting the `convert_datetime` tag inside the component tag. The `convert_datetime` tag has several attributes that allow you to specify the format and type of the data. Table 21–12 lists the attributes for `convert_datetime`.

Here is a simple example of using a `convert_datetime` tag:

```
<h:input_text value="#{LoginBean.date}">
  <f:convert_datetime dateStyle="long" timeStyle="medium"
    type="both" />
</h:input_text>
```

One example of a date and time that this tag can display is:

```
Saturday, Feb 22, 2003 6:10:15 PM
```

You can also display the same date and time with this tag:

```
<h:input_text value="#{LoginBean.date}">
  <f:convert_datetime
    pattern="EEEEEEEE, MMM d, yyyy hh:mm:ss a"
</h:input_text>
```

If you wanted to display the example date above in Spanish, you can use the `parseLocale` attribute:

```
<h:input_text value="#{LoginBean.date}">
  <f:convert_datetime dateStyle="full"
    parseLocale="Locale.SPAIN"
    timeStyle="long" type="both" />
</h:input_text>
```

This tag would display:

```
Sabado, Feb 22, 2003 18:10:15
```

Please refer to the Customizing Formats lesson of the Java Tutorial for more information on how to format the output using the pattern attribute of the `convert_datetime` tag.

Table 21–12 convert_datetime Tag Attributes

Attribute	Type	Description
`dateStyle`	`String`	Defines the format, as specified by `java.text.DateFormat` of a date or the date part of a `date` string. Applied only if `type` is `date` or both and `pattern` is not defined. Valid values: `default`,`short`,`medium`, `long`, and `full`. If no value is specified, `default` is used.
`parseLocale`	`String` or `Locale`	`Locale` whose predefined styles for dates and times are used during formatting or parsing. If not specified, the `Locale` returned by `FacesContext.getLocale` will be used.
`pattern`	`String`	Custom formatting pattern that determines how the date/time string should be formatted and parsed. If this attribute is specified, `dateStyle`, `timeStyle`, and `type` attributes are ignored.

Table 21–12 convert_datetime Tag Attributes (Continued)

Attribute	Type	Description
timeStyle	String	Defines the format, as specified by java.text.DateFormat of a time or the time part of a date string. Applied only if type is time or both and pattern is not defined. Valid values: default, short, medium, long, and full. If no value is specified, default is used.
timeZone	String or TimeZone	Time zone in which to interpret any time information in the date string.
type	String	Specifies whether the string value will contain a date, time, or both. Valid values are date, time, or both. If no value is specified, date is used.

Using the Number Converter

You can convert a component's data to a java.lang.Number by nesting the convert_number tag inside the component tag. The convert_number tag has several attributes that allow you to specify the format and type of the data. Table 21–13 lists the attributes for convert_number.

Here is a simple example of using a convert_number tag:

```
<h:input_text id="netIncome" value="#{Form1040EZ.netIncome}" >
  <f:convert_number currencySymbol="$"
     groupingUsed="true" integerOnly="true"
     maxIntegerDigits="6" type="currency"/>
</h:input_text>
```

An example of a number this tag can display is:

```
$95,234
```

This number can also be displayed with this tag:

```
<h:input_text id="netIncome" value="#{Form1040EZ.netIncome}" >
  <f:convert_number pattern="$##,###" />
</h:input_text>
```

Please refer to the Customizing Formats lesson of the Java Tutorial for more information on how to format the output using the `pattern` attribute of the `convert_number` tag.

Table 21–13 convert_number Attributes

Attribute	Type	Description
`currencyCode`	`String`	ISO4217 currency code, only used when formatting currencies.
`currencySymbol`	`String`	Currency symbol, applied only when formatting currencies
`groupingUsed`	`boolean`	Specifies whether formatted output contains grouping separators
`integerOnly`	`boolean`	Specifies whether only the integer part of the value will be parsed
`maxFractionDigits`	`int`	Maximum number of digits formatted in the fractional part of the output
`maxIntegerDigits`	`int`	Maximum number of digits formatted in the integer part of the output
`minFractionDigits`	`int`	Minimum number of digits formatted in the fractional part of the output
`minIntegerDigits`	`int`	Minimum number of digits formatted in the integer part of the output
`parseLocale`	`String` or `Locale`	`Locale` whose number styles are used to format or parse data.
`pattern`	`String`	Custom formatting pattern that determines how the number string is formatted and parsed
`type`	`String`	Specifies whether the string value is parsed and formatted as a `number`, `currency`, or `percentage`. If not specified, `number` is used.

Creating and Using a Custom Converter

If the standard `Converter` implementations don't perform the kind of data conversion you need, you can easily create a custom `Converter` implementation for this purpose. To create and use a custom `Converter`, you need to perform these steps:

1. Implement the `Converter` interface
2. Register the `Converter` with the application
3. Use the `Converter` in the page

The `cardemo` application uses a custom `Converter`, called `CreditCardConverter`, to convert the data entered in the Credit Card Number field. It strips blanks and dashes from the text string and formats the text string so that a blank space separates every four characters. This section explains how this converter works.

Implement the Converter Interface

All custom converters must implement the `Converter` interface. This implementation—at a minimum—must define how to convert data both ways between the two views of the data.

To define how the data is converted from the presentation view to the model view, the `Converter` implementation must implement the `getAsObject(FacesContext, UIComponent, String)` method from the `Converter` interface. Here is the implementation of this method from `CreditCardConverter`:

```
public Object getAsObject(FacesContext context,
     UIComponent component, String newValue)
        throws ConverterException {
   String convertedValue = null;
   if ( newValue == null ) {
      return newValue;
   }
   convertedValue = newValue.trim();
   if ( ((convertedValue.indexOf("-")) != -1) ||
      ((convertedValue.indexOf(" ")) != -1)) {
      char[] input = convertedValue.toCharArray();
      StringBuffer buffer = new StringBuffer(50);
      for ( int i = 0; i < input.length; ++i ) {
         if ( input[i] == '-' || input[i] == ' '  ) {
            continue;
         } else {
```

```
                buffer.append(input[i]);
            }
        }
        convertedValue = buffer.toString();
    }
    return convertedValue;
}
```

During the Apply Request Values phase, when the components' `decode` methods are processed, the JavaServer Faces implementation looks up the component's local value in the request and calls the `getAsObject` method. When calling this method, the JavaServer Faces implementation passes in the current `FacesContext`, the component whose data needs conversion, and the local value as a `String`. The method then writes the local value to a character array, trims the dashes and blanks, adds the rest of the characters to a `String`, and returns the `String`.

To define how the data is converted from the model view to the presentation view, the `Converter` implementation must implement the `getAsString(FacesContext, UIComponent, Object)` method from the `Converter` interface. Here is the implementation of this method from `CreditCardConverter`:

```
public String getAsString(FacesContext context,
    UIComponent component,Object value)
        throws ConverterException {
    String inputVal = null;
    if ( value == null ) {
        return null;
    }
    try {
        inputVal = (String)value;
    } catch (ClassCastException ce) {
        throw new ConverterException(Util.getExceptionMessage(
            Util.CONVERSION_ERROR_MESSAGE_ID));
        }
        char[] input = inputVal.toCharArray();
        StringBuffer buffer = new StringBuffer(50);
        for ( int i = 0; i < input.length; ++i ) {
            if ( (i % 4) == 0 && i != 0) {
                if (input[i] != ' ' || input[i] != '-'){
                    buffer.append(" ");
                } else if (input[i] == '-') {
                        buffer.append(" ");
                }
            }
            buffer.append(input[i]);
```

```
        }
        String convertedValue = buffer.toString();
        return convertedValue;
    }
```

During the Render Response phase, in which the components' encode methods are called, the JavaServer Faces implementation calls the getAsString method in order to generate the appropriate output. When the JavaServer Faces implementation calls this method, it passes in the current FacesContext, the UIComponent whose value needs to be converted, and the bean value to be converted. Since this Converter does a String-to-String conversion, this method can cast the bean value to a String. It then reads the String to a character array and loops through the array, adding a space after every four characters.

Register the Converter

When you create a custom Converter, you need to register it with the application. Here is the converter declaration from the application configuration file:

```
<converter>
  <description>
    Registers the concrete Converter implementation,
    cardemo.CreditCardConverter using the ID,
    creditcard.
  </description>
  <converter-id>creditcard</converter-id>
  <converter-class>
    cardemo.CreditCardConverter
  </converter-class>
</converter>
```

The converter element represents a Converter implementation. The converter element contains required converter-id and converter-class elements.

The converter-id element identifies an ID that is used by the converter attribute of a UI component tag to apply the converter to the component's data.

The converter-class element identifies the Converter implementation.

Use the Converter in the Page

To apply the data conversion performed by your Converter to a particular component's value, you need to set the converter attribute of the component's tag to the Converter implementation's identifier. You provided this identifier when you registered the Converter with the application, as explained in the previous section.

The identifier for the CreditCardConverter is creditcard. The CreditCardConverter is attached to the ccno component, as shown in this tag from the Customer.jsp page:

```
<h:input_text id="ccno" size="16"
  converter="#{creditcard}" >
  ...
</h:input_text>
```

By setting the converter attribute of a component's tag to the identifier of a Converter, you cause that component's local value to be automatically converted according to the rules specified in the Converter.

A page author can use the same custom Converter for any similar component by simply supplying the Converter implementation's identifier to the converter attribute of the component's tag.

Handling Events

As explained in Event and Listener Model (page 797), the JavaServer Faces event and listener model is similar to the JavaBeans event model in that it has strongly typed event classes and listener interfaces. JavaServer Faces technology supports three different kinds of component events: action events, value-change events, and data-model events.

The discussion of data-model events is an advanced topic, which is not covered in this tutorial, but might be discussed in future versions of this tutorial.

Action events occur when the user activates a component that implements ActionSource. These components include buttons and hyperlinks. These events are represented by the javax.faces.event.ActionEvent class. Either an implementation of the javax.faces.event.ActionListener or a method that takes an ActionEvent parameter can handle action events.

Value-change events result in a change to the local value of a component represented by UIInput or one of its subclasses. One example of a value-change event is that generated by entering a value in a text field. These events are represented by the javax.faces.event.ValueChangeEvent class. Either an implementation of the javax.faces.event.ValueChangeListener or a method that takes a ValueChangeEvent parameter can handle value-change events.

Value-change events are processed during the Process Validations phase. Action events are processed during the Apply Request Values phase or the Invoke Application phase.

Both ActionListener and ValueChangeListener extend from the common FacesListener interface, which is a base class for all events, including custom events, but has no methods of its own.

There are two ways to cause your application to react to action events or value-change events emitted by a standard component:

- Implement an event listener to handle the event and nest a listener tag inside the component tag
- Implement a method of a backing bean to handle the event and refer to the method with a JSF EL expression from the appropriate attribute of the component

When emitting events from custom components, you need to manually queue the event on the component. Handling Events for Custom Components (page 935) explains how to do this. The UIInput and UICommand components automatically queue events.

The rest of this section first explains how to implement a ValueChangeListener and ActionListener and how to register the listener on the component by nesting a listener tag inside the component tag. This section then explains how to create a backing bean method to handle the event and how to refer to the method from the component tag.

Implementing an Event Listener

Listeners that handle the action events in an application must implement javax.faces.event.ActionListener. Similarly, listeners that handle the value-change events must implement javax.faces.event.ValueChangeListener.

Note: You need not create an ActionListener to handle an event that results solely in navigating to a page and does not perform any other application-specific processing. See Navigating Between Pages (page 896) for information on how to manage page navigation.

Implementing a Value-Change Listener

A `ValueChangeListener` implementation must include a `processValueChange(ValueChangeEvent)` method.

The `processValueChange(ValueChangeEvent)` method processes the specified `ValueChangeEvent` and is invoked by the JavaServer Faces implementation when the `ValueChangeEvent` occurs. The `ValueChangeEvent` instance stores the old and the new values of the component that fired the event.

The `FirstNameChanged` listener implementation is registered on the `firstName` `UIInput` component on the `customerInfo` page. This listener stores the first name that the user enters into the component's tag into session scope. When the `finish` page is loaded, it displays the first name inside the message, "Thanks {0} for using cardemo. Your car will ship soon." Here is part of the `FirstNameChanged` listener implementation:

```
...
public class FirstNameChanged extends Object implements
  ValueChangeListener {

  public void processValueChange(ValueChangeEvent event)
    throws AbortProcessingException {

    if (null != event.getNewValue()) {
      FacesContext.getCurrentInstance().
        getExternalContext().getSessionMap().
          put("firstName", event.getNewValue());
    }
  }
}
```

When the user enters his first name in the text field, a `ValueChangeEvent` is generated, and the `processValueChange` method of the `FirstNameChanged` listener implementation is invoked. This method first gets the ID of the component that fired the event from the `ValueChangeEvent`. Next, it puts the value, along with an attribute name, into the session map of the `FacesContext`.

Implementing Action Listeners

An `ActionListener` implementation must include a `processAction(ActionEvent)` method.

The processAction(ActionEvent) processes the specified ActionEvent and is invoked by the JavaServer Faces implementation when the ActionEvent occurs.

The cardemo application does not use any ActionListener implementations. Instead, it uses actionListener expressions that point to bean methods that handle events. This section explains how to turn one of these methods into an ActionListener implementation.

The chooseLocale page allows the user to select a locale for the application by clicking on one of a set of hyperlinks. When the user clicks one of the hyperlinks, an ActionEvent is generated, and the chooseLocaleFromLink(ActionEvent) method of the CarStore bean is invoked. Instead of implementing a bean method to handle this event, you can create a listener implementation to handle it. To do this, you need to:

- Move the chooseLocaleFromLink(ActionEvent) method to a class that implements ActionListener
- Rename the method to processAction(ActionEvent)

The listener implementation would look something like this:

```
...
public class LocaleChange extends Object implements
  ActionListener {

  public void processAction(ActionEvent event)
    throws AbortProcessingException {

    String current = event.getComponent().getId();
    FacesContext context =
      FacesContext.getCurrentInstance();
    context.getViewRoot().setLocale((Locale)
      locales.get(current));
    resetMaps();
  }
}
```

Registering Listeners on Components

A page author can register a listener implementation on a component by nesting either a f:valuechange_listener tag or an f:action_listener tag within the component's tag on the page. A page author can also register these listeners on components with the valueChangeListener and actionListener attributes, as described in Referencing a Method That Handles an ActionEvent (page 895) and Referencing a Method That Handles a ValueChangeEvent (page 896).

This section explains how to register the `FirstNameChanged` listener and the `LocaleChange` listener implementations (explained in the previous two sections) on components.

Registering a ValueChangeListener on a Component

A page author can register a `ValueChangeListener` on a `UIInput` component or a component that extends from `UIInput` by nesting a `valuechange_listener` tag within the component's tag on the page. Here is the tag corresponding to the `firstName` component from the `customerInfo` page:

```
<h:input_text  id="firstName" value="#{customer.firstName}"
  required="true">
  <f:valuechange_listener type="carstore.FirstNameChanged" />
</h:input_text>
```

The `type` attribute of the `valuechange_listener` tag specifies the fully-qualified class name of the `ValueChangeListener` implementation.

After this component tag is processed and local values have been validated, the component instance represented by this tag will automatically queue the `ValueChangeEvent` associated with the specified `ValueChangeListener` to the component. This listener processes the event after the phase specified by the `getPhaseID` method of the listener implementation.

Registering an ActionListener on a Component

A page author can register an `ActionListener` on a `UICommand` component by nesting an `action_listener` tag within the component's tag on the page. Here is one of the `command_link` tags on the `chooseLocale` page, changed to reference an `ActionListener` implementation rather than a bean method:

```
<h:command_link id="NAmerica" action="storeFront">
   <f:action_listener type="carstore.LocaleChange" />
</h:command_link>
```

The `type` attribute of the `action_listener` tag specifies the fully-qualified class name of the `ActionListener` implementation.

When the component associated with this tag is activated, the component's `decode` method (or its associated `Renderer`) automatically queues the `ActionEvent` associated with the specified `ActionListener` to the component. This listener processes the event after the phase specified by the `getPhaseID` method of the listener implementation.

Handling Events With A Backing Bean Method

As explained in Writing Backing-Bean Methods (page 848), an application developer can write methods in a backing bean that can perform application-specific processing, such as validation, conversion, and event-handling.

Component tags reference the methods on a backing bean through special attributes using JSF EL expressions, as described in Referencing a Backing Bean Method (page 865). `UICommand` components or components that implement `ActionSource` can reference methods that handle `ActionEvents`. `UIInput` components can reference methods that handle `ValueChangeEvents`.

This section explains how to implement methods that handle events and how to reference the methods from a component tag.

Handling an ActionEvent

A backing bean method that handles action events must be a public method that accepts an `ActionEvent` and has a return type of void. This method performs the same tasks as the `processAction` method of `ActionListener`.

The `chooseLocale` page of the `cardemo` application allows the user to choose a locale by clicking on a link. The clicking of the link generates an `ActionEvent`. Here is the method that handles this `ActionEvent`:

```
public void chooseLocaleFromLink(ActionEvent event) {
  String current = event.getComponent().getId();
  FacesContext context = FacesContext.getCurrentInstance();
  context.getViewRoot().
    setLocale((Locale) locales.get(current));
  resetMaps();
}
```

Referencing a Method That Handles an ActionEvent

To reference the chooseLocaleFromLink method (described in the previous section) from the component tag, you need to use the tag's actionListener attribute, as shown in the NAmerica tag from the chooseLocale page:

```
<h:command_link id="NAmerica" action="storeFront"
  actionListener="#{carstore.chooseLocaleFromLink}">
```

The actionListener attribute can only be used with the tags of components that are UICommand components or that implement ActionSource.

Handling ValueChangeEvents

A backing bean method that handles ValueChangeEvents must be a public method that accepts a ValueChangeEvent and has a return type of void. This method performs the same tasks as the processValueChange method of a ValueChangeListener.

The cardemo application does not have any backing bean methods that handle value-change events. It does have a ValueChangeEvent implementation, explained in the Implementing a Value-Change Listener (page 891) section. This section explains how write a backing bean method to replace the ValueChangeEvent implementation.

As explained in Registering a ValueChangeListener on a Component (page 893), the firstName component of the customerInfo page has a ValueChangeListener registered on it. This ValueChangeListener handles the event of entering a value in the field corresponding to the component. When the user enters a value, a ValueChangeEvent is generated, and the processValueChangeEvent(ValueChangeEvent) method of the FirstNameChanged class is invoked.

Instead of implementing a ValueChangeListener, you can write a bean method to handle this event. To do this, you just need to move the processValueChangeEvent(ActionEvent) method to your backing bean.

The next section explains how to refer to the bean method from the component tag.

Referencing a Method That Handles a ValueChangeEvent

To reference the backing bean method (described in the previous section) from the component tag, you need to use the tag's `valueChangeListener` attribute, as shown in the `firstName` tag from the `customerInfo` page, changed to refer to the backing bean method rather than the listener implementation:

```
<h:input_text  id="firstName" value="#{customer.firstName}"
  required="true"
  valueChangeListener="#{carstore.FirstNameChanged}" />
```

The `valueChangeListener` attribute can only be used with the tags of `UIInput` components and components that extend `UIInput`.

Navigating Between Pages

As explained in Navigation Model (page 798), page navigation rules are defined in a centralized XML file called the application configuration resource file. See Application Configuration (page 801) for more information on this file.

You can include any application-specific processing associated with navigation in a backing bean method, called an *action* method. This method is referenced by the `action` attribute of the component tag corresponding to the component that triggers navigation. A custom component that implements `ActionSource` can also refer to an action method through the `action` attribute of its tag.

The action method returns a logical outcome based on the results of its processing. This outcome describes what happened during the processing. The method that was invoked and the outcome that is returned are two criteria a navigation rule uses for choosing which page to navigate to.

This rest of this section explains:

- What navigation is
- How an application navigates between pages
- How to define navigation rules in the application configuration file
- How to include any processing associated with page navigation in an action method
- How to reference the action method from a component tag

What is Navigation?

Navigation is a set of rules for choosing the next page to be displayed after a button or hyperlink is clicked. The selection of the next page is determined by:

- The page that is currently displayed
- The action method invoked by the `action` property of the component that generated the event.
- An outcome string that was returned by the action method or passed from the component.

A single navigation rule defines how to navigate from one particular page to any number of other pages in an application. The JavaServer Faces implementation chooses the proper navigation rule according to what page is currently displayed.

Once the proper navigation rule is selected, the choice of which page to access next from the current page depends on the action method that was invoked and the outcome that was returned.

The component either specifies an outcome or refers to an action method by using its `action` property. The method performs some processing and returns a particular outcome string.

The outcome can be anything the developer chooses, but Table 21–14 on page 897 lists some outcomes commonly used in Web applications.

Table 21–14 Common outcome strings

Outcome	What it Means
"success"	Everything worked. Go on to the next page.
"failure"	Something is wrong. Go on to an error page.
"logon"	The user needs to log on first. Go on to the logon page.
"no results"	The search did not find anything. Go to the search page again.

Usually, the action method performs some processing on the form data of the current page. For example, the method might check if the username and password entered in the form match the username and password on file. If they match, the method returns the outcome "success". Otherwise, it returns the outcome "failure". As this example demonstrates, both the method used to process the action and the outcome returned are necessary to determine the proper page to access.

Here is a navigation rule that could be used with the example described in the previous paragraph:

```
<navigation-rule>
  <from-view-id>/logon.jsp</from-view-id>
  <navigation-case>
    <from-action>#{LogonForm.logon}</from-action>
    <from-outcome>success</from-outcome>
    <to-view-id>/storefront.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-action>#{LogonForm.logon}</from-action>
    <from-outcome>failure</from-outcome>
    <to-view-id>/logon.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
```

This navigation rule defines the possible ways to navigate from `logon.jsp`. Each `navigation-case` element defines one possible navigation path from `logon.jsp`. The first `navigation-case` says that if `LogonForm.logon` returns an outcome of "success", `storefront.jsp` will be accessed. The second `navigation-case` says that `logon.jsp` will be re-rendered if `LogonForm.logon` returns "failure".

For a complete description of how to define navigation rules, see Configuring Navigation Rules in faces-config.xml (page 899).

The next section describes what happens behind the scenes when navigation occurs.

How Navigation Works

As The Lifecycle of a JavaServer Faces Page (page 783) explains, a JavaServer Faces page is represented by a component tree, called a view, which is comprised of all of the components on a page. To load another page, the JavaServer Faces implementation accesses a component tree identifier and stores the tree in the `FacesContext`. The new navigation model determines how this tree is selected.

Any components implementing `ActionSource` (such as `UICommand`) in the tree are automatically registered with the default `ActionListener`. When one of the components is activated—such as by a button click—an `ActionEvent` is emitted. If the Invoke Application phase is reached, the default `ActionListener` handles this event.

The `ActionListener` retrieves an outcome—such as "success" or "failure"—from the component generating the event. The component either literally specifies an outcome or refers to an action method with its `action` property. The method performs some processing and returns a particular outcome string.

After receiving the outcome string, the `ActionListener` passes it to the default `NavigationHandler`. Based on the outcome, the currently displayed page, and the action method that was invoked, the `NavigationHandler` selects the appropriate component tree by consulting the application configuration file (`faces-config.xml`).

The next section explains how to define navigation rules for your application in the `faces-config.xml` file.

Configuring Navigation Rules in faces-config.xml

An application's navigation configuration consists of a set of navigation rules. Each rule is defined by the `navigation-rule` element in the `faces-config.xml` file. See Setting Up The Application Configuration File (page 811) for information on how to set up the `faces-config.xml` file for use in your application.

Here are two example navigation rules:

```
<navigation-rule>
  <from-view-id>/more.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/buy.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>out of stock</from-outcome>
    <from-action>
      #{CarOptionServer.carBuyAction}
    </from-action>
    <to-view-id>/outofstock.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

```
<navigation-rule>
  <navigation-case>
    <from-outcome>error</from-outcome>
    <to-view-id>/error.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

The first navigation rule in this example says that the application will navigate from `more.jsp` to:

- `buy.jsp` if the item ordered is in stock.
- `outofstock.jsp` if the item is out of stock.

The second navigation rule says that the application will navigate from any page to `error.jsp` if the application encountered an error.

Each `navigation-rule` element corresponds to one component tree identifier, defined by the optional `from-view-id` element. This means that each rule defines all the possible ways to navigate from one particular page in the application. If there is no `from-view-id` element, the navigation rules defined in the `navigation-rule` element apply to all the pages in the application. The `from-view-id` element also allows wildcard matching patterns. For example, this `from-view-id` element says the navigation rule applies to all the pages in the `cars` directory:

```
<from-view-id>/cars/*</from-view-id>
```

As shown in the example navigation rule, a `navigation-rule` element can contain zero or more `navigation-case` elements. The `navigation-case` element defines a set of matching criteria. When these criteria are satisfied, the application will navigate to the page defined by the `to-view-id` element contained in the same `navigation-case` element.

The navigation criteria are defined by optional `from-outcome` and `from-action` elements.

The `from-outcome` element defines a logical outcome, such as "success". The `from-action` element refers to an action method that returns a `String`, which is the logical outcome. The method performs some logic to determine the outcome and returns the outcome.

The `navigation-case` elements are checked against the outcome and the JSF EL expression in this order:

- Cases specifying both a `from-outcome` value and a `from-action` value. Both of these elements can be used if the action method returns different outcomes depending on the result of the processing it performs.
- Cases specifying only a `from-outcome` value. The `from-outcome` element must match either the outcome defined by the `action` attribute of the `UICommand` component or the outcome returned by the method referred to by the `UICommand` component.
- Cases specifying only a `from-action` value. This value must match the `action` expression specified by the component tag.

Once any of these cases are matched, the component tree defined by the `to-view-id` element will be selected for rendering.

Referencing an Action Method (page 901) explains how to write the tag corresponding to the component to return an outcome.

Referencing an Action Method

The `command_button` and `command_link` tags use the action attribute to specify an outcome, which is matched against the `from-outcome` elements in the application configuration file in order to select the next page to be rendered. The value of the action attribute can be either a literal `String` representing the outcome or a JSF EL expression that points at a method that takes no arguments and returns a `String` that is the outcome.

For example, these two `command_button` takes demonstrate two valid uses of the `action` attribute:

```
<h:command_button action="#{carstore.buyCurrentCar}"
  value="#{bundle.buy}" />
<h:command_button action="success" value="#{bundle.buy}" />
```

The `action` attribute of the first `command_button` tag refers to `carstore.buyCurrentCar`, an action method that returns the logical outcome. The `action` attribute of the second `command_button` tag specifies the literal `String` "outcome" directly, resulting in an outcome of success occurring whenever the button corresponding to the tag is clicked.

If the outcome matches an outcome defined by a `from-outcome` element in the application configuration file, the component tree specified in that navigation case is selected for rendering if one of these is true:

- No `from-action` is defined for that navigation case
- There is a `from-action` also defined for that navigation case, and the method reference matches the method reference identified by the command component's `action` attribute.

Suppose that the `command_button` tag used the literal value instead of the JSF EL expression for the `action` attribute:

```
<h:command_button action="confirmChoices"
  value="#{bundle.buy}" />>
```

If this outcome matches an outcome defined by a `from-outcome` element in the application configuration file, the component tree corresponding to this navigation case is selected for rendering, regardless of whether or not the same navigation case also contains a `from-action` element.

The next section explains how to write the action method referred to by the component's `action` attribute.

Using an Action Method With a Navigation Rule

It's common for applications to have a choice of pages to navigate to from a given page. You usually need to provide some application-specific processing that determines which page to access in a certain situation. This processing code goes into an action method on a backing bean. Here is the action method referenced by the Buy button on the `carDetail` page of the `cardemo` application:

```
public String buyCurrentCar() {
  getCurrentModel().getCurrentPrice();
  return "confirmChoices";
}
```

The `CarStore.buyCurrentCar` method gets the current price of the chosen car and then returns the logical outcome, `confirmChoices`.

All methods that are referenced by `action` attributes that are JSF EL expressions are required to be public, to return a `String`, and to take no parameters.

When the `NavigationHandler` receives the "confirmChoices" outcome, it selects the `/confirmChoices.jsp` view.

As shown in the example code in this section, it's sometimes a good idea to include your action method inside the same bean class that defines other properties used by the application. This is because the method will often need to access the bean's data to determine what outcome to return, as it does in the example in this section.

Performing Localization

All data and messages in the `cardemo` application have been completely localized for French, German, Latin-American Spanish, and American English.

The image map on the first page allows you to select your preferred locale. See Chapter 22 for information on how the image map custom component was created.

This section explains how to localize static and dynamic data and messages for JavaServer Faces applications. If you are not familiar with the basics of localizing Web applications, see Internationalizing and Localizing Web Applications (page 941).

Localizing Static Data

Static data can be localized using the `f:loadBundle` tag, defined in `jsf_core.tld`, by following these steps:

1. Create a ResourceBundle
2. Reference the bundle from the page
3. Reference the localized message located within the bundle

Create a Resource Bundle

A `ResourceBundle` contains a set of localized message. Refer to the Internationalization trail of the Java Tutorial to learn how to create a `ResourceBundle`. Once you create the ResourceBundle, put it in the same directory as your classes. Much of the data for `cardemo` is stored in a `ResourceBundle` called `Resources`.

Reference the Bundle from the Page

In order for a JavaServer Faces page to use the localized messages contained in a `ResourceBundle`, it must reference the `ResourceBundle` using an `f:loadBundle` tag.

The `f:loadBundle` tag from `customerInfo.jsp` is:

```
<f:loadBundle
  basename="carstore.bundles.Resources" var="bundle"/>
```

The `basename` attribute value refers to the `ResourceBundle`, located in the `cardemo` package. Make sure the `basename` attribute specifies the fully qualified class name of the file.

The `var` attribute is an alias to the `ResourceBundle`. This alias can be used by other tags in the page in order to access the localized messages.

Reference the Localized Message

To reference a localized message from a ResourceBundle, you use a value reference expression from an attribute of the component tag that will display the localized data. You can reference the message from any component tag attribute that supports the value binding language.

The value reference expression has the notation "var.message", in which var matches the `var` attribute of the `f:loadBundle` tag and message matches the key of the message contained in the `ResourceBundle` referred to by the `var` attribute. Here is an example from `customerInfo`:

```
<h:output_text value="#{bundle.customerTitle}" />
```

Notice that `bundle` matches the `var` attribute from the `f:loadBundle` tag and `customerTitle` matches the key in the `ResourceBundle`.

Another example is the `graphic_image` tag from `chooseLocale`:

```
<h:graphic_image id="mapImage" url="/images/world.jpg"
  alt="#{bundle.chooseLocale}"
  usemap="#worldMap" />
```

The `alt` attribute is value binding-enabled, which means it can use value reference expressions. In this case, the `alt` attribute refers to localized text, which will be included in the alt text of the image rendered by this tag.

See Creating the Component Tag Handler (page 916) and Defining Properties for the Component's Values (page 928) for information on how to enable value binding on your custom component's attributes.

Localizing Dynamic Data

The `cardemo` application has some data that is set dynamically in JavaBeans classes. Because of this, the beans must load the localized data themselves; the data can't be loaded from the page.

One example of dynamically-loaded data includes the data associated with a `UISelectOne` component. The `titleOptions` component on the `customerInfo` page is a `UISelectOne` component that holds data that needs to be localized. When the locale is set to english, this component displays the values Mr., Ms., Mrs.

In the `CustomerBean`, the localized titles are loaded with the `getTitleOptions` method. Here is the `getTitleOptions` method:

```
public Collection getTitleOptions() {
  if (null == titleOptions) {
    titleOptions = new ArrayList();
    ResourceBundle rb =
      ResourceBundle.getBundle("carstore.bundles.Resources",
        (FacesContext.getCurrentInstance().
          getViewRoot().getLocale()));
    String titleStr = (String)rb.getObject("mrLabel");
    titleOptions.add(
      new SelectItem(titleStr, titleStr, titleStr));
    titleStr = (String)rb.getObject("mrsLabel");
    titleOptions.add(
      new SelectItem(titleStr, titleStr, titleStr));
    titleStr = (String)rb.getObject("msLabel");
    titleOptions.add(
      new SelectItem(titleStr, titleStr, titleStr));
  }
  return titleOptions;
}
```

This method gets the current locale from the `UIViewRoot` of the current request and loads the localized data for the titles using the `getBundle` method, passing in the path to the `ResourceBundle` and the current locale. A SelectItem instance is created with a localized string and is loaded into an `ArrayList`. The `titleOp-`

`tions` component tag references the `titleOptions` property, thus displaying the returned list of localized title strings.

Localizing Messages

The JavaServer Faces API provides the `FacesMessage` class, which represents a single localized message. You can either create the message with one of the `FacesMessage` constructors or access the message through a message factory implementation. The message factory is required to access error messages that are registered with the `Application` instance.

All JavaServer Faces applications that use custom error messages must provide a message factory implementation. Instead of writing your own message factory, you can use the `MessageFactory` class that is included with the `cardemo` application.

You will most likely use custom error messages with a custom validator. Performing Validation (page 866) includes an example of registering and using validation error messages.

The steps for using localized messages in your application is:

1. Create a `ResourceBundle` and add the localized messages to the bundle.
2. Register the `ResourceBundle` with the application. This is explained in Register the Error Messages (page 874).
3. Create a message factory implementation. You can simply copy the `MessageFactory` class included with the `cardemo` application to your application.
4. Access a message from your application by calling the `getMessage(FacesContext, String, Object)` of the `MessageFactory` class. The `MessageFactory` uses the `FacesContext` to access the `Application` instance, on which the messages are registered. The `String` argument is the key that corresponds to the message in the `ResourceBundle`. The `Object` typically contains the substitution parameters that are embedded in the message. For example, the custom validator described in Implement the Validator Interface (page 871) will substitute the format pattern for the {0} in this error message:

   ```
   Input must match one of the following patterns {0}
   ```

Implement the Validator Interface (page 871) gives an example of accessing messages.

22

Creating Custom UI Components

JavaServer Faces technology offers a rich set of standard, reusable UI components that enable you to quickly and easily construct UIs for Web applications. But often you need a component with some additional functionality or a completely new component, like a client-side image map. Although JavaServer Faces technology doesn't furnish all components you might need, its component architecture allows you to extend the standard components to enhance their functionality or create your own unique components.

In addition to extending the functionality of standard components, you might also want to change their appearance on the page or render them to a different client. Enabled by the flexible JavaServer Faces architecture, you can separate the definition of the component behavior from its appearance by delegating the rendering of the component to a separate renderer. This way, you can define the behavior of a custom component once, but create multiple renderers, each of which defines a different way to render the component.

As well as providing a means to easily create custom components and renderers, the JavaServer Faces design also makes it easy to reference them from the page through JSP custom tag library technology.

This chapter uses an image map custom component to explain all you need to know to create simple custom components, custom renderers, and associated custom tags, and to take care of all the other details associated with using the components and renderers in an application.

Determining if You Need a Custom Component or Renderer

The JavaServer Faces implementation already supports a rich set of components and associated renderers, which are enough for most simple applications. This section helps you decide if you need a custom component or custom renderer or if you can use a standard component and renderer.

When to Use a Custom Component

A component class defines the state and behavior of a UI component. This behavior includes: converting the value of a component to the appropriate markup, queuing events on components, performing validation, and other functionality.

Situations in which you need to create a custom component include:

- If you need to add new behavior to a standard component, such as generating an additional type of event.
- If you need to aggregate components to create a new component that has its own unique behavior. The new component must be a custom component. One example is a date chooser component consisting of three drop-down lists.
- If you need a component that is supported by an HTML client, but is not currently implemented by JavaServer Faces technology. The current release does not contain standard components for complex HTML components, like frames; however, because of the extensibility of the component architecture, you can use JavaServer Faces technology to create components like these.
- If you need to render to a non-HTML client, which requires extra components not supported by HTML. Eventually, the standard HTML render kit will provide support for all standard HTML components. However, if you are rendering to a different client—such as a phone—you might need to create custom components to represent the controls uniquely supported by the client. For example, the MIDP component architecture includes support for tickers and progress bars, which are not available on an HTML client. In this case, you might also need a custom renderer along with the component; or, you might just need a custom renderer.

You do not need to create a custom component if:

- You need to simply manipulate data on the component or add application-specific functionality to it. In this situation, you should create a backing bean for this purpose and bind it to the standard component rather than create a custom component. See Working With Backing Beans (page 839) for more information on creating a backing bean.
- You need to convert a component's data to a type not supported by its renderer. See Performing Data Conversions (page 880) for more information about converting a component's data.
- You need to perform validation on the component data. Both standard validators and custom validators can be added to a component by using the validator tags from the page. See Performing Validation (page 866) for more information about validating a component's data.
- You need to register event listeners on components. You can either register event listeners on components with the `valuechange_listener` and `action_listener` tags, or you can point at an event processing method on a backing bean using the component's `actionListener` or `valuechangeListener` attribute. See Handling Events (page 889) for more information on using these tags.

When to Use a Custom Renderer

If you are creating a custom component, you need to ensure—among other things—that your component class performs these operations:

- Decoding: converting the incoming request parameters to the local value of the component.
- Encoding: converting the current local value of the component into the corresponding markup that represents it in the response.

The JavaServer Faces specification supports two programming models for handling encoding and decoding:

- Direct implementation: The component class itself implements the decoding and encoding.
- Delegated implementation: The component class delegates the implementation of encoding and decoding to a separate renderer

By delegating the operations to the renderer, you have the option of associating your custom component with different renderers so that you can represent the component in different ways on the page. If you don't plan to render a particular component in different ways, it's simpler to let the component class handle the rendering.

If you aren't sure if you will need the flexibility offered by separate renderers, but want to use the simpler direct-implementation approach, you can actually use both models. Your component class can include some default rendering code, but it can delegate rendering to a renderer if there is one.

Component, Renderer, and Tag Combinations

When you create a custom component, you will usually create a custom renderer to go with it. You will also need a custom tag to associate the component with the renderer and to reference the component from the page.

In rare situations, however, you might use a custom renderer with a standard component rather than a custom component. Or, you might use a custom tag without a renderer or a component. This section gives examples of these situations and provides a summary of what's required for a custom component, renderer, and tag.

One example of using a custom renderer without a custom component is when you want to add some client-side validation on a standard component. You would implement the validation code with a client-side scripting language, such as JavaScript. You render the JavaScript with the custom renderer. In this situation, you will need a custom tag to go with the renderer so that its tag handler can register the renderer on the standard component.

Both custom components and custom renderers need custom tags associated with them. However, you can have a custom tag without a custom renderer or custom component. One example is when you need to create a custom validator that requires extra attributes on the validator tag. In this case, the custom tag corresponds to a custom validator, not to a custom component or custom renderer. In any case, you still need to associate the custom tag with a server-side object.

Table 22–1 summarizes what you must or can associate with a custom component, custom renderer, or custom tag.

Table 22–1 Requirements for Custom Components, Custom Renderers, and Custom Tags

	Must have	**Can have**
custom component	custom tag	custom renderer
custom renderer	custom tag	custom component or standard component
custom JavaServer Faces tag	some server-side object, like a component, a custom renderer, or custom validator	custom component or standard component associated with a custom renderer

Understanding the Image Map Example

The `cardemo` application now includes a custom image map component on the `chooseLocale.jsf` page. This image map displays a map of the world. When the user clicks on one of a particular set of regions in the map, the application sets the locale on the `UIViewRoot` of the current `FacesContext` to the language spoken in the selected region. The hot spots of the map are: the United States, Spanish-speaking Central and South America, France, and Germany.

Why Use JavaServer Faces Technology to Implement an Image Map?

JavaServer Faces technology is an ideal framework to use for implementing this kind of image map because it can perform the work that must be done on the server without requiring you to create a server-side image map.

In general, client-side image maps are preferred over server-side image maps for a few reasons. One reason is that the client-side image map allows the browser to provide immediate feedback when a user positions her mouse over a hot spot. Another reason is that client-side image maps perform better because they don't require round-trips to the server. However, in some situations, your image map might need to access the server to retrieve some data or to change the appearance of non-form controls, which a client-side image map cannot do.

The image map custom component—because it uses JavaServer Faces technology—has the best of both style of image maps: It can handle the parts of the application that need to be performed on the server, while allowing the other parts of the application to be performed on the client side.

Understanding the Rendered HTML

Here is an abbreviated version of the form part of the HTML page that the application needs to render:

```
<form id="_id0" method="post"
  action="/jsf-cardemo/chooseLocale.jsf">
  ...
  <img id="_id0:mapImage" src="/jsf-cardemo/images/world.jpg"
     alt="Choose Your Preferred Locale From the Map"
     usemap="#worldMap" />
     <map name="worldMap">
        <area alt="NAmerica"
           coords="53,109,1,110,2,167,,..."
           shape="poly"
           onmouseout=
              "document.forms[0]['_id0:mapImage'].src=
                 '/jsf-cardemo/world.jpg'"
           onmouseover=
              "document.forms[0]['_id0:mapImage'].src=
                 '/jsf-cardemo/world_namer.jpg'"
           onclick=
              "document.forms[0]['worldMap_current'].
                 value=
                    'NAmerica';document.forms[0].submit()"
        />
        <input type="hidden" name="worldMap_current">
     </map>
     ...
</form>
```

The `img` tag associates an image (`world.jpg`) with an image map, referenced in the `usemap` attribute value.

The `map` tag specifies the image map and contains a set of `area` tags.

Each `area` tag specifies a region of the image map. The `onmouseover`, `onmouseout`, and `onmouseclick` attributes define which JavaScript code is executed when these events occur. When the user moves her mouse over a region, the `onmouseover` function associated with the region displays the map with that region highlighted. When the user moves her mouse out of a region, the `onmouseout` function redisplays the original image. If the user clicks on a region, the `onclick` function sets the value of the `input` tag to the id of the selected area and submits the page.

The `input` tag represents a hidden control that stores the value of the currently-selected area between client/server exchanges so that the server-side component classes can retrieve the value.

The server side objects retrieve the value of `worldMap_current` and set the locale in the `FacesContext` according to what region was selected.

Understanding the JSP Page

Here is an abbreviated form of the JSP page that the image map component will use to generate the HTML page shown in the previous section:

```
<f:loadBundle basename="carstore.bundles.Resources"
     var="bundle"/>
<f:view>
<h:form>
  ...
  <h:graphic_image id="mapImage" url="/images/world.jpg"
     alt="#{bundle.chooseLocale}"
     usemap="#worldMap" />
  <d:map id="worldMap"
     current="NAmericas">
     immediate="true"
     action="storefront"
     actionListener="#{carstore.chooseLocaleFromMap}"
     <d:area id="NAmerica" value="#{NA}"
        onmouseover="/images/world_namer.jpg"
        onmouseout="/images/world.jpg"
        targetImage="mapImage"/>
     ...
  </d:map>
  ...
</h:form>
</f:view>
```

The `alt` attribute of `graphic_image` maps to the localized string, "Choose Your Locale from the Map".

The `actionListener` attribute of the `map` tag points at a method in `CarStore` that accepts an `ActionEvent`. This method changes the locale according to the area selected from the image map. The way this event is handled is explained more in Handling Events for Custom Components (page 936).

The `action` attribute specifies a logical outcome `String`, which is matched against the navigation rules in the application configuration file. For more information on navigation, see section Navigating Between Pages (page 896).

The `immediate` attribute of the map tag is set to `true`, which indicates that the default `ActionListener` should execute during the Apply Request Values phase of the request processing lifecycle, instead of waiting for the Invoke Application phase. Since the request resulting from clicking the map does not require any validation, data conversion, or server-side object updates, it makes since to skip directly to the Invoke Application phase.

The `current` attribute of the `map` tag is set to the default area, which is `NAmerica`.

Notice that the `area` tags do not contain any of the JavaScript, coordinate, or shape data that is displayed on the HTML page. The JavaScript is generated by the `AreaRenderer` class. The `onmouseover` and `onmouseout` attribute values indicate the image to be loaded when these events occur. How the JavaScript is generated is explained more in Performing Encoding (page 925).

The coordinate, shape, and alt data are obtained through the `value` attribute, whose value refers to an attribute in application scope. The value of this attribute is a bean, which stores the coordinate, shape, and alt data. How these beans are stored in the application scope is explained more in Simplifying the JSP Page (page 912).

Simplifying the JSP Page

One of the primary goals of JavaServer Faces technology is ease-of-use. This includes separating out the code from the page so that a wider range of page authors can easily contribute to the Web development process. For this reason, all JavaScript is rendered by the component classes rather than being included in the page.

Ease-of-use also includes compartmentalizing the tasks of developing a Web application. For example, rather than requiring the page author to hardcode the coordinates of the hot spots in the page, the application should allow the coordinates to be retrieved from a database or generated by one of the many image map tools available.

In a JavaServer Faces application, data such as coordinates would be retrieved via a bean from the `value` attribute. However, the shape and coordinates of a hotspot should be defined together because the coordinates are interpreted differently depending on what shape the hotspot is. Since a component's `value` can only be bound to one property, the `value` attribute cannot refer to both the shape and the coordinates.

To solve this problem, the application encapsulates all of this information in a set of `ImageArea` objects. These objects are initialized into application scope by the Managed Bean Facility (see Backing Bean Management, page 799). Here is part of the managed-bean declaration for the `ImageArea` bean corresponding to the South America hotspot:

```
<managed-bean>
  ...
  <managed-bean-name>SA</managed-bean-name>
  <managed-bean-class>
    components.model.ImageArea
  </managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <managed-property>
    <property-name>shape</property-name>
    <value>poly</value>
  </managed-property>
  <managed-property>
    <property-name>alt</property-name>
    <value>SAmerica</value>
  </managed-property>
  <managed-property>
    <property-name>coords</property-name>
    <value>89,217,95,100...</value>
  </managed-property>
</managed-bean>
```

For more information on initializing managed beans with the Managed Bean Facility, see section Using the JavaServer Faces Tag Libraries (page 813).

The `value` attributes of the `area` tags refer to the beans in the application scope, as shown in this `area` tag from `chooseLocale.jsf`:

```
<d:area id="NAmerica"
    value="#{NA}"
    onmouseover="/images/world_namer.jpg"
    onmouseout="/images/world.jpg" />
```

To reference the `ImageArea` model object bean values from the component class, you need to implement a `getValue` method in a component class. This method calls `super.getValue`. The superclass of `AreaComponent`, `UIOutput`, has a `getValue` method that does the work of finding the `ImageArea` object associated with `AreaComponent`. The `AreaRenderer` class, which needs to render the alt, shape, and coords values from the `ImageArea` object, calls the `getValue` method of `AreaComponent` to retrieve the `ImageArea` object.

```
ImageArea iarea = (ImageArea) area.getValue();
```

`ImageArea` is just a simple bean, so you can access the shape, coordinates, and alt values by calling the appropriate accessor methods of `ImageArea`. Performing Encoding (page 925) explains how to do this in the `AreaRenderer` class.

Summary of the Application Classes

Table 22–2 summarizes all of the classes needed to implement the image map component.

Table 22–2 Image Map Classes

Class	Function
`AreaSelectedEvent`	The `ActionEvent` indicating that an `AreaComponent` from the `MapComponent` has been selected
`AreaTag`	The tag handler that implements the area custom tag
`MapTag`	The tag handler that implements the map custom tag

Table 22–2 Image Map Classes (Continued)

Class	Function
`AreaComponent`	The class that defines the `AreaComponent` component, corresponding to the area custom tag
`MapComponent`	The class that defines the `MapComponent` component, corresponding to the map custom tag
`AreaRenderer`	This `Renderer` performs the delegated rendering for `AreaComponent`.
`ImageArea`	The bean that stores the shape and coordinates of the hot spots
`CarStore`	The backing-file bean for the cardemo application

`AreaSelectedEvent` and `AreaSelectedListener` are located in `<JSF_HOME>/samples/components/src/components/components`.

`AreaTag` and `MapTag` are located in `<JSF_HOME>/samples/components/src/components/taglib/`.

`AreaComponent` and `MapComponent` are located in `<JSF_HOME>/samples/components/src/components/components/`.

`AreaRenderer` is located in `<JSF_HOME>/samples/components/src/components/renderkit/`.

`ImageArea` is located in `<JSF_HOME>/samples/components/src/components/model/`.

`CarStore` is located in `<JSF_HOME>/samples/cardemo/src/carstore/`.

Steps for Creating a Custom Component

Before describing how the image map works, it helps to summarize the basic steps needed to create an application that uses custom components. You can apply the following steps while developing your own custom component example.

1. Write a tag handler class that extends `javax.faces.webapp.UIComponentTag`. In this class, you need:
   - A `getRendererType` method, which returns the type of your custom renderer, if you are using one (explained in step 4).
   - A `getComponentType` method, which returns the type of the custom component.
   - An `setProperties` method, in which you set all of the new attributes of your component.
2. Create a tag library descriptor (TLD) that defines the custom tag.
3. Create a custom component class
4. Include the rendering code in the component class or delegate it to a renderer (explained in step 7).
5. If your component generates events, queue the event on the component.
6. Save and restore the component state.
7. Delegate rendering to a renderer if your component does not handle the rendering.
   a. Create a custom renderer class by extending `javax.faces.render.Renderer.`
   b. Register the renderer to a render kit.
   c. Identify the renderer type in the component tag handler.
8. Register the component
9. Create an event handler if your component generates events.
10. Declare your new TLD in your JSP page and use the tag in the page.

Creating the Component Tag Handler

If you've created your own JSP custom tags before, creating a component tag and tag handler should be easy for you.

In JavaServer Faces applications, the tag handler class associated with a component drives the Render Response phase of the JavaServer Faces lifecycle. For more information on the JavaServer Faces lifecycle, see The Lifecycle of a JavaServer Faces Page (page 783).

The first thing that the tag handler does is retrieve the type of the component associated with the tag. Next, it sets the component's attributes to the values given in the page. Finally, it returns the type of the renderer (if there is one) to the JavaServer Faces implementation so that the component's encoding can be performed when the tag is processed.

The image map custom component includes two tag handlers: `AreaTag` and `MapTag`. To see how the operations on a JavaServer Faces tag handler are implemented, let's take a look at `MapTag`:

```
public class MapTag extends UIComponentTag {
  private String current = null;
  public void setCurrent(String current) {
    this.current = current;
  }
  private String actionListener = null;
  public void setActionListener(String actionListener) {
    this.actionListener = actionListener;
  }
  private String action = null;
  public void setAction(String action) {
    this.action = action;
  }
  private String immediate = null;
  public void setImmediate(String immediate) {
    this.immediate = immediate;
  }
  private String styleClass = null;
  public void setStyleClass(String styleClass) {
    this.styleClass = styleClass;
  }
  public String getComponentType() {
    return ("DemoMap");
  }
  public String getRendererType() {
    return ("DemoMap");
  }
  public void release() {
    super.release();
    current = null;
    styleClass = null;
    actionListener = null;
    action = null;
    immediate = null;
  }
  protected void setProperties(UIComponent component) {
    super.setProperties(component);
```

```
MapComponent map = (MapComponent) component;
if (styleClass != null) {
   if (isValueReference(styleClass)) {
      ValueBinding vb =
         FacesContext.getCurrentInstance().
            getApplication().
               createValueBinding(styleClass);
      map.setValueBinding("styleClass", vb);
   } else {
      map.getAttributes().put("styleClass", styleClass);
   }
}
if(actionListener != null) {
   if(isValueReference(actionListener)) {
      Class args[] = {ActionEvent.class};
      MethodBinding mb =
         FacesContext.getCurrentInstance().
            getApplication().
               createMethodBinding(actionListener, args);
      map.setActionListener(mb);
   } else {
      Object params[] = {actionListener};
      throw new javax.faces.FacesException();
   }
}
if (action != null) {
   if (isValueReference(action)) {
      MethodBinding vb = FacesContext.
         getCurrentInstance().getApplication().
            createMethodBinding(action, null);
      map.setAction(vb);
   } else {
      map.setAction(
         Util.createConstantMethodBinding(action));
   }
}
if (immediate != null) {
   if (isValueReference(immediate)) {
      ValueBinding vb = FacesContext.
         getCurrentInstance().getApplication().
            createValueBinding(immediate);
      map.setValueBinding("immediate", vb);
   } else {
      boolean _immediate =
         new Boolean(immediate).booleanValue();
```

```
            map.setImmediate(_immediate);
         }
      }
   }
```

The first thing to notice is that `MapTag` extends `UIComponentTag`, which supports `jsp.tagext.Tag` functionality as well as JavaServer Faces-specific functionality. `UIComponentTag` is the base class for all JavaServer Faces tags that correspond to a component. Tags that need to process their tag bodies should subclass `UIComponentBodyTag` instead.

As explained above, the first thing `MapTag` does is to retrieve the type of the component. This is done with the `getComponentType` operation,:

```
public String getComponentType() {
   return ("DemoMap");
}
```

Next, the tag handler sets the component's attribute values to those supplied as tag attributes in the page. The `MapTag` handler gets the attribute values from the page via JavaBeans properties that correspond to the attributes. `MapComponent` has several attributes. Here is the property used to access the value of `immediate`:

```
private String immediate = null;
public void setImmediate(String immediate) {
   this.immediate = immediate;
}
```

To pass the value of the tag attributes to `MapComponent`, the tag handler implements the `setProperties` method.

Some tag attributes can refer to literal values or use JSF EL expressions, which point to values typically stored in a bean. It is recommended to enable your component attributes to accept JSF EL expresssions because this is what a page author expects.

If you do make your tag attributes accept JSF EL expresssions and you are updating a property of the underlying component then the component property needs to be enabled for JSF EL expressions as well. See Creating Custom Component Classes (page 922) for more information on doing this. In addition, an attribute that accepts a JSF EL expression must be of type `String`. This is why `immediate` is of type `String`, as shown in the previous code snippet.

For each MapComponent attribute that accepts a JSF EL expression, the setProperties method needs to either get a MethodBinding or a ValueBinding for it from the Application instance. A ValueBinding is an object used to evaluate value reference expressions that refer to backing-bean properties. A MethodBinding is an object used to evaluate reference expressions that refer to backing-bean methods.

For example, the value of the actionListener attribute must be a JSF EL expression that points to a method on a backing bean that takes an ActionEvent as its argument. The setProperties method of MapTag creates a MethodBinding for the actionListener attribute, passing in the signature that this method must have, and it sets the MethodBinding as the value of the actionListener attribute of the MapComponent.

The action attribute can take a literal String or a JSF EL expression that points to a backing-bean method that takes no parameters and returns a literal String. To handle the case of the literal String, the setProperties method creates a special "constant" method binding around the literal String in order to satisfy the requirement that the argument to the action attribute of the MapComponent is a MethodBinding instance. The components demo provides a utility method for this purpose. To handle the JSF EL expression, setProperties just creates the MethodBinding as it does for the actionListener attribute.

MapComponent's immediate attribute value is a JSF EL expression. This expression points to a backing-bean property. Therefore, setProperties needs to obtain a ValueBinding for it. After obtaining the ValueBinding, the setProperties method sets the value of the property on the MapComponent by calling the MapComponent's setValueBinding method, passing in the ValueBinding obtained from the Application and the name of the attribute.

The following piece of setProperties sets the immediate property of MapComponent:

```
...
if (immediate != null) {
  if (isValueReference(immediate)) {
    ValueBinding vb = FacesContext.
      getCurrentInstance().getApplication().
        createValueBinding(immediate);
    map.setValueBinding("immediate", vb);
  } else {
    boolean _immediate =
```

```
            new Boolean(immediate).booleanValue();
        map.setImmediate(_immediate);
    }
}
```

Finally, the tag handler provides a renderer type—if there is a renderer associated with the component—to the JavaServer Faces implementation. It does this with the `getRendererType` method:

```
    public String getRendererType() {return "DemoMap";}
```

The renderer type that is returned is the name under which the renderer is registered with the application. See Delegating Rendering to a Renderer (page 931) for more information. If your component does not have a renderer associated with it, `getRendererType` should return `null`.

It's recommended practive that all tag handlers are also required to implement a `release` method, which releases resources allocated during the execution of the tag handler. The release method of `MapTag` is:

```
public void release() {
    super.release();
    current = null;
    styleClass = null;
    actionListener = null;
    immediate = null;
    action = null;
}
```

This method first calls the `UIComponentTag.release` method to release resources associated with `UIComponentTag`. Next, the method sets all attribute values to `null`.

Defining the Custom Component Tag in a Tag Library Descriptor

To define a tag, you need to declare it in a tag library descriptor (TLD), which is an XML document that describes a tag library. A TLD contains information about a library and each tag contained in the library. TLDs are used by a Web container to validate the tags. The set of tags that are part of the HTML render kit are defined in the `html_basic` TLD.

The custom tags, `area` and `map`, are defined in `components.tld`, which is stored in the `components/src/components/taglib` directory of your installation. The `components.tld` defines tags for all of the custom components included in this release.

All tag definitions must be nested inside the `taglib` element in the TLD. Each tag is defined by a `tag` element. Here is part of the tag definition of the `map` tag:

```
<tag>
  <name>map</name>
  <tag-class>components.taglib.MapTag</tag-class>
  <attribute>
    <name>binding</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>current</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>id</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  ...
</tag>
```

At a minimum, each tag must have a `name` (the name of the tag) and a `tag-class` (the tag handler) attribute. For more information on defining tags in a TLD, please consult the Tag Library Descriptors (page 729) section of this tutorial.

Creating Custom Component Classes

As explained in When to Use a Custom Component (page 906), a component class defines the state and behavior of a UI component. Some of the state information includes the component's type, identifier, and local value. Some of the behavior defined by the component class includes:

- Decoding (converting the request parameter to the component's local value)
- Encoding (converting the local value into the corresponding markup)
- Saving the state of the component
- Updating the bean value with the local value
- Processing validation on the local value
- Queueing events

The `UIComponentBase` class defines the default behavior of a component class. All of the classes representing the standard components extend from `UIComponentBase`. These classes add their own behavior definitions, as your custom component class will do.

Your custom component class needs to either extend `UIComponentBase` directly or extend a class representing one of the standard components. These classes are located in the `javax.faces.component` package and their names begin with UI.

If your custom component serves the same purpose as a standard component, you should extend that standard component rather than directly extending `UIComponentBase`. For example, suppose you want to create an editable menu component. It makes sense to have this component extend `UISelectOne` rather than `UIComponentBase` because you can reuse the behavior already defined in `UISelectOne`. The only new functionality you need to define is that which makes the menu editable.

Whether you decide to have your component extend `UIComponentBase` or a standard component, you might also want your component to implement one or more of these behavioral interfaces:

- `ActionSource`: indicates that the component can fire an `ActionEvent`
- `ConvertibleValueHolder`: indicates that the component supports data type conversions
- `NamingContainer`: Mandates that all components rooted at this component have unique IDs
- `StateHolder`: Denotes that a component has state that needs to be saved between requests
- `ValueHolder`: Indicates that the component maintains a local value as well as the option of accessing data in the model tier.

If your component extends `UICommand`, it automatically implements `ActionSource`, `StateHolder`, and `ValueHolder`. If your component extends `UIOutput` or one of the components classes that extend `UIOutput`, it automatically implements `ConvertibleValueHolder`, `StateHolder`, and `ValueHolder`. If your component extends `UIComponentBase`, it automatically implements only `StateHolder`. See the JavaServer Faces API Javadoc to find out what the other component classes implement.

So if you created a component that extends `UIInput`, for example, and you want this component to fire events, your component also needs to implement `ActionSource`.

The image map example has two component classes: `AreaComponent` and `MapComponent`. The `MapComponent` class extends `UIComponentBase`. The `AreaComponent` class extends the standard component, `UIOutput`.

The `MapComponent` class represents the component corresponding to the `map` tag:

```
<d:map id="worldMap" current="NAmericas" immediate="true"
  action="storeFront"
  actionListener="#{carstore.chooseLocaleFromMap}">
```

The `AreaComponent` class represents the component corresponding to the `area` tag:

```
<d:area id="NAmerica" value="#{NA}"
  onmouseover="/images/world_namer.jpg"
  onmouseout="/images/world.jpg"
  targetImage="mapImage" />
```

`MapComponent` has one or more `AreaComponents` as children. Its behavior consists of:

- Retrieving the value of the currently-selected area.
- Defines the properties corresponding to the component's values
- Generating an event when the user clicks on the image map
- Queuing the event
- Saving its state
- Rendering the `map` tag and the `input` tag

The rendering of the map and input tags are actually performed by the `MapRenderer`, but `MapComponent` delegates this rendering to `MapRenderer`.

The `MapComponent` class also implements `ActionSource` so that it can fire an `ActionEvent` when a user clicks on the map.

The `AreaComponent` class extends `UIOutput` because `AreaComponent` requires a `value` attribute, which is already defined by `UIOutput`.

`AreaComponent` is bound to a bean that stores the shape and coordinates of the region of the image map. You'll see how all of this data is accessed through the `value` expression in Create the Renderer Class (page 931). The behavior of the `AreaComponent` component consists of:

- Retrieving the shape and coordinate data from the bean
- Setting the value of the hidden tag to the `id` of this component
- Rendering the `area` tag, including the JavaScript for the `onmouseover`, `onmouseout`, and `onclick` functions

Although these tasks are actually performed by `AreaRenderer`, the `AreaComponent` class must delegate the tasks to `AreaRenderer`. See Delegating Rendering to a Renderer (page 931) for more information.

The rest of this section details how the `MapComponent` class performs encoding and decoding, how it defines properties for the component's local values, and how it saves the state of `MapComponent`. Handling Events for Custom Components (page 936) details how `MapComponent` handles events.

Performing Encoding

During the Render Response phase, the JavaServer Faces implementation processes the encoding methods of all components and their associated renderers in the tree. The encoding methods convert the current local value of the component into the corresponding markup that represents it in the response.

The `UIComponentBase` class defines a set of methods for rendering markup: `encodeBegin`, `encodeChildren`, `encodeEnd`. If the component has child components, you might need to use more than one of these methods to render the component; otherwise, all rendering should be done in `encodeEnd`.

Since `MapComponent` is a parent component of `AreaComponent`, the `area` tags must be rendered after the beginning `map` tag and before the ending `map` tag. To accomplish this, the `MapRenderer` class renders the beginning `map` tag in `encodeBegin` and the rest of the `map` tag in `encodeEnd`.

The JavaServer Faces implementation will automatically invoke the encodeEnd method of AreaComponent's renderer after it invokes MapRenderer's encodeBegin method and before it invokes MapRenderer's encodeEnd method. If a component needs to perform the rendering for its children, it does this in the encodeChildren method.

Here are the encodeBegin and encodeEnd methods of MapRenderer:

```
public void encodeBegin(FacesContext context,
  UIComponent component) throws IOException {
  if ((context == null)|| (component == null)){
     throw new NullPointerException();
  }
  MapComponent map=(MapComponent) component;
  ResponseWriter writer = context.getResponseWriter();
  writer.startElement("map", map);
  writer.writeAttribute("name", map.getId(),"id");
}

public void encodeEnd(FacesContext context) throws IOException
{
  if ((context == null) || (component == null)){
     throw new NullPointerException();
  }
  MapComponent map = (MapComponent) component;
  ResponseWriter writer = context.getResponseWriter();
  writer.startElement("input", map);
  writer.writeAttribute("type", "hidden", null);
  writer.writeAttribute("name",
     getName(context,map), "clientId");(
  writer.endElement("input");
  writer.endElement("map");
}
```

Notice that encodeBegin renders only the beginning map tag. The encodeEnd method renders the input tag and the ending map tag.

The encoding methods accept a UIComponent argument and a FacesContext argument. The FacesContext contains all of the information associated with the current request. The UIComponent argument is the component that needs to be rendered. The renderer needs to be told what component it is rendering. So you need to pass the component to the encoding methods of the renderer.

The rest of the method renders the markup to the ResponseWriter, which writes out the markup to the current response. This basically involves passing the HTML tag names and attribute names to the ResponseWriter as strings, retrieving the values of the component attributes, and passing these values to the ResponseWriter.

The startElement method takes a String, which is the name of the tag, and the component to which the tag corresponds, in this case, map. (Passing this information to the ResponseWriter helps design-time tools know which portion of the generated markup are related to which components.)

After calling startElement, you can call writeAttribute to render the tag's attributes. The writeAttribute method takes the name of the attribute, it's value, and the name of a property or attribute of the containing component corresponding to the attribute. The last parameter can be null, and it won't be rendered.

The name attribute value of the map tag is retrieved with the getId method of UIComponent, which returns the component's unique identifier. The name attribute vale of the input tag is retrieved with the getName(FacesContext, UIComponent) method of MapRenderer.

If you want your component to perform its own rendering but delegate to a Renderer if there is one, include the following lines in the encode method to check if there is a renderer associated with this component.

```
if (getRendererType() != null) {
   super.encodeEnd(context);
   return;
}
```

If there is a Renderer available, this method invokes the superclass' encodeEnd method, which does the work of finding the renderer. The MapComponent class delegates all rendering to MapRenderer, so it does not need to check for available renderers.

In some custom component classes that extend standard components, you might need to implement additional methods besides encodeEnd. For example, if you need to retrieve the component's value from the request parameters—such as to update a bean's values—you also have to implement the decode method.

Performing Decoding

During the Apply Request Values phase, the JavaServer Faces implementation processes the decode methods of all components in the tree. The decode method extracts a component's local value from incoming request parameters and converts the value to a type acceptable to the component class.

A custom component class needs to implement the `decode` method only if it must retrieve the local value, or it needs to queue events. The `MapComponent` component do both of these tasks. `MapComponent` must retrieve the value of the hidden `input` field and set its `current` attribute to this value. The `setCurrent` method queues the event by calling `queueEvent`, passing in the `AreaSelectedEvent` generated my `MapComponent`.

Here is the `decode` method of `MapComponent`:

```
public void decode(FacesContext context, UIComponent component)
{
  if ((context == null) || (component == null)) {
    throw new NullPointerException();
  }
  MapComponent map = (MapComponent) component;
  String key = getName(context, map);
  String value = (String)context.getExternalContext().
    getRequestParameterMap().get(key);
  if (value != null)
    map.setCurrent(value);
  }
}
```

The `decode` method first gets the name of the hidden `input` field by calling `getName(FacesContext, UIComponent)`. It then uses that name as the key to the request parameter map to retrieve the value current value of the `input` field. This value represents the currently-selected area. Finally, it sets the value of the `MapComponent`'s current attribute to the value of the input field.

Defining Properties for the Component's Values

Creating the Component Tag Handler (page 916) described how `MapTag` sets the component's values when processing the tag. For those component attributes that take JSF EL expressions that point to a backing-bean property, `MapTag` uses a `ValueBinding` to evaluate the expression.

To get the value of a component attribute that accepts a JSF EL expression pointing to a backing-bean property, the component class needs to get the `Value-`

Binding associated with the attribute. For example, here is the isImmediate method from MapComponent that does this:

```
public boolean isImmediate() {
  if (this.immediateSet) {
      return (this.immediate);
  }
  ValueBinding vb = getValueBinding("immediate");
  if (vb != null) {
     Boolean value = (Boolean) vb.getValue(getFacesContext());
     return (value.booleanValue());
  } else {
      return (this.immediate);
  }
}
```

The properties corresponding to the component attribute that accepts a JSF EL expression pointing to a backing-bean method, needs to accept and return a MethodBinding. For example here is the action property of MapComponent:

```
public MethodBinding getAction() {
  return (this.action);
}
public void setAction(MethodBinding action) {
  this.action = action;
}
```

Saving and Restoring State

Because component classes implement StateHolder, they must implement the saveState(FacesContext) and restoreState(FacesContext, Object) methods to help the JavaServer Faces implementation save and resotre the state of your component across multiple requests.

To save a set of values, you need to implement the saveState(FacesContext) method. This method is called during the Render Response phase, during which the state of the response is saved for processing on subsequent requests. Here is the method from MapComponent:

```
public Object saveState(FacesContext context) {
  removeDefaultActionListener(context)
  Object values[] = new Object[6];
  values[0] = super.saveState(context);
```

```
    values[1] = current;
    values[2] = saveAttachedState(context, action);
    values[3] = saveAttachedState(context, actionListener);
    values[4] = immediate ? Boolean.TRUE : Boolean.FALSE;
    values[5] = immediateSet ? Boolean.TRUE : Boolean.FALSE;
    addDefaultActionListener(context);
    return (values);
  }
```

This method first *must* remove the default `ActionListener` so that it is not saved along with the other state. The method then initializes an array, which will hold the saved state. It next saves all of the state associated with `UIComponentBase`, the superclass to `MapComponent`. The calls, `saveAttachedState(context, action)` and `saveAttachedState(context, actionListener)` take the attached objects, `action` and `actionListener` and saves their state while checking if the objects already implement `StateHolder`. The `action` and `actionListener` objects are `MethodBinding` objects, which might represent `StateHolder`. Next, the method saves the values associated with `MapComponent` into the array. Finally, the method adds the default `ActionListener` back to the `FacesContext`.

A component that implements `StateHolder` must also provide an implementation for `restoreState(FacesContext, Object)`, which restores the state of the component to that saved with the `saveState(FacesContext)` method. The `restoreState(FacesContext, Object)` is called during the Restore View phase, during which the JavaServer Faces implementation checks if there is any state that was saved during the last Render Response phase and needs to be restored in preparation for the next post back. Here is the `restoreState(FacesContext, Object)` method from `MapComponent`:

```
  public void restoreState(FacesContext context, Object state) {
    removeDefaultActionListener(context);
    Object values[] = (Object[]) state;
    super.restoreState(context, values[0]);
    current = (String) values[1];
    action = (MethodBinding)
      restoreAttachedState(context, values[2]);
    actionListener = (MethodBinding)
      restoreAttachedState(context, values[3]);
    immediate = ((Boolean) values[4]).booleanValue();
    immediateSet = ((Boolean) values[5]).booleanValue();
    addDefaultActionListener(context);
  }
```

This method takes the `FacesContext` and the `Object`, representing the array that is holding the state for the component. Like `saveState(FacesContext)`, the `restoreState(FacesContext, Object)` *must* remove the default `ActionListener` at the beginning of the method and add it again at the end. The rest of the method sets the component's properties to the values saved in the `Object` array. The `restoreAttachedState(Context, Object)` calls restore the state of the `MethodBinding` objects, `action` and `actionListener`.

When implementing these methods in your component class, be sure to specify in your web.xml file where you want the state to be saved, either client or server. Here is the `context-param` element from the components demo `web.xml` file that specifies that state must be saved in the client:

```
<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>client</param-value>
</context-param>
```

If state is saved on the client, the state of the entire view is rendered to a hidden field on the page.

Delegating Rendering to a Renderer

Both the `MapComponent` and the `AreaComponent` delegate all of their rendering to a separate renderer. The section Performing Encoding (page 925) explained how the `MapRenderer` performs the encoding for the `MapComponent`. This section explains the process of delegating rendering to a renderer in more detail using the `AreaRenderer`, which performs the rendering for `AreaComponent`.

To delegate rendering, you need to perform these tasks:

- Create the renderer class
- Register the renderer with a render kit
- Identify the renderer type in the component's tag handler

Create the Renderer Class

When delegating rendering to a renderer, you can delegate all encoding and decoding to the renderer, or you can choose to do part of it in the component class. The `AreaComponent` class only requires encoding.

To perform the rendering for AreaComponent, the AreaRenderer needs to implement an encodeEnd method. The encodeEnd method of AreaRenderer must retrieve the shape, coordinates, and alt values stored in the ImageArea bean that is bound to AreaComponent. Suppose that the area tag currently being rendered has a value attribute value of "fraA". The following line from encodeEnd gets the value of the attribute "fraA" from the FacesContext.

```
ImageArea ia = (ImageArea)area.getValue();
```

The attribute value is the ImageArea bean instance, which contains the shape, coordinates, and alt values associated with the fraA AreaComponent instance. Simplifying the JSP Page (page 912) describes how the application stores these values.

After retrieving the ImageArea object, you render the values for shape, coords, and alt by simply calling the associated accessor methods and passing the returned values to the ResponseWriter, as shown by these lines of code, which write out the shape and coordinates:

```
writer.startElement("area", area);
writer.writeAttribute("alt", iarea.getAlt(), "alt");
writer.writeAttribute("coords", iarea.getCoords(), "coords");
writer.writeAttribute("shape", iarea.getShape(), "shape");
```

The encodeEnd method also renders the JavaScript for the onmouseout, onmouseover, and onclick attributes. The page author only needs to provide the path to the images that are to be loaded during an onmouseover or onmouseout action:

```
<d:area id="France" value="#{fraA}"
  onmouseover="/images/world_france.jpg"
  onmouseout="/images/world.jpg" targetImage="mapImage" />
```

The AreaRenderer class takes care of generating the JavaScript for these actions, as shown in the code below from encodeEnd. The JavaScript that AreaRenderer generates for the onclick action sets the value of the hidden field to the value of the current area's component ID and submits the page.

```
sb = new StringBuffer("document.forms[0]['").
  append(targetImageId).append("'].src='");
sb.append(getURI(context,
  (String) area.getAttributes().get("onmouseout")));
sb.append("'");
writer.writeAttribute("onmouseout", sb.toString(),
```

```
  "onmouseout");
sb = new StringBuffer("document.forms[0]['").
  append(targetImageId).append("'].src='");
sb.append(getURI(context,
  (String) area.getAttributes().get("onmouseover")));
sb.append("'");
writer.writeAttribute("onmouseover", sb.toString(),
  "onmouseover");
sb = new StringBuffer("document.forms[0]['");
sb.append(getName(context, area));
sb.append("'].value='");
sb.append(iarea.getAlt());
sb.append("'; document.forms[0].submit()");
writer.writeAttribute("onclick", sb.toString(), "value");
writer.endElement("area");
```

By submitting the page, this code causes the JavaServer Faces lifecycle to return back to the RestoreView phase. This phase saves any state information—including the value of the hidden field—so that a new request component tree is constructed. This value is retrieved by the `decode` method of the `MapComponent` class. This `decode` method is called by the JavaServer Faces implementation during the Apply Request Values phase, which follows the RestoreView phase.

In addition to the `encodeEnd` method, `AreaRenderer` also contains an empty constructor. This will be used to create an instance of `AreaRenderer` so that it can be added to the render kit.

Note that `AreaRenderer` extends `BaseRenderer`, which in turn extends `Renderer`. The `BaseRenderer` class is included in *<JWSDP_HOME>*`/jsf/samples/components/src/components/renderkit/`. It contains definitions of the `Renderer` class methods so that you don't have to include them in your renderer class.

Register the Renderer with a Render Kit

For every UI component that a render kit supports, the render kit defines a set of `Renderer` objects that can render the component in different ways to the client supported by the render kit. For example, the standard `UISelectOne` component class defines a component that allows a user to select one item out of a group of items. This component can be rendered with the `Listbox` renderer, the `Menu` renderer, or the `Radio` renderer. Each renderer produces a different appearance for the component. The `Listbox` renderer renders a menu that displays all possible values. The `Menu` renderer renders a subset of all possible values. The `Radio` renderer renders a set of radio buttons.

When you create a custom renderer, you need to register it with the appropriate render kit. Since the image map application implements an HTML image map, `AreaRenderer` (as well as `MapRenderer`) should be registered with the HTML render kit.

You register the renderer using the application configuration file (see Application Configuration, page 801):

```
<render-kit>
  <renderer>
    <renderer-type>DemoArea</renderer-type>
    <renderer-class>
      components.renderkit.AreaRenderer
    </renderer-class>
    <attribute>
      <attribute-name>onmouseout</attribute-name>
      <attribute-class>java.lang.String</attribute-class>
    </attribute>
    <attribute>
      <attribute-name>onmouseover</attribute-name>
      <attribute-class>java.lang.String</attribute-class>
    </attribute>
    <attribute>
      <attribute-name>styleClass</attribute-name>
      <attribute-class>java.lang.String</attribute-class>
    </attribute>
    <supported-component-class>
      <component-class>
        components.components.AreaComponent
      </component-class>
    </supported-component-class>
  </renderer>
```

The `render-kit` element represents a `RenderKit` implementation. If no `render-kit-id` is specified, the default HTML render kit is assumed. The renderer element represents a `Renderer` implementation. By nesting the `renderer` element inside the `render-kit` element, you are registering the renderer with the `RenderKit` associated with the `render-kit` element.

The `renderer-type` will be used by the tag handler, as explained in the next section. The `renderer-class` is the fully-qualified classname of the `Renderer`.

Each of the attribute tags specify the render-dependent attributes and their types.

The `supported-component-class` tag specifies the UI component that delegates its rendering to this renderer.

Identify the Renderer Type

During the Render Response phase, the JavaServer Faces implementation calls the `getRendererType` method of the component's tag to determine which renderer to invoke, if there is one.

The `getRendererType` method of `AreaTag` must return the type associated with `AreaRenderer`. Recall that you identified this type when you registered `AreaRenderer` with the render kit. Here is the `getRendererType` method from the `cardemo` application's `AreaTag` class:

```
public String getRendererType() { return ("DemoArea");}
```

Register the Component

After writing your component classes, you need to register them with the application using the application configuration file (see Application Configuration, page 801).

Here are the declarations that register the `AreaComponent`:

```
<component>
  <component-type>DemoArea</component-type>
  <component-class>
    components.components.AreaComponent
  </component-class>
  <property>
    <property-name>alt</property-name>
    <property-class>java.lang.String</property-class>
  </property>
  <property>
    <property-name>coords</property-name>
    <property-class>java.lang.String</property-class>
  </property>
  <property>
    <property-name>shape</property-name>
    <property-class>java.lang.String</property-class>
  </property>
</component>
```

The `component-type` element indicates the name under which the component should be registered. Other objects referring to this component use this name. The `component-class` element indicates the fully-qualified class name of the component. The `property` elements specify the component properties and their types.

Handling Events for Custom Components

As explained in Handling Events (page 889), events are automatically queued on standard components that fire events. A custom component on the other hand must manually queue events from its `decode` method if it fires events.

Performing Decoding (page 927) explained how to queue an event on `MapComponent` in its `decode` method. This section explains how to write the class representing the event of clicking on the map and how to write the method that processes this event.

As explained in Understanding the JSP Page (page 911), the `actionListenerRef` attribute of the `map` tag points to the `chooseLocaleFromMap` method of the bean, `CarStore`. This method processes the event of clicking the image map. Here is the `chooseLocaleFromMap` method of `CarStore`:

```
public void chooseLocaleFromMap(ActionEvent actionEvent) {
  AreaSelectedEvent event = (AreaSelectedEvent) actionEvent;
  String current = event.getMapComponent().getCurrent();
  FacesContext context = FacesContext.getCurrentInstance();
  context.getViewRoot().setLocale((Locale)
    locales.get(current));
  ...
}
```

When the JavaServer Faces implementation calls this method, it passes in an `ActionEvent`, representing the event generated by clicking on the image map. Next, it casts it to an `AreaSelectedEvent`. Then, this method gets the `MapComponent` associated with the event. It then gets the value of the `MapComponent`'s current attribute, which indicates the currently-selected area. The method then uses the value of the current property to get the `Locale` object from a `HashMap`, which is constructed elsewhere in the `CarStore` class. Finally the method sets the locale of the `FacesContext` to the `Locale` obtained from the `HashMap`.

In addition to the method that processes the event, you also need the event class itself. For the image map, this is the AreaSelectedEvent class. This class is very simple to write. You only need to have it extend ActionEvent and provide a constructor that takes the component on which the event is queued and a method that returns the component:

```
public class AreaSelectedEvent extends ActionEvent {
  ...
  public AreaSelectedEvent(MapComponent map) {
    super(map);
  }
  public MapComponent getMapComponent() {
    return ((MapComponent) getComponent());
  }
}
```

As explained in section Creating Custom Component Classes (page 922), in order for the MapComponent to fire events in the first place, it must implement ActionSource. You could have MapComponent extend UICommand instead of implementing ActionSource directly so that you could avoid having to provide an implementation of all of the methods that are already implemented by UICommand. However, a MapComponent doesn't really serve the same purpose as a UICommand. The MapComponent does not perform any application-specific processing as a result of firing the event, like a UICommand should do.

The queueEvent(FacesEvent) method of MapComponent sets the PhaseId to APPLY_REQUEST_VALUES, as specified by the component's immediate property. As a result, the event is processed during the Apply Request Values phase rather than the Invoke Application phase. This makes sense because the MapComponent does not require any validation, data conversion, or model updates, so the phases that take care of those tasks can be skipped.

MapComponent also implements methods for adding a default listener and removing a default listener. These methods are called during the restoreState(FacesContext, Object) and saveState(FacesContext) methods. See Saving and Restoring State (page 929) for more information on these methods.

Finally, MapComponent also implements the broadcast method, which broadcasts events to interested listeners. In MapComponent's implementation, the broadcast method broadcasts the AreaSelectedEvent to the processAreaSelected method of the CarStore bean. The broadcast method is called by the processDecodes method of UIViewRoot during the Apply Request Values phase of the lifecycle.

Using the Custom Component in the Page

After you've created your custom component and written all the accompanying code, you are ready to use the component from the page.

To use the custom component in the JSP page, you first need to package the TLD in your application. TLD files are stored in the `WEB-INF` directory or subdirectory of the WAR file or in the `META-INF/` directory or subdirectory of a tag library packaged in a JAR.

Next, you need to declare the custom tag library that defines the custom tag corresponding to the custom component. The tag library is described in Defining the Custom Component Tag in a Tag Library Descriptor (page 921).

To declare the custom tag library, include a `taglib` directive at the top of each page that will contain the tags included in the tag library. Here is the `taglib` directive that declares the JavaServer Faces components tag library:

```
<%@ taglib uri="http://java.sun.com/jsf/demo/components"
  prefix="d" %>
```

When referring to your own tag library packaged in your application, you would include a `taglib` directive such as this one:

```
<%@ taglib uri="WEB-INF/lib/mytags.tld" prefix="mytag" %>
```

The `uri` attribute value uniquely identifies the tag library. The `prefix` attribute value is used to distinguish tags belonging to the tag library. For example, the `map` tag must be referenced in the page with the `d` prefix, like this:

```
<d:map ...>
```

Don't forget to also include the `taglib` directive for the standard tags included with the RI:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
```

When you reference any JavaServer Faces tags—custom or standard—from within a JSP page, you must enclose all of them in the `view` tag:

```
<f:view>
  ... other faces tags, custom tags, and possibly mixed with
  other content
</f:view>
```

All form elements must also be enclosed within the `form` tag, which is also nested within the `view` tag:

```
<f:view>
  <h:form>
     ... other faces tags, custom tags, and possibly mixed with
     other content
  </h:form>
<f:view>
```

The `form` tag encloses all of the controls that display or collect data from the user.

Now that you've set up your page, you can add the custom tags in between the `form` tags, as shown here in the `chooseLocale.jsf` page:

```
...
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/demo/components"
          prefix="d" %>
<f:loadBundle basename="carstore.bundles.Resources"
     var="bundle"/>
<f:view>
  <h:form>
     ...
     <h:graphic_image id="mapImage" url="/images/world.jpg"
        alt="#{bundle.chooseLocale}"
        usemap="#worldMap" />
        <d:map id="worldMap" current="NAmericas"
          immediate="true"
          action="storeFront"
          actionListener="#{carstore.chooseLocaleFromMap}">
          <d:area id="NAmerica" value="#{NA}"
             onmouseover="/images/world_namer.jpg"
             onmouseout="/images/world.jpg"
```

```
                targetImage="mapImage" />
        </d:map>
    </h:form>
</f:view>
```

Further Information

For further information on the technologies discussed in this tutorial see the Web sites listed below.

- The JavaServer Faces Web site
  `http://java.sun.com/j2ee/javaserverfaces`
- The JavaServer Faces 1.0 Specification
  `http://java.sun.com/j2ee/javaserverfaces/download.html`
- The Java Servlets Web site
  `http://java.sun.com/products/servlet`
- The JavaServer Pages Web site
  `http://java.sun.com/products/jsp`
- The JavaServer Pages Standard Tag Library Web site
  `http://java.sun.com/products/jsp/jstl`

23

Internationalizing and Localizing Web Applications

Internationalization is the process of preparing an application to support more than one language and data format. *Localization* is the process of adapting an internationalized application to support a specific region or locale. Examples of locale-dependent information include messages and user interface labels, character sets and encoding, and date and currency formats.Although all client user interfaces should be internationalized and localized, it is particularly important for Web applications because of the global nature of the Web.

Java Platform Localization Classes

In the Java 2 platform, `java.util.Locale` represents a specific geographical, political, or cultural region. The string representation of a locale consists of the international standard 2-character abbreviation for language and country and an optional variant, all separated by underscore _ characters. Examples of locale strings include fr (French), de_CH (Swiss German), and en_US_POSIX (United States English on a POSIX-compliant platform).

Locale-sensitive data is stored in a `java.util.ResourceBundle`. A resource bundle contains key-value pairs, where the keys uniquely identify a locale-specific object in the bundle. A resource bundle can be backed by a text file (properties resource bundle) or a class (list resource bundle) containing the pairs. A resource bundle instance is constructed by appending a locale string representation to a base name.

For more details on internationalization and localization in the Java 2 platform, see

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

In the Web technology chapters, the Duke's Bookstore example contains resource bundles with the base name `messages.BookstoreMessages` for the locales `en_US` and `es_ES`. See Internationalization Tags (page 696) for information on the JSTL i18n tags.

Providing Localized Messages and Labels

Messages and labels should be tailored according to the conventions of a user's language and region. There are two approaches to providing localized messages and labels in 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 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.

The Duke's Bookstore application follows the second approach. Here are a few lines from the default resource bundle `messages.BookstoreMessages.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 using the `getLocale` method, opens the resource bundle for that locale, and then saves the bundle as a session attribute (see Associating Attributes with a Session, page 633):

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

The JSP versions of the Duke's Bookstore application uses the `fmt:message` tag to provide localized strings for introductory messages, HTML link text, button labels, and error messages. For more information on the JSTL messaging tags, see Messaging Tags (page 697).

Date and Number Formatting

Java programs use the `DateFormat.getDateInstance(int, locale)` to parse and format dates in a locale-sensitive manner. Java programs use the `NumberFormat.get`*`XXX`*`Instance(locale)` method, where *`XXX`* can be `Currency`, `Number`, or `Percent`, to parse and format numerical values in a locale-sensitive manner. The servlet version of Duke's Bookstore uses the currency version of this method to format book prices.

JSTL applications use the `fmt:formatDate` and `fmt:parseDate` tags to handle localized dates, and `fmt:formatNumber` and `fmt:parseNumber` tags to handle localized numbers, including currency values. For more information on the JSTL formatting tags, see Formatting Tags (page 698). The JSTL version of Duke's bookstore uses the `fmt:formatNumber` tag to format book prices and the `fmt:formatDate` tag to format delivery dates.

Character Sets and Encodings

Character Sets

A *character set* is a set of textual and graphic symbols, each of which is mapped to a set of nonnegative integers.

The first character set used in computing was ASCII. It is limited in that it can only represent American English. ASCII contains upper- and lower-case Latin alphabets, numerals, punctuation, a set of control codes, and a few miscellaneous symbols.

Unicode defines a standardized, universal character set that can be extended to accommodate additions. Unicode characters may be represented as escape sequences, using the notation `\u`*`XXXX`*, where *`XXXX`* is the character's 16-bit representation in hexadecimal when the Java program source file encoding doesn't support Unicode. For example, the Spanish version of the Duke's Bookstore message file uses Unicode for non-ASCII characters:

```
{"TitleCashier", "Cajero"},
{"TitleBookDescription", "Descripci" + "\u00f3" + "n del
Libro"},
{"Visitor", "Es visitanten" + "\u00fa" + "mero "},
{"What", "Qu" + "\u00e9" + " libros leemos"},
```

```
{"Talk", " describe como componentes de software de web pueden
transformar la manera en que desrrollamos aplicaciones para el
web. Este libro es obligatorio para cualquier programador de
respeto!"},
{"Start", "Empezar a Comprar"},
```

Character Encoding

A *character encoding* maps a character set to units of a specific width, and defines byte serialization and ordering rules. Many character sets have more than one encoding. For example, Java programs can represent Japanese character sets using the EUC-JP or Shift-JIS encodings, among others. Each encoding has rules for representing and serializing a character set.

The ISO 8859 series defines thirteen character encodings that can represent texts in dozens of languages. Each ISO 8859 character encoding may have up to 256 characters. ISO 8859-1 (Latin-1) comprises the ASCII character set, characters with diacritics (accents, diaereses, cedillas, circumflexes, and so on), and additional symbols.

UTF-8 (Unicode Transformation Format, 8 bit form) is a variable-width character encoding that encodes 16-bit Unicode characters as one to four bytes. A byte in UTF-8 is equivalent to 7-bit ASCII if its high-order bit is zero; otherwise, the character comprises a variable number of bytes.

UTF-8 is compatible with the majority of existing Web content and provides access to the Unicode character set. Current versions of browsers and email clients support UTF-8. In addition, many new Web standards specify UTF-8 as their character encoding. For example, UTF-8 is one of the two required encodings for XML documents (the other is UTF-16).

See Appendix F for more information on character encodings in the Java 2 platform.

Web components usually use `PrintWriter` to produce responses, which automatically encodes using ISO 8859-1. Servlets may also output binary data with `OutputStream` classes, which perform no encoding. An application that uses a character set that cannot use the default encoding must explicitly set a different encoding.

For Web components, three encodings must be considered:

- Request
- Page (JSP pages)
- Response

Request Encoding

The *request encoding* is the character encoding in which parameters in an incoming request are interpreted. Currently, many browsers do not send a request encoding qualifier with the `Content-Type` header. In such cases, a Web container will use the default encoding—ISO-8859-1—to parse request data.

If the client hasn't set character encoding and the request data is encoded with a different encoding than the default, the data won't be interpreted correctly. To remedy this situation, you can use the `ServletRequest.setCharacterEncoding(String enc)` method to override the character encoding supplied by the container. This method must be called prior to reading request parameters or reading input using `getReader`. To control the request encoding from JSP pages, you can use the JSTL `fmt:requestEncoding` tag.

This method must be called prior to parsing any request parameters or reading any input from the request. Calling this method once data has been read will not affect the encoding.

Page Encoding

For JSP pages, the *page encoding* is the character encoding in which the file is encoded.

For JSP pages in standard syntax, the page encoding is determined from the following sources:

- The Page Encoding value of a JSP property group (see Setting Properties for Groups of JSP Pages, page 677) whose URL pattern matches the page.
- The `pageEncoding` attribute of the `page` directive of the page. It is a translation-time error to name different encodings in the `pageEncoding` attribute of the page directive of a JSP page and in a JSP property group.
- The `CHARSET` value of the `contentType` attribute of the `page` directive.

If none of the above is provided, ISO-8859-1 is used as the default page encoding.

For JSP pages in XML syntax (JSP documents), the page encoding is determined as described in section 4.3.3 and appendix F.1 of the XMLspecification.

The `pageEncoding` and `contentType` attributes determine the page character encoding of only the file that physically contains the `page` directive. A Web container raises a translation-time error if an unsupported page encoding is specified.

Response Encoding

The *response encoding* is the character encoding of the textual response generated from a Web component. The response encoding must be set appropriately so that the characters are rendered correctly for a given locale. A Web container sets an initial response encoding for a JSP page from the following sources:

- The `CHARSET` value of the `contentType` attribute of the `page` directive.
- The encoding specified by the `pageEncoding` attribute of the `page` directive
- The Page Encoding value of a JSP property group whose URL pattern matches the page.

If none of the above is provided, ISO-8859-1 is used as the default response encoding.

The `setCharacterEncoding`, `setContentType`, and `setLocale` methods can be called repeatedly to change the character encoding. Calls made after the servlet response's `getWriter` method has been called or after the response is committed have no effect on the character encoding. Data is sent to the response stream on buffer flushes for buffered pages, or on encountering the first content on unbuffered pages.

Calls to `setContentType` set the character encoding only if the given content type string provides a value for the charset attribute. Calls to `setLocale` set the character encoding only if neither `setCharacterEncoding` nor `setContentType` has set the character encoding before. To control the response encoding from JSP pages, you can use the JSTL `fmt.setLocale` tag.

To obtain the character encoding for a locale, the `setLocale` method checks the locale encoding mapping for the Web application. For example, to map Japanese to the Japanese specific encoding `Shift_JIS`, add the following element to the Web application deployment descriptor:

```
<locale-encoding-mapping-list>
  <locale-encoding-mapping>
  <locale>ja</locale>
  <encoding>Shift_JIS</encoding>
  </locale-encoding-mapping>
</locale-encoding-mapping-list>
```

If a mapping is not set for the Web application, `setLocale` uses a J2EE 1.4 Application Server mapping.

The first application in Chapter 16 allows a user to choose an English string representation of a locale from all the locales available to the Java 2 platform and then outputs a date localized for that locale. To ensure that the characters in the date can be rendered correctly for a wide variety of character sets, the JSP page that generates the date sets the response encoding to UTF-8 with the following directive:

```
<%@ page contentType="text/html; charset=UTF-8" %>
```

Further Information

For a detailed discussion on internationalizing Web applications, see the Java BluePrints for the Enterprise:

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

24

Security

THE 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. 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.

Security 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. 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:

- <security-role>

The `<security-role>` element represents which roles from a defined group for the realm are authorized to access this Web Resource Collection. Security roles are discussed in Realms, Users, Groups, and Roles, page 952.

- `<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 Specifying Security Constraints, page 955.

- `<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 Using Login Authentication, page 958.

These elements of the deployment descriptor are entered directly into the `web.xml` file. If, for example, we were to create a deployment descriptor for a simple application that implements security, the `web.xml` file might look something like this example from Section SRV.13.5.2, *An Example of Security*, from the Java Servlet Specification, version 2.4:

```
<?xml version="1.0"encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version=?2.4 ?>
  <display-name>A Secure Application</display-name>
  <servlet>
    <servlet-name>catalog</servlet-name>
      <servlet-class>com.mycorp.CatalogServlet
      </servlet-class>
    <init-param>
      <param-name>catalog</param-name>
      <param-value>Spring</param-value>
    </init-param>
    <security-role-ref>
      <role-name>MGR</role-name>
      <!--role name used in code -->
      <role-link>manager</role-link>
    </security-role-ref>
  </servlet>
  <security-role>
    <role-name>manager</role-name>
```

```
    </security-role>
    <servlet-mapping>
      <servlet-name>catalog</servlet-name>
      <url-pattern>/catalog/*</url-pattern>
    </servlet-mapping>

    <!-- SECURITY CONSTRAINT -->
    <security-constraint>
      <web-resource-collection>
        <web-resource-name>SalesInfo</web-resource-name>
        <url-pattern>/salesinfo/*</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 AUTHENTICATION -->
    <login-config>
      <auth-method>BASIC</auth-method>
    </login-config>

    <!-- SECURITY ROLES -->
    <security-role>
      <role-name>manager</role-name>
    </security-role>

  </web-app>
```

Configuring authorized users and configuring the server to use SSL is addressed not in the application's deployment descriptor, but in the Web server's configuration files. In Tomcat, the user configuration is done in the `<JWSDP_HOME>/conf/tomcat-users.xml` file and the SSL configuration is done in the `<JWSDP_HOME>/conf/server.xml` file. Configuring SSL is discussed in Installing and Configuring SSL Support, page 970. Configuring authorized users is discussed in Setting up Security Roles, page 952 and Using Programmatic Security in the Web Tier, page 969.

Realms, 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 name 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's authentication service includes and interacts with the following components:

- *Realm* - For a Web application, 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).
- *User* - An individual (or application program) identity that has been defined in the. In a Web application, a user can have a set of *roles* associated with that identity, which entitles them to access all resources protected by those roles. Users can be associated with a group.
- *Group* - A set of authenticated *users*, classified by common traits, defined in. In most cases for Web applications, you will map users directly to roles and have no need to define a group.
- *Role* - An abstract name for the permission to access a particular set of resources in an application. 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.

Setting up 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 security role on the server that can be used by many Web services applications, you set up the users and roles that are defined for the server using `admintool` or by entering the information directly into `<JWSDP_HOME>/conf/tomcat-users.xml`. For information on setting up users and roles using `admintool`, see Administering Roles, Groups, and Users (page 1081).

To authorize one of the roles set up on the server to access a particular application, you list the authorized security roles in the application's deployment descriptor, `web.xml`.

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.

1. Select or open a Web application deployment descriptor, for example, `<INSTALL>/jwstutorial13/examples/security/formbasedauth/web/WEB-INF/web.xml`.
2. Add or modify the security constraint so that it contains the same elements as the one shown below. In this example, the role of `admin` is authorized to access this application, and is assigned a security role.

```
  <!-- 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>
<login-config>
    <auth-method>BASIC</auth-method>
</login-config>

  <!-- SECURITY ROLES -->
  <security-role>
    <description>the administrator role</description<
```

```
    <role-name>admin</role-name>
  </security-role>
```

3. Make sure that the `<role-name>` that you specify in the deployment descriptor has a corresponding entry in your server-specific file that contains the list of users and their assigned roles. 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>
  <role rolename="customer" description="Customer of Java
Web
    Service"/>
  <role rolename="manager"/>
  <role rolename="admin"/>
  <user username="your_name" password="your_password"
    roles="admin,manager"/>
  <user username="Anil" password="13345" fullName=""
    roles="customer"/>
</tomcat-users>
```

4. Add any necessary security code to the client. One example is shown in *<INSTALL>*`/jwstutorial13/examples/security/formbasedauth/web/index.jsp`.

Managing Roles and Users

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, the roles that have been defined for this server, and any users or roles you added after installation. The user name defined during installation is initially associated with the predefined roles of `admin` and `manager`. 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. We recommend that you use `admintool` in order to maintain the integrity of the users file.

Initially, the `tomcat-users.xml` file looks like this:

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

When you add roles and users using `admintool`, a GUI tool that enables you to make changes to the running Tomcat server, the file `<JWSDP_HOME>/conf/tomcat-users.xml` is updated as the changes are made in `admintool`. See Administering Roles, Groups, and Users, page 1081 for information on adding users and roles using `admintool`.

Specifying Security Constraints

You protect Web resources by specifying a security constraint. A *security constraint* determines who is authorized to access a *Web resource collection*, which is a list of URL patterns and HTTP methods that describe a set of resources to be protected. Security constraints are defined in a deployment descriptor.

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.

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.

The following items are defined within `<security-constraint>` tags in an application deployment descriptor:

- Security constraint—used to define the access privileges to a collection of resources using their URL mapping.
- Web resource collection—a list of URL patterns (the part of a URL *after* the host name and port which you want to constrain) and HTTP methods (the methods within the files that match the URL pattern which you want to constrain (for example, `POST`, `GET`)) that describe a set of resources to be protected.
- Authorized security role—the role that is authorized to access the parts of the application set up within this security constraint. Security roles are discussed in Setting up Security Roles, page 952.
- Guarantees on how the data will be transported between client and server —the choices include `NONE`, `INTEGRAL`, and `CONFIDENTIAL`. These options are discussed in Specifying a Secure Connection, page 957.

This is an example of a security constraint from the example application `<`*`INSTALL`*`>/jwstutorial13/examples/security/mutualauth/web.xml`:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>SecureHello</web-resource-name>
    <url-pattern>/hello</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
```

```
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
```

Specifying a Secure Connection

When the login authentication method in the `<login-config>` tags is set to `BASIC` or `FORM`, passwords are not protected, meaning that passwords sent between a client and a server on a non-protected session can be viewed and intercepted by third parties.

To configure HTTP basic or form-based authentication over SSL, specify `CONFIDENTIAL` or `INTEGRAL` 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>user</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.

If the default configuration of your Web server does not support SSL, you must configure it with an SSL connector to make this work. By default, Tomcat is not configured with an SSL Connector. To set up an SSL connector, see Installing and Configuring SSL Support (page 970).

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 Login Authentication

The `<login-config>` element in the application deployment descriptor specifies how the user is prompted to login in. If this element is present and contains a value other than `NONE`, the user must be authenticated before it can access any resource that is constrained by a `<security-constraint>`.

When you try to access a protected Web resource, the Web container activates the authentication mechanism that has been configured for that resource in the deployment descriptor (`web.xml`) between `<login-config>` elements within `<auth-method>` tags, like this:

```
<login-config>
  <auth-method>BASIC</auth-method>
</login-config>
```

The following choices are valid options for the authentication methods 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. HTTP basic authentication is not particularly secure. 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. An example application that uses HTTP Basic Authentication in a JAX-RPC service is described in Example: Basic Authentication with JAX-RPC (page 983).

- `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.

  Form-based authentication is not particularly secure. In form-based authentication, the content of the user dialog is sent as plain text, and the target server is not authenticated. This form of authentication 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. An example application using form-based authentication is included in the tutorial and is discussed in Example: Using Form-Based Authentication (page 960).

- `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 one another 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 970) and set up the public key certificate (see Setting Up Digital Certificates, page 971). An example application that uses `CLIENT-CERT` authentication is discussed in Example: Client-Certificate Authentication over HTTP/SSL with JAX-RPC (page 988).

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 above.
- 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.

Example: Using Form-Based Authentication

In this section, we discuss how to add form-based authentication to a basic JSP page. With *form-based authentication* (`<auth-method>FORM</auth-method>`), you can customize the login screen and error pages that are presented to the Web client for authentication of their user name and password. If the topic of authentication is new to you, please refer to the section titled Using Login Authentication (page 958).

The example application discussed in this tutorial can be found in `<INSTALL>/jwstutorial13/examples/security/formbasedauth/`. In general, the following steps are necessary to add form-based authentication to a Web client. In the example application included with this tutorial, most of these steps have been completed for you and are listed here expressly for the purpose of listing what needs to be done should you wish to create a similar application outside of this tutorial.

- Add the role name to the `tomcat-users.xml` file and authorize one of the users to assume this role. This example uses the previously unspecified role of `loginUser`, so you must add this role prior to starting Tomcat. See Adding Authorized Roles and Users (page 961) for more information on needed modifications.
- Edit the `build.properties` files. The `build.properties` file needs to be modified because the properties in this file are specific to your installation

of the Java WSDP and Java WSDP Tutorial. See Building and Running the Examples (page xxv) for information on which properties need to be set.

- Create the Web client. For this example, the Web client, a very simple JSP page, is already created. The client is discussed in Creating a Web Client for Form-Based Authentication (page 962).
- Create the login form and login error form pages. For this example, these files are already created. These pages are discussed in Creating the Login Form and Error Page (page 963).
- Add the appropriate security elements to the `web.xml` deployment descriptor. For this example, these have been added. See Specifying Security Elements for Form-Based Authentication (page 963) for further description of these elements.
- Build, install, and run the Web application (see Building, Installing, and Running the Form-Based Authentication Example, page 965). You will use the `Ant` tool to compile and install the example application.

Adding Authorized Roles and Users

This example application authorizes a the role of `loginUser` that is not already authorized for Tomcat. To add this role so that it is recognized by Tomcat, you add the information to the `tomcat-users.xml` file either by hand or using `admintool`. Information on adding usersand rolesusing `admintool` is discussed inAdministering Roles, Groups, and Users, page 1081. This section describes adding the information directly into the `tomcat-users.xml` file.

When Tomcatis started, it reads the settings in the `tomcat-users.xml` file. When a constrained resource is accessed, Tomcatverifies that the user name and password are authorized to access that resource before granting access to the requestor. The roles that are authorized to access a resource are specified in the security constraint in the deployment descriptor for this application. Because these values are read when Tomcat is started, you must stop Tomcat, make the changes, then restart Tomcat for it to recognize the new information.

1. Stop Tomcat. To do this,
    - On Unix, type the following command in a terminal window.

    ```
    <JWSDP_HOME>/bin/shutdown.sh
    ```

    - On Microsoft Windows, select Start→Programs→Java Web Services Developer Pack→Stop Tomcat.

Documentation for Tomcat can be found at `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/index.html`.

2. Open the file `<JWSDP_HOME>/conf/tomcat-users.xml` in a text editor. The file should contain at the very least the user name for the installer of the Java WSDP, the password specified by that user during installation, and the roles of `admin` and `manager`. Add the new role of `loginUser` to this file, and authorize at least one of the users to assume this role. The completed file should look like this, with the information that needs to be added highlighted in **bold** type:

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

3. Restart Tomcat as described in ***.

Creating a Web Client for Form-Based Authentication

The Web client is a standard JSP page. None of the code that adds form-based authentication to the example is included in the Web client. The information that adds the form-based authentication to this example is specified in the deployment descriptor. The code for the JSP page used in this example, `formbased-auth/web/index.jsp`, is listed below. The running application is shown in Figure 24–2.

```
<html>
<head><title>Hello</title></head>
<body bgcolor="white">

<img src="duke.waving.gif">
<h2>My name is Duke.</h2>
<h2><font color="black">Hello,
   ${pageContext.request.userPrincipal.name}!</font></h2>
</body>
</html>
```

Creating the Login Form and Error Page

The deployment descriptor specifies the JSP page that contains the form to be used to obtain the user name and password in order to verify that access to the client is authorized to that user. If login authentication fails, the error page is displayed in place of the requested page.

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 Java Servlet 2.4 Specification for more information on these requirements). 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>
```

The full code for the login page used in this example can be found at `<INSTALL>/jwstutorial13/examples/security/formbasedauth/web/logon.jsp`. An example of the running login form page is shown in Figure 24–1.

The login error page is displayed if a user name and password combination that is not authorized to access the protected URI is entered on the login page. For this example, the login error page can be found at `<INSTALL>/jwstutorial13/examples/security/formbasedauth/web/logonError.jsp`. The code for this page is displayed below:

```
<html>
<head>
<title>
 Login Error
</title>
</head>
<c:url var="url" value="/logon.jsp"/>
<p><a href="${url}">Try again.</a></p>
</html>
```

Specifying Security Elements for Form-Based Authentication

The following sample code shows the deployment descriptor used in this example of form-based login authentication. For this example application, the deployment descriptor can be found in `<INSTALL>/jwstutorial13/examples/security/formbasedauth/web/WEB-INF/web.xml`. This example contains the following elements that are necessary for this application to run properly.

- The `<security-constraint>` element is used to define the access privileges to a collection of resources using their URL mapping.
- The `<web-resource-collection>` element is used to identify a subset of the resources within a Web application to which a security constraint applies. In this example, by specifying `<url-pattern>/</url-pattern>`, we are specifying that all resources in this application are protected.
- The `<auth-constraint>` element indicates the user roles that should be permitted access to this resource collection. In this example, it is users assigned the role of `loginUser`. If no role name is provided, no user is allowed to access the portion of the Web application described by the security constraint.
- The `<login-config>` element is used to configure the authentication method that should be used and the attributes needed by the form login mechanism. 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 `<form-error-page>` element requires a URI of a Web resource relative to the document root that send a response when authentication has failed.

The following code is the deployment descriptor for the form-based authentication example:

```
<!-- FORM-BASED LOGIN AUTHENTICATION EXAMPLE -->
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://
java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <display-name>hello</display-name>
  <servlet>
    <display-name>index</display-name>
    <servlet-name>index</servlet-name>
    <jsp-file>/index.jsp</jsp-file>
  </servlet>
  <security-constraint>
    <display-name>SecurityConstraint</display-name>
    <web-resource-collection>
      <web-resource-name>WRCollection</web-resource-name>
      <url-pattern>/</url-pattern>
    </web-resource-collection>
```

```
      <auth-constraint>
        <role-name>loginUser</role-name>
      </auth-constraint>
      <user-data-constraint>
        <transport-guarantee>NONE</transport-guarantee>
      </user-data-constraint>
    </security-constraint>
    <login-config>
      <auth-method>FORM</auth-method>
      <form-login-config>
        <form-login-page>/logon.jsp</form-login-page>
        <form-error-page>/logonError.jsp</form-error-page>
      </form-login-config>
    </login-config>
    <security-role>
      <role-name>loginUser</role-name>
    </security-role>
  </web-app>
```

Building, Installing, and Running the Form-Based Authentication Example

To build, install, and run the `security/formbasedauth` example, which uses form-based authentication, follow these steps.

Building the Form-Based Authentication Example

1. Follow the instructions in Building and Running the Examples (page xxv).
2. Follow the instructions in Adding Authorized Roles and Users (page 961).
3. Go to the `<INSTALL>/jwstutorial13/examples/security/formbased-auth/` directory.
4. Build the Web application by entering the following at the terminal window or command prompt in the `formbasedauth/` directory (this and the following steps that use `Ant` assume that you have the executable for `Ant` in your path: if not, you will need to provide the fully-qualified path to the `Ant` executable). This command runs the `Ant` target named `build` in the `build.xml` file. The build target compiles any Java files in the application and copies Web components to the appropriate directories for deployment.

   ```
   ant build
   ```

Installing the Web Application

1. Start Tomcat. To do this,
   - On Unix, type the following command in a terminal window.

     ```
     <JWSDP_HOME>/bin/startup.sh
     ```

   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.

   - On Microsoft Windows, select Start→Programs→Java Web Services Developer Pack→Start Tomcat.

2. Install the Web application by entering the following at the terminal window or command prompt in the `/formbasedauth` directory:

   ```
   ant install
   ```

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 Java WSDP index page displays, you may continue. If the index page does not load immediately, wait up to several minutes and then retry. If, after several minutes, the index page does not display, refer to the troubleshooting tips in “Unable to Locate the Server localhost:8080” Error, page 67.

Documentation for Tomcat can be found at `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/index.html`. Errors commonly encountered when starting Tomcat are discussed in Errors Starting Tomcat, page 67.

Running the Web Application

Run the Web client by entering the following URL in your Web browser:

```
http://localhost:8080/formbasedauth
```

The login form displays in the browser, as shown in Figure 24–1. Enter a user name and password combination that corresponds to the role of `loginUser`, then click the Submit button.

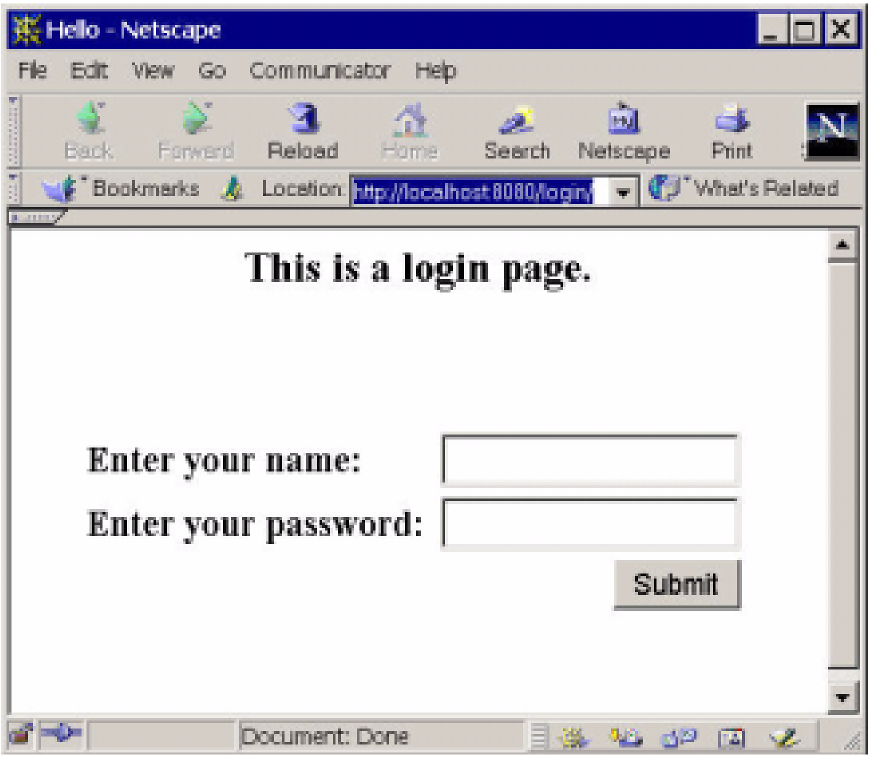


Figure 24–1 Form-Based Login Page

If you entered `Debbie` as the name, and there is a user defined in `tomcat-users.xml` with the user name of `Debbie` that also matches the password you entered and which is assigned to the role of `loginUser`, the display will appear as in Figure 24–2 after you click the Submit button.

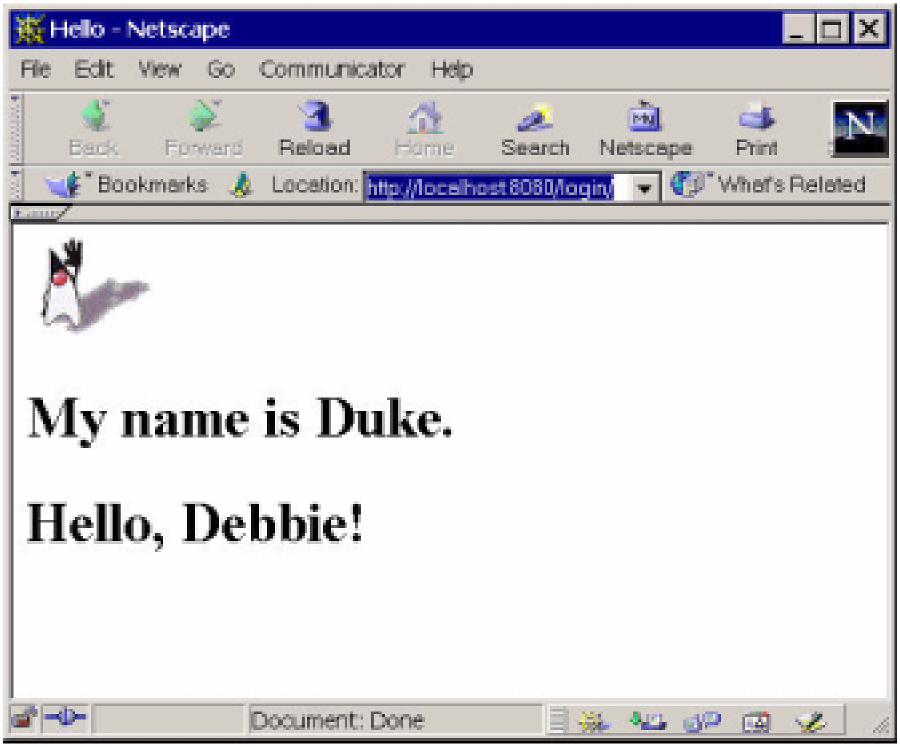


Figure 24–2 The Running Form-Based Authentication Example

Using Authentication with SSL

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. To configure HTTP basic or form-based authentication over SSL, specify `CONFIDENTIAL` or `INTEGRAL` as the Network Security Requirement on the WAR's Security page in `deploytool` within the `<transport-guarantee>` elements. Read the section Specifying a Secure Connection (page 957) for more information.

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.

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 JSP pages 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)` 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.

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, as shown here:

```
<servlet>
   ...
  <security-role-ref>
    <role-name>cust</role-name>
    <role-link>bankCustomer</role-link>
  </security-role-ref>
   ...
</servlet>

<security-role>
   <role-name>bankCustomer</role-name>
</security-role>
```

As discussed in Setting up Security Roles, page 952, there also must be a corresponding entry in the Web server users file, which is *<JWSDP_HOME>*/conf/tomcat-users.xml for Tomcat. The role of admin is defined by default in the file, as shown below:

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

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.

- A server certificate keystore (see Setting Up Digital Certificates, page 971).
- An HTTPS connector (see Configuring the SSL Connector, page 977).

To verify that SSL support is enabled, see Verifying SSL Support (page 978).

Setting Up Digital Certificates

In order to use 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 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 J2SE. 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 `keytool` and public key cryptography, read the `keytool` documentation at the following URL:

```
http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/key-
tool.html
```

Creating a Server Certificate

You can use `keytool` to generate certificates. The `keytool` stores the keys and certificates in a file termed a *keystore*. A keystore is a repository of certificates used for identifying a client or a server. Typically, a keystore contains one client or one server's identity. The default keystore implementation implements the keystore as a file. It protects private keys with a password.

The keystores are created in the directory from which you run `keytool`. This can be the directory where the application resides or it can be a directory common to many applications.

To create a server certificate,

1. Create the keystore. If you create a server certificate, you will reference it from the Tomcat deployment descriptor so that you can use SSL.
2. Export the certificate from the keystore.
3. Sign the certificate.
4. Import the certificate into a trust-store. A trust-store is a repository of certificates used for verifying the certificates. A trust-store typically contains more than one certificate. An example using a trust-store for SSL-based mutual authentication is discussed in Example: Client-Certificate Authentication over HTTP/SSL with JAX-RPC (page 988).

Run `keytool` to generate the server keystore, which we will name `server-keystore.jks`. This step uses the alias `server-alias` to generate a new public/private key pair and wrap the public key into a self-signed certificate inside `server-keystore.jks`. The key pair is generated using an algorithm of type RSA, with a default password of `changeit`. For more information on `keytool` options, see its online help at `http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/keytool.html`.

From the directory in which you want to create the keystore, run `keytool` with the following parameters. 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`. The host specified in the keystore must match the host identified in the `host` variable specified in the `<`*`INSTALL`*`>/jwstutorial13/examples/common/build.properties`.

1. Generate the server certificate.

```
<JAVA_HOME>\bin\keytool -genkey -alias server-alias
-keyalg RSA -keypass changeit -storepass changeit
-keystore keystore.jks
```

2. Export the generated server certificate in `keystore.jks` into the file `server.cer`.

```
<JAVA_HOME>\bin\keytool -export -alias server-alias
-storepass changeit -file server.cer -keystore keystore.jks
```

3. If you want to have the certificate signed by a CA, read Signing Digital Certificates (page 975) for more information.
4. To create the trust-store file `cacerts.jks` and add the server certificate to the trust-store, run `keytool` from the directory where you created the keystore and server certificate with the following parameters:

```
<JAVA_HOME>\bin\keytool -import -v -trustcacerts
-alias server-alias -file server.cer
-keystore cacerts.jks -keypass changeit
-storepass changeit
```

Information on the certificate, such as that shown below will display.

```
<INSTALL>/jwstutorial13/examples/gs 60% keytool -import
-v -trustcacerts -alias server-alias -file server.cer
-keystore cacerts.jks -keypass changeit -storepass changeit
Owner: CN=localhost, OU=Sun Micro, O=Docs, L=Santa Clara,
ST=CA, C=US
Issuer: CN=localhost, OU=Sun Micro, O=Docs, L=Santa Clara,
ST=CA, C=US
Serial number: 3e932169
Valid from: Tue Apr 08
Certificate fingerprints:
MD5: 52:9F:49:68:ED:78:6F:39:87:F3:98:B3:6A:6B:0F:90
SHA1: EE:2E:2A:A6:9E:03:9A:3A:1C:17:4A:28:5E:97:20:78:3F:
Trust this certificate? [no]:
```

5. Enter `yes`, then strike the `Enter` or `Return` key. The following information displays:

```
Certificate was added to keystore
[Saving cacerts.jks]
```

Signing Digital Certificates

Once you've created a digital certificate, you will want to have it signed by its owner. Once the digital certificate is cryptographically signed by its owner, it 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 the self-signed certificate.

Creating a Client Certificate for Mutual 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. From the directory where you want to create the client certificate, run `keytool` as outlined below. When you press Enter, `keytool` prompts you to enter the server name, organizational unit, organization, locality, state, and country code.

Note: 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`. The host specified in the keystore must match the host identified in the `host` variable specified in the `<INSTALL>/jwstutorial13/examples/common/build.properties` file. If this example is to verify mutual authentication, and you receive a runtime error stating that the HTTPS host name is wrong, recreate the client certificate, being sure to use the same host name that you will use when running the example. For example, if your machine name is `duke`, then enter `duke` as the certificate CN or when prompted for first and last names. When accessing the application, enter an URL that points to the same location, for example, `https://duke:1043/mutualauth/hello`. This is necessary because during SSL handshake, the server verifies the client certificate by comparing the certificate name and the host name from which it originates.

To create a keystore named `client-keystore.jks` that contains a client certificate named `client.cer`, follow these steps:

1. Generate the client certificate.

```
<JAVA_HOME>\bin\keytool -genkey -alias client-alias -keyalg
RSA -keypass changeit -storepass changeit
-keystore keystore.jks
```

2. Export the generated client certificate into the file `client.cer`.

```
<JAVA_HOME>\bin\keytool -export -alias client-alias
-storepass changeit -file client.cer -keystore keystore.jks
```

3. Add the certificate to the trust-store file `cacerts.jks`. Run `keytool` from the directory where you created the keystore and client certificate with the following parameters:

```
<JAVA_HOME>\bin\keytool -import -v -trustcacerts
-alias client-alias -file client.cer
-keystore cacerts.jks -keypass changeit
-storepass 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
```

For an example application that uses mutual authentication, see Example: Client-Certificate Authentication over HTTP/SSL with JAX-RPC (page 988). For information on verifying that mutual authentication is running, see Verifying Mutual Authentication is Running (page 980).

Miscellaneous Commands for Certificates

- To check the contents of a keystore that contains a certificate with an alias `server-alias`:

  ```
  keytool -list -keystore keystore.jks -alias server-alias -v
  ```

- To check the contents of the `cacerts` file:

  ```
  keytool -list -keystore cacerts.jks
  ```

Configuring the SSL Connector

Note: An SSL Connector needs to be configured for Tomcat.

Depending on your Web Server, an SSL HTTPS Connector may or may not be enabled. If you are using Tomcat, its SSL connector needs to be configured, and this section describes how to do so. 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. Also, in order to use SSL you must add information about where to locate the keystore file and what its password is to the deployment descriptor. To enable the SSL connector for Tomcat and add the information about the keystore, follow these steps:

1. Shut down the server. If you don't know how to do this, refer to the section Shutting Down Tomcat, page 65.
2. Enable or add an SSL HTTPS Connector to your Web server using either of these two methods:
   - Add the `Connector` using `admintool`. See the documentation for `admintool` for more information on how to do this.
   - Add a `Connector` element for an SSL connector to the server's deployment descriptor.

   To enable the Connector element for Tomcat, find the following section in the file *<JWSDP_HOME>*`/conf/server.xml`, remove the comment tags surrounding it, and add the code in **bold** to specify the keystore information.

   ```
   <!-- Define a SSL Coyote HTTP/1.1 Connector on port 8443 -->
   <!-- REMOVE the comment tag on the next line-->
   <!--
   <!-- Define a SSL Coyote HTTP/1.1 Connector on port 8443 -->
   ```

```
<Connector
    className="org.apache.coyote.tomcat5.CoyoteConnector"
    port="8443" minProcessors="5" maxProcessors="75"
    enableLookups="true" disableUploadTimeout="true"
    acceptCount="100" debug="0" scheme="https"
    secure="true">
<Factory className=
    "org.apache.coyote.tomcat5.CoyoteServerSocketFactory"
   keystoreFile="<path_to_keystore>/keystore.jks"
   keystorePass="changeit"
    clientAuth="false" protocol="TLS" />
</Connector>
<!-- REMOVE the comment tag on the line below-->
-->
```

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 `localhost` in this example assumes you are running the example on your local machine as part of the development process. The `8443` in this example is the secure port that was specified where the SSL Connector was created in Configuring the SSL Connector (page 977). If you are using a different server or port, modify this value accordingly.

The first time a user loads this application, the New Site Certificate or Security Alert dialog displays. Select Next to move through the series of dialogs, select Finish when you reach the last dialog. The certificates will only display the first time. When you accept the certificates, subsequent hits to this site assume that you still trust the content.

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.

Enabling Mutual Authentication Over SSL

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. Set the `clientAuth` method to `true` in the Web server configuration file 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-CERT` authentication. If you turn `clientAuth` on, it is on all of the time, which can be a tremendous performance hit. However, if you regulate it through the application (via the `CLIENT-CERT` authentication requirement), the check is only performed when the application requires client authentication. No matter which way you choose, you must enter the keystore location and password in the Web server configuration file to enable SSL, as discussed in Configuring the SSL Connector (page 977).

The two ways to enable mutual authentication over SSL are:

- Configure the SSL Socket Factory in the *<JWSDP_HOME>*/conf/server.xml file as shown. As with all changes to the Web server configuration file, you must stop and restart the Web server for this change to become effective.

```
<Connector
     className="org.apache.coyote.tomcat5.CoyoteConnector"
     port="8443" minProcessors="5" maxProcessors="75"
     enableLookups="true" disableUploadTimeout="true"
     acceptCount="100" debug="0" scheme="https"
     secure="true">
<Factory>
     className=

"org.apache.coyote.tomcat5.CoyoteServerSocketFactory"
   keystoreFile="<path_to_keystore>/keystore.jks"
   keystorePass="changeit" clientAuth="true"
     protocol="TLS"
   debug="0" />
</Connector>
```

  When you enable client authentication by setting the clientAuth property to "true", client authentication will be required for all the requests going through the specified SSL port.

- Set the method of authentication in the web.xml file to CLIENT-CERT, as shown below. By enabling client authentication in this way, client authentication is enabled only for a specific resource controlled by the security constraint. Setting client authentication in this way is discussed in Example: Client-Certificate Authentication over HTTP/SSL with JAX-RPC (page 988).

```
<login-config>
  <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

When client authentication is enabled in both ways mentioned above, client authentication will be performed twice.

Verifying Mutual Authentication is Running

You can verify that mutual authentication is working by obtaining debug messages. This should be done at the client end, and this example shows how to pass a system property in targets.xml so that targets.xml forks a client with javax.net.debug in its system properties, which could be added in a file such as *<INSTALL>*/jwstutorial13/examples/security/common/targets.xml.

To enable debug messages for SSL mutual authentication, pass the system property `javax.net.debug=ssl,handshake`, which will provide information on whether mutual authentication is working or not. The following example modifies the `run-mutualauth-client` target from the `<INSTALL>/jwstutorial13/examples/security/common/targets.xml` file by adding `sysproperty` as shown in **bold**:

```
<target name="run-mutualauth-client"
description="Runs a client with mutual authentication over
SSL">
  <java classname="${client.class}" fork="yes" >
    <arg line="${key.store} ${key.store.password}
          ${trust.store} ${trust.store.password}
          ${endpoint.address}" />
    <sysproperty key="javax.net.debug" value="ssl,
          handshake" />
    <sysproperty key="javax.net.ssl.keyStore"
          value="${key.store}" />
    <sysproperty key="java.net.ssl.keyStorePassword"
          value="${key.store.password}"/>
    <classpath refid="run.classpath" />
  </java>
</target>
```

XML and Web Services Security

XML and Web Services Security can include two use cases. These use cases include the following:

- Transport-Level Security, page 982, is where security is addressed by the transport layer. Adding security in this way is discussed in the following example sections:
  - Example: Basic Authentication with JAX-RPC, page 983
  - Example: Client-Certificate Authentication over HTTP/SSL with JAX-RPC, page 988.
- Message-Level Security, page 996, is where the security information is contained within the SOAP message, which allows security information to travel along with the message. This model enables message parts to be transported without intermediate nodes seeing or modifying the message. Adding security in this way is discussed in the following section:
  - Message-Level Security, page 996.

Transport-Level Security

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. There are several ways in which this can happen, the following ways are discussed in this section:

- A user authentication method can be defined for an application in its deployment descriptor. When a user authentication method is specified for an application, the Web container activates the specified authentication mechanism when you attempt to access a protected resource. The options for user authentication methods are discussed in Using Login Authentication, page 958. The example application discussed in Example: Basic Authentication with JAX-RPC (page 983) shows how to add basic authentication to a JAX-RPC application. The example discussed in Example: Client-Certificate Authentication over HTTP/SSL with JAX-RPC, page 988 shows how to add client-certificate, or mutual, authentication to a JAX-RPC application.
- A transport guarantee can be defined for an application in its deployment descriptor. Use this method to run over an SSL-protected session and ensure that all message content is protected for confidentiality. The options for transport guarantees are discussed in Specifying a Secure Connection, page 957. An example application that discusses running over an SSL-protected session is discussed in Example: Client-Certificate Authentication over HTTP/SSL with JAX-RPC (page 988).

  When running over an SSL-protected session, the server and client can authenticate one another and negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data.

  Secure Socket Layer (SSL) technology 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. For more information, see What is Secure Socket Layer Technology? (page 970).

  Digital certificates are necessary when running HTTP over SSL (HTTPS). The HTTPS service of most Web servers will not run unless a digital certificate has been installed. Use the procedure outlined in Setting Up Digital Certificates, page 971 to set up a digital certificate that can be used by your Web server to enable SSL.

Example: Basic Authentication with JAX-RPC

In this section, we discuss how to configure JAX-RPC-based Web service applications for HTTP basic authentication. With *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. If the topic of authentication is new to you, please refer to the section titled Using Login Authentication, page 958.

Note: The instructions in this section apply to the Java WSDP version 1.3.

For this tutorial, we begin with the example application in `<INSTALL>/jwstutorial13/examples/jaxrpc/staticstub/` and `<INSTALL>/jwstutorial13/examples/jaxrpc/helloservice/` and add user name/password authentication. The resulting application can be found in the directories `<INSTALL>/jwstutorial13/examples/security/basicauth/` and `<INSTALL>/jwstutorial13/examples/security/basicauthclient/`. In general, the following steps are necessary to add basic authentication to a JAX-RPC application. In the example application included with this tutorial, many of these steps have been completed for you and are listed here expressly for the purpose of listing what needs to be done should you wish to create a similar application outside of this tutorial.

- Add the appropriate security elements to the `web.xml` deployment descriptor. For this example, these have been added. Refer to Add Security Elements to the Deployment Descriptor, page 984 for further description of these elements.
- Edit the `build.properties` files. The `build.properties` file needs to be modified because the properties in this file are specific to your installationof the Java WSDP and The Java Web Services Tutorial. See Building and Running the Examples (page xxv) for information on which properties need to be set.
- Set security properties in the client code. For the example application, this step has been completed. The code for this example is shown in Set Security Properties in the Client Code (page 985).
- Build, deploy, and run the Web service (see Building, Deploying, and Running the Example for Basic Authentication, page 986). You will use the `Ant` tool to compile and run the example application.

Add Security Elements to the Deployment Descriptor

For HTTP basic authentication, the application deployment descriptor, `web.xml`, includes the information on who is authorized to access the application, which URL patterns and HTTP methods are protected, and what type of user authentication method this application uses. This information is added to the deployment descriptor inside <security-constraint>, <login-config>, and <security-role> elements. These security elements are discussed in more detail in Specifying Security Constraints, page 955 and in the Java Servlet Specification, which can be browsed or downloaded online at `http://java.sun.com/products/servlet/`. Code in **bold** is added to the deployment descriptor, `<INSTALL>/jwstutorial13/examples/security/basicauth/web.xml`, to enable HTTP basic authentication:

```
<?xml version="1.0" ?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://
java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <display-name>Basic Authentication Security</display-name>
  <session-config>
    <session-timeout>60</session-timeout>
  </session-config>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>SecureHello</web-resource-name>
      <url-pattern>/hello</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</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>
  <login-config>
    <auth-method>BASIC</auth-method>
  </login-config>
  <security-role>
    <role-name>admin</role-name>
  </security-role>
</web-app>
```

Note that the <role-name> element specifies `admin`, a role that has already been specified in the Tomcat user's file. For more information on defining and linking roles, see Setting up Security Roles, page 952.

Set Security Properties in the Client Code

The source code for the client is in the `HelloClient.java` file of the `<INSTALL>/jwstutorial13/examples/security/basicauthclient/src/` directory. For basic authentication, the client code must set `username` and `password` properties. The username and password properties correspond to the `admin` role, which includes the user name and password combination entered during installation and provided in the application deployment descriptor as an authorized role for secure transactions. (See Setting up Security Roles, page 952.)

The client sets the aforementioned security properties as shown in the code below. The code in **bold** is the code that had been added from the original version of the `jaxrpc/staticstub` example application.

```
package basicauthclient;

import javax.xml.rpc.Stub;

public class HelloClient {

    public static void main(String[] args) {

        if (args.length !=3) {
        System.out.println("HelloClient Error: Wrong
           number of runtime arguments!");
        System.exit(1);
        }

        String username=args[0];
        String password=args[1];
        String endpointAddress=args[2];

       // print to display for verification purposes
```

```
            System.out.println("username: " + username);
            System.out.println("password: " + password);
            System.out.println("Endpoint address = " +
               endpointAddress);


        try {
           Stub stub = createProxy();
              stub._setProperty(
                 javax.xml.rpc.Stub.USERNAME_PROPERTY,
                    username);
              stub._setProperty(
                 javax.xml.rpc.Stub.PASSWORD_PROPERTY,
                    password);
              stub._setProperty
                 (javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,
                 endpointAddress);

           HelloIF hello = (HelloIF)stub;
           System.out.println(hello.sayHello("Duke (secure)"));
            } catch (Exception ex) {
                ex.printStackTrace();
            }
       }

       private static Stub createProxy() {
          // Note: MyHelloService_Impl is implementation-specific.
            return (Stub)(new
                 MyHelloService_Impl().getHelloIFPort());
       }
   }
```

Building, Deploying, and Running the Example for Basic Authentication

To build, deploy, and run the `security/basicauth` example using basic authentication, follow these steps,

Build the Basic Authentication Example

1. Set up your system for running the tutorial examples if you haven't done so already by following the instructions in Building and Running the Examples (page xxv).

2. From a terminal window or command prompt, go to the `<INSTALL>/jwstutorial13/examples/security/basicauth/` directory.
3. Build the JAX-RPC service by entering the following at the terminal window or command prompt in the `basicauth/` directory (this and the following steps that use `Ant` assume that you have the executable for `Ant` in your path: if not, you will need to provide the fully-qualified path to the `Ant` executable). This command runs the `Ant` target named `build` in the `build.xml` file.

    ```
    ant build
    ```

4. Change to the `<INSTALL>/jwstutorial13/examples/security/basicauthclient/` directory. Build the JAX-RPC client by entering the following at the terminal window or command prompt:

    ```
    ant build
    ```

Deploy the Basic Authentication Example

5. Start Tomcat.
6. Deploy the JAX-RPC service by entering the following at the terminal window or command prompt in the `/basicauth` directory:

    ```
    ant deploy
    ```

Running the Basic Authentication Example

7. Run the JAX-RPC client by entering the following at the terminal window or command prompt in the `basicauthclient/` directory:

    ```
    ant run
    ```

The client should display the following output:

```
Buildfile: build.xml

run-secure-client:
   [java] username: your_name
   [java] password: your_pwd
   [java] Endpoint address = http://localhost:8080/basicauth-
jaxrpc/hello
   [java] Hello Duke (secure)

BUILD SUCCESSFUL
```

Example: Client-Certificate Authentication over HTTP/SSL with JAX-RPC

In this section, we discuss how to configure a simple JAX-RPC-based Web service application for client-certificate authentication over HTTP/SSL. *Client-certificate authentication* (`<auth-method>CLIENT-CERT</auth-method>`) uses HTTP over SSL, in which the server and, optionally, the client authenticate one another with Public Key Certificates. If the topic of authentication is new to you, please refer to the section titled Using Login Authentication (page 958).

This example application starts with the example application in `<INSTALL>/jwstutorial13/examples/jaxrpc/helloservice/` and adds both client and server authentication to the example. In SSL certificate-based basic authentication, the server presents its certificate to the client, and the client authenticates to the server by sending its user name and password. This type of authentication is sometimes called server authentication. Mutual authentication adds the dimension of client authentication. For mutual authentication, we need both the client's identity, as contained in a client certificate, and the server's identity, as contained in a server certificate inside a keystore file (`keystore.jks`), and we need both of these identities to be contained in a mutual trust-store (`cacerts.jks`) where they can be verified.

To add mutual authentication to the `<INSTALL>/jwstutorial13/examples/jaxrpc/helloservice/` example, we need to complete the following steps. In the example application included with this tutorial, many of these steps have been completed for you and are listed here expressly for the purpose of listing what needs to be done should you wish to create a similar application outside of this tutorial.

1. Create the appropriate certificates and keystores. For this example, the certificates and keystores are created for a generic `localhost` and are included with the example application. See the section Keystores and Trust-Stores in the Mutual Authentication Example (page 989) for a discussion of these files. If you are creating a different application, refer to the section Setting Up Digital Certificates (page 971) for more information on creating the keystores and certificates and importing the client and server identity into the trust-store.
2. Configure the SSL Connector, if necessary. For this release of Tomcat, the SSL connector needs to be configured before you can run the example application, and the location and password of the server keystore must be

specified in the `server.xml` file as well. Read the instructions on how to do this in the section Configuring the SSL Connector for Certificate Authentication (page 990).

3. Edit the `build.properties` files to add the location and password to the trust-store, and other properties, as appropriate. For a discussion of the modifications that need to be made to `build.properties`, see Modifying the Build Properties (page 991).
4. Set security properties in the client code. For the example application, this step has been completed. For a discussion of the security properties that have been set in `HelloClient`, see Setting Security Properties in the Client Code (page 991).
5. Add the appropriate security elements to the `web.xml` deployment descriptor. For this example, these have been added.The security elements are discussed in the section Enabling Mutual Authentication over SSL (page 993).
6. Build the client and server files, deploy the server, and run the client (see Build, Deploy, and Run the Mutual Authentication Example, page 994). You will use the `Ant` tool to compile and deploy the example application.

Keystores and Trust-Stores in the Mutual Authentication Example

In this example, the keystore file (`keystore.jks`) and the trust-store file (`cacerts.jks`) have already been created for a generic `localhost` and are included with the example applicationin the directory *<INSTALL>*/jwstutorial13/examples/security/mutualauth/. These files were created using the following steps, which are discussed in more detail in Setting Up Digital Certificates (page 971).

1. Create a server certificate in the file `keystore.jks`.
2. Export the certificate.
3. Import the certificate into the trust-store, `cacerts.jks`.
4. Create a client certificate in the client keystore.
5. Export the certificate.
6. Import the certificate into the trust-store, `cacerts.jks`.

Configuring the SSL Connector for Certificate Authentication

By default, the SSL Connector is not enabled for Tomcat for this release of the Java WSDP. To use the SSL Connector that comes pre-configured for Tomcat, you need to uncomment the section that includes the SSL connector, as discussed in Configuring the SSL Connector, page 977. In this same file, you must add the information on the location and password of the server's keystore file. The following code snippet is from the Tomcat Server Configuration file, which is located at `<JWSDP_HOME>/conf/server.xml`. First, stop the server if it is running. Open this file, remove the comment tags around the SSL Connector if you haven't done so already (highlighted in bold), and add the keystore information as shown in **bold** below. Be sure to enter the fully-qualified path to the keystore files.

```
<!-- Define a SSL Coyote HTTP/1.1 Connector on port 8443 -->
<!-- REMOVE the comment tag on the next line-->
<!--
<Connector
     className="org.apache.coyote.tomcat5.CoyoteConnector"
     port="8443" minProcessors="5" maxProcessors="75"
     enableLookups="true" disableUploadTimeout="true"
     acceptCount="100" debug="0" scheme="https"
     secure="true">
<Factory
     className=
        "org.apache.coyote.tomcat5.CoyoteServerSocketFactory"
     keystoreFile=
        "<INSTALL>/jwstutorial13/examples/
           security/mutualauth/keystore.jks"
     keystorePass="changeit"
     clientAuth="false" protocol="TLS" />
</Connector>
<!-- REMOVE the comment tag on the line below-->
-->
```

Restart the server, and it will recognize the secure port and keystore.

You might notice the `clientAuth` property in the `Factory` section. You would set this to `true` to enable client authentication for all traffic through this server. The different ways of authorizing client authentication are discussed in Enabling Mutual Authentication Over SSL, page 979.

However, the information on the location and password of the server's keystore file must be provided to the server. To do this, refer to Configuring the SSL Connector (page 977).

Modifying the Build Properties

To build and run the application with mutual authentication, we have set up the example so that some of the values are passed to the application from various `build.properties` files.

To run any of the examples, you need to modify the `build.properties` file located in the `<INSTALL>/jwstutorial13/examples/common/` directory to provide the location where the tutorial is installed and to specify your user name and password. If you need more information, see Building and Running the Examples (page xxv).

If you're not using Tomcat and its default settings, you should also verify that the `host` and `port` properties are set correctly. If you have modified the default host and/or port, you must also modify these settings in the `<INSTALL>/jwstutorial13/examples/security/mutualauthclient/SecureHello.wsdl` file.

For this example, the `build.properties` file that is specific to this application, `<INSTALL>/jwstutorial13/examples/security/common/build.properties`, has been modified for you. This file provides specific information about the JAX-RPC examples to the `Ant` targets we will be running later regarding the location of the keystore and trust-store files and their associated passwords.

Make sure that the following properties exist and are correctly defined.

```
trust.store=${tutorial.home}/examples/security/
  mutualauth/cacerts.jks
trust.store.password=changeit
key.store=${tutorial.home}/examples/security/mutualauth/
  keystore.jks
key.store.password=changeit
```

Setting Security Properties in the Client Code

The source code for the client is in the `HelloClient.java` file of the `<INSTALL>/jwstutorial13/examples/security/mutualauthclient/src/` directory. For mutual authentication, the client code must set several security-related properties. These values are passed into the client code when the `Ant` `build` and `run` tasks are executed.

- `trustStore`. The value of the `trustStore` property is the fully qualified name of the trust-store file: `<INSTALL>/jwstutorial13/examples/security/mutualauth/cacerts.jks`.
- `trustStorePassword`. The `trustStorePassword` property is the password of the trust-store. The default value of this password is `changeit`.
- `keyStore`. The value of the `keyStore` property is the fully qualified name of the keystore file: `<INSTALL>/jwstutorial13/examples/security/mutualauth/keystore.jks`.
- `keyStorePassword`. The `keyStorePassword` property is the password of the keystore. The default value of this password is `changeit`.
- `ENDPOINT_ADDRESS_PROPERTY`. The `endpointAddress` property sets the endpoint address that the stub uses to access the service.

The client sets the aforementioned security properties as shown in the code below. The code in **bold** is the code that had been added from the original version of the `jaxrpc/staticstub` example application.

```
package mutualauthclient;

import javax.xml.rpc.Stub;

public class HelloClient {

    public static void main(String[] args) {

        if (args.length !=5) {
        System.out.println("HelloClient Error: Need 5
          runtime arguments!");
        System.exit(1);
        }

        String keyStore=args[0];
        String keyStorePassword=args[1];
        String trustStore=args[2];
        String trustStorePassword=args[3];
        String endpointAddress=args[4];


       // print to display for verification purposes
        System.out.println("keystore: " + keyStore);
        System.out.println("keystorePassword: " +
          keyStorePassword);
```

```
        System.out.println("trustStore: " + trustStore);
        System.out.println("trustStorePassword: " +
          trustStorePassword);
        System.out.println("Endpoint address: " +
          endpointAddress);

    try {
       Stub stub = createProxy();
       System.setProperty("javax.net.ssl.keyStore",
          keyStore);
       System.setProperty("javax.net.ssl.keyStorePassword",
          keyStorePassword);
       System.setProperty("javax.net.ssl.trustStore",
          trustStore);
       System.setProperty("javax.net.ssl.trustStorePassword",
          trustStorePassword);
       stub._setProperty(
            javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,
               endpointAddress);

       HelloIF hello = (HelloIF)stub;
       System.out.println(hello.sayHello("Duke! (secure!"));
        } catch (Exception ex) {
           ex.printStackTrace();
        }
    }

    private static Stub createProxy() {
       // Note: MyHelloService_Impl is implementation-specific.
        return (Stub)(new
            MySecureHelloService_Impl().getHelloIFPort());
    }
}
```

Enabling Mutual Authentication over SSL

The two ways of implementing client authentication are discussed in Enabling Mutual Authentication Over SSL (page 979). You can set client authentication for all applications (by specifying this in the deployment descriptor for the server) or for just a single application (by specifying this in the deployment descriptor for the application). For this example, we are enabling client authentication for this application only, so we specify the login authentication method in the deployment descriptor, `web.xml`, as being`CLIENT-CERT` .

To view the application deployment descriptor, open the `web.xml` file located in the `<INSTALL>/jwstutorial13/examples/security/mutualauth` directory.

Security constraints were discussed in Specifying Security Constraints, page 955. The following section discusses the parts of the deployment descriptor that add client certification to this example:

```
<login-config>
  <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

For more information on `<login-config>` options, read Using Login Authentication (page 958).

The user authentication method specifies a client-certificate method of authentication in this example. For this authentication to run over SSL, we also need to specify which type of transport guarantee to use. For this example, we have chosen CONFIDENTIAL, which is specified in the web.xml file as follows:

```
<user-data-constraint>
  <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
```

For more information on this type of constraint, read Specifying a Secure Connection (page 957).

Build, Deploy, and Run the Mutual Authentication Example

To build, deploy, and run the JAX-RPC service example with mutual authentication, follow these steps.

Build the Mutual Authentication Example

To compile the application files and copy them to the correct directories, run the `Ant build` task. More information on what happens when the `build` task is called can be found in Building the Service (page 461).

1. If you haven't already done so, follow these steps for setting up the example.
   - Configuring the SSL Connector (page 977).
   - Building and Running the Examples (page xxv).
2. Go to the `<`*`INSTALL`*`>/jwstutorial13/examples/security/mutual-auth/` directory.

3. Build the JAX-RPC service by entering the following at the terminal window or command prompt in the `mutualauth/` directory (this and the following steps that use `Ant` assume that you have the executable for `Ant` in your path: if not, you will need to provide the fully-qualified path to the `Ant` executable):

   ```
   ant build
   ```

4. Change to the directory `<INSTALL>/jwstutorial13/examples/security/mutualauthclient/`.
5. Build the JAX-RPC client by entering the following at the terminal window or command prompt:

   ```
   ant build
   ```

Deploy the Mutual Authentication Example

1. Deploy the JAX-RPC service by entering the following at the terminal window or command prompt in the `/mutualauth` directory:

   ```
   ant deploy
   ```

Run the Mutual Authentication Example

Enter the following command from the `mutualauthclient/` directory at the terminal window or command prompt to run the JAX-RPC client:

```
ant  run
```

The client should display the following output:

```
Buildfile: build.xml

run-mutualauth-client:
    [java] keyStore: <INSTALL>/jwstutorial13/examples/secu-
rity/mutualauth/keystore.jks
    [java] keyStorePassword: changeit
    [java] trustStore: <INSTALL>/jwstutorial13/examples/secu-
rity/mutualauth/cacerts.jks
    [java] trustStorePassword: changeit
    [java] endpointAddress = https://localhost:8443/secure-
mutualauth/hello
    [java] Hello Duke (secure)

BUILD SUCCESSFUL
```

For information on verifying that mutual authentication is running, see Verifying Mutual Authentication is Running (page 980).

Message-Level Security

In message-level security, security information is contained within the SOAP message, which allows security information to travel along with the message. For example, a portion of the message may be signed by a sender and encrypted for a particular receiver. When the message is sent from the initial sender, it may pass through intermediate nodes before reaching its intended receiver. In this scenario, the encrypted portions continue to be opaque to any intermediate nodes and can only be decrypted by the intended receiver. For this reason, message-level security is also sometimes referred to as end-to-end security.

This version of XML and Web Services Security provides a framework with which a JAX-RPC application developer will be able to sign and verify SOAP messages. This implementation of XML and Web Services Security attempts to implement portions of the OASIS Web Services Security Working Draft, which may be viewed at the W3C Web site, `http://www.w3.org/Signature/`.

Note: Currently, the Java standard for XML Digital Signatures is undergoing definition under the Java Community Process. This Java standard is JSR 105-XML Digital Signature APIs, which you can read at `http://www.jcp.org/en/jsr/detail?id=105`. The security solution provided in this release of the Java WSDP is based on *nonstandard* APIs, which are subject to change with new revisions of the technology. As standards are defined in the Web Services Security space, these nonstandard APIs will be replaced with standards-based APIs.

This release of the Java WSDP includes samples that illustrate how a JAX-RPC developer can use the XML and Web Services Security framework, as well as documentation for the nonstandard APIs. The example applications can be found in the `<JWSDP_HOME>/xws-security/samples/` directory. The examples start from the `<INSTALL>/jwstutorial13/examples/jaxrpc/helloservice/` example shipped with the Java WSDP Tutorial and add XML Digital Signature. Instructions for running the examples are included in the `README.txt` files included with each sample.

In this release, the default trust-store bundled with the Java WSDP is the only certificate that will be accepted for signing and verifying the requests and responses. This trust-store contains both the client and server certificates.

The following sample applications are included:

- Sample application `dump` prints out both the client and server request and response SOAP messages.
- Sample application `sign` begins with the JAX-RPC `helloservice` sample and configures it so that the response is signed by the server and verified by the client. The request message is not changed.
- Sample application `sign2` starts with the JAX-RPC `helloservice` sample. In this example, the client signs the request, the message is dumped out, the message travels over the network, the server verifies the signature, the business method is called, the server signs the response, the message travels back over the network, and the client verifies the response. This sample also demonstrates how the calling client identity can be retrieved in the business method.
- API documentation for the nonstandard APIs can be viewed at `<`*`JWSDP_HOME`*`>/xws-security/docs/api/index.html`.

Signing and Verifying a SOAP Message, page 997 describes how to use the Java APIs to digitally sign a SOAP message.

Signing and Verifying a SOAP Message

This section discusses using XML and Web Services Security (XWS-Security) to sign and verify SOAP request and response messages. Starting with an unsecured JAX-RPC client, the call gets access to a client proxy object, in this case a static stub. To secure this client, a `ClientHelper` must be created and bound to that proxy object. There can be several kinds of `ClientHelpers` depending on the kind of credentials the client uses. A `ClientHelper` has no credentials associated with it, while a `CertificateClientHelper` carries X509 certificate credentials.

Use the `createFor()` static factory method to create an instance of a `ClientHelper`. Then configure the `ClientHelper` for the actions you want to take. See the example code in Configuring the Server to Verify a Request Received from the Client and to Sign Responses Sent to the Client, page 999 for an example of how to configure the proxy to sign requests and verify responses, and then call the business method as usual:

```
proxy.someBusinessMethod(arg1, arg2);
```

On the server side, there is only one kind of credential, an X509 Certificate credential, which means that there is only one `ServerHelper` class. As with the client, a ServerHelper instance needs to be created and this must be done before a business method is called. Use the init(Object) method of the ServiceLifecycle interface to do this. The unsecured JAX-RPC endpoint must implement ServiceLifecycle and add an init(Object context) method, as shown in the example code in Configuring the Server to Verify a Request Received from the Client and to Sign Responses Sent to the Client, page 999.

In this release, only programmatic security is supported. None of the security information is added to the deployment descriptors.

Configuring the Server to Sign a Server Response

To configure a server to sign all responses to a client, you add a ServerHelper to your server implementation. In the example at *<JWSDP_HOME>*/xws-security/samples/sign/server/src/sign/HelloImpl.java, when a client invokes an RPC service, the server signs the response with the default credential. The code that does this looks like this:

```
public void init(Object context) throws ServiceException {
  // Configure this endpoint to sign the response with the
  // server's credentials.
  ServerHelper.createFor(context).addSignResponse();
}
```

The createFor(context) method creates a ServerHelper instance and binds it to a servlet endpoint implementation. This method will attempt to initialize the server credentials and client credential databases, but will not throw any exceptions upon failure. The server credentials use a JAAS entry name of XwsSecurityServerKey and the client databases use XwsSecurityClientCertificateDatabase.

To read the API documentation for ServerHelper, open *<JWSDP_HOME>*/xws-security/docs/api/index.html, and click the link to ServerHelper.

Configuring a Static Client to Verify Server Responses

To configure a static client to verify server responses, you add a ClientHelper, a utility API used to add security to a proxy by adding a handler to verify all of the responses from the server. A ClientHelper has no client credential but may be associated with an optional server certificate credential. The following code (in bold), from *<JWSDP_HOME>*/xws-security/samples/sign/client/src/sign/

`StaticHelloClient.java`, shows one example of configuring a static client to verify server responses:

```
public class StaticHelloClient {
  public static void main(String[] args) throws Exception {
    Remote proxy = (Remote) createProxy();

    // Create a ClientHelper to verify the response from
    // the server
    ClientHelper.createFor(proxy).addVerifyResponse();

    HelloIF hello = (HelloIF) proxy;
    System.out.println(hello.sayHello("to Duke!"));
  }
}
```

The line of code in **bold** gets the proxy and configures the helper for the proxy using the `ClientHelper` API. The `addVerifyResponse` method adds an action to verify a response message from the server and returns this object in support of method-call chaining. The `createFor` method attempts to bind the Helper to the proxy. The server credentials will be initialized using a JAAS default entry name of `XwsSecurityServerCertificate`. If a server credential could not be located, no exception is thrown because it is possible to call methods on this class that do not require server credentials.

To read the API documentation for `ClientHelper`, open `<JWSDP_HOME>/xws-security/docs/api/index.html`, and click the link to `ClientHelper`.

Configuring the Server to Verify a Request Received from the Client and to Sign Responses Sent to the Client

In Configuring the Server to Sign a Server Response, page 998, configuring a server to sign responses to the client was discussed. This section adds to that example by verifying that the request is from a valid client before sending a response. The `sign2` sample application includes server-side source code that illustrates how to verify a request received from the client in addition to showing how to sign responses sent to the client. This sample also shows you how to extract information about the client principal with which the request was signed.

In these examples, a *Subject* is the term used to identify the source of a request. It could be a person, a service, etc. Each subject could have multiple *Principals*. Each principal is a way of identifying the subject. For example, a subject which is a person could have a name principal and a phone number principal. A subject encompasses the entity's principals and its security credentials.

The following code snippet demonstrates one way to configure the server as described above (from `<JWSDP_HOME>/xws-security/samples/sign2/server/src/sign2/HelloImpl.java`):

```
public class HelloImpl implements HelloIF, ServiceLifecycle {
  private static final String prompt = "Hello ";
  private ServerHelper sh;

  public String sayHello(String s) {
    // The following illustrates how to access the Principal
    // associated with the client signature in a business
    // method. A Subject containing the public credentials
    // can also be accessed, but it is not shown here.
    return prompt + s + " and also to " +
      sh.getClientPrincipal();
  }
...

  public void init(Object context) throws ServiceException {
    // Configure this endpoint to sign the response with the
    // server's credentials.  Also, save the ServerHelper
    // instance in a field so we can access it later from a
    // business method.
    sh = ServerHelper.createFor(context);
    sh.addCertificateVerifyRequest().addSignResponse();

   }

}
```

When you get a request that was signed by the client, use the `getClientPrincipal` method to find out who signed the request. Use the `addCertificateVerifyRequest` method to verify a request message from the client. The client `Subject` and `Principal` will be set to the client identity if verification is successful.

Of course, for this scenario to work, you need to add a sign request to your client code as well. The client source code illustrates how to sign a request sent to the server and to verify the response that is received. In this example, we use the `CertificateClientHelper` API to sign the request from the client. The server code, above, signs the response, and the client verifies the response. The CertificateClientHelper class is used to help set up client-side security using X509 Certificate credentials. A `CertificateClientHelper` is typically associated with a client credential and may be associated with a server certificate credential. The following code snippet is from the sample application at *<JWSDP_HOME>*`/xws-security/samples/sign2/client/src/sign2/StaticHelloClient.java`:

```
// Create a CertificateClientHelper for a client-side
// stub/proxy
CertificateClientHelper cch =
  CertificateClientHelper.createFor(proxy);

// Sign the request and then dump the message for debugging
cch.addSignRequest().addDumpRequest();

// Verify the response which was signed by the server
cch.addVerifyResponse();

// Call the business method
HelloIF hello = (HelloIF) proxy;
System.out.println(hello.sayHello("to Duke!"));
```

The `addSignRequest` method adds an action to sign a request message to the server. If `getClientSubject()` returns null, then this method will throw a `ServiceException` because a client private key is needed to sign a request. The `addDumpRequest` method is used to dump a stack trace to the console for debugging purposes.

To read the API documentation for `ClientCertificateHelper`, open *<JWSDP_HOME>*`/xws-security/docs/api/index.html`, and click on the link to `ClientCertificateHelper`.

Further Discussion of XML Digital Signatures

With this implementation of the XML Digital Signature technology, you can verify the integrity of the message, but anyone who picks up the XML document will be able to see its contents. To add authentication to this example, you can use DSig in combination with SSL technology to encrypt the actual contents of the document. Adding certificate-based authentication to a JAX-RPC application is discussed in Example: Client-Certificate Authentication over HTTP/SSL with JAX-RPC, page 988.

25

The Coffee Break Application

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. There are two versions of the Coffee Break server that you can run: One version consists of Java Servlets, JSP pages, and JavaBeans components; The second version uses JavaServer Faces technology (see Chapter 20) as well Java Servlets, JSP pages, and beans. Using either version, 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 25–1.

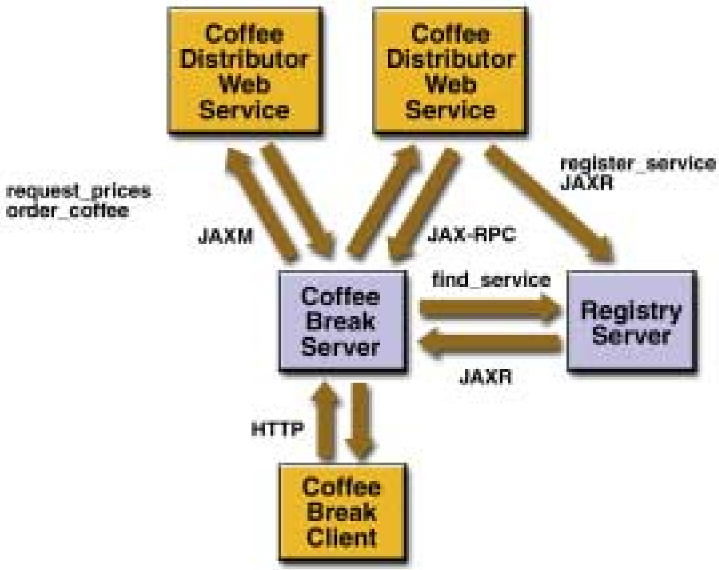


Figure 25–1 Coffee Break Application Flow

Both versions of the Coffee Break server obtain the coffee varieties they sell and their prices by querying distributors at startup and on demand.

1. The Coffee Break servers uses SAAJ messaging to communicate with one of their distributors. The Coffee Break has been dealing with this distributor for some time and has previously made the necessary arrangements for doing request-response SAAJ 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 servers use JAXR to send a query searching for coffee distributors that support JAX-RPC to the Java WSDP Registry Server.
3. The Coffee Break servers request price lists from each of the coffee distributors. The servers make the appropriate remote procedure calls and wait for the response, which is a JavaBeans component representing a price list. The SAAJ distributor returns price lists as XML documents.
4. Upon receiving the responses, the Coffee Break servers process the price lists from the JavaBeans components returned by calls to the distributors.
5. The Coffee Break servers create 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.

Common Code

The Coffee Break servers share a number of JavaBeans components with the JAX-RPC and SAAJ services. The source code for these components resides in the `<INSTALL>/jwstutorial13/examples/cb/common/src/com/sun/cb/` directory:

- `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.

In addition, the common directory contains the `CoffeeBreak.properties` file, which contains the URLs exposed by the Registry Server and the JAX-RPC and SAAJ distributors, the `URLHelper` class, which is used by the server and client classes to retrieve the URLs, and the `DateHelper` utility class.

JAX-RPC Distributor Service

The Coffee Break servers are clients of 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 the JavaBeans components listed in the previous section.

The source code for the `SupplierIF` interface, which follows, resides in the `<INSTALL>/jwstutorial13/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. The source code for the `placeOrder` method follows:

```
public ConfirmationBean placeOrder(OrderBean order) {

  Date tomorrow = DateHelper.addDays(new Date(), 1);
  ConfirmationBean confirmation =
    new ConfirmationBean(order.getId(),
      DateHelper.dateToCalendar(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(DateHelper.dateToCalendar(today),
      DateHelper.dateToCalendar(endDate), priceItems);

  return priceList;
}
```

Publishing the Service in the Registry

Because we want customers to find our service, we publish it in a registry. The objects that publish and remove the 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 and take-down-service commands.

The OrgPublisher program begins by loading String values from the URLHelper class and 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 reads the key from orgkey.txt so that it can delete the service. See Deleting the Service From the Registry, page 1010. 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) {

     String queryURL = URLHelper.getQueryURL();
     String publishURL = URLHelper.getPublishURL();
     String endpoint = URLHelper.getEndpointURL();

     ResourceBundle registryBundle =
        ResourceBundle.getBundle("com.sun.cb.CoffeeRegistry");

     String username =
        registryBundle.getString("registry.username");
     String password =
        registryBundle.getString("registry.password");

     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 581).

First, the `makeConnection` method creates a connection to the Registry Server. See Establishing a Connection (page 572) for more information. To do this, it first specifies a set of connection properties using the query and publish URLs retrieved from `URLHelper`. 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. This code is almost identical to the code in the JAXR examples.

```
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

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);
binding.setValidateURI(false);
binding.setAccessURI(endpoint);
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. Like the OrgPublisher program, the OrgRemover program starts by fetching URLs from URLHelper and other values from the CoffeeRegistry.properties file. One these values, keyFile, is the name of the file that contains the key that uniquely identifies the service. OrgRemover 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 = URLHelper.getQueryURL();
        String publishURL = URLHelper.getPublishURL();
        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 589).

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 `BusinessLifeCycleManager` 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);
}
```

SAAJ Distributor Service

In contrast to the JAX-RPC service, the SAAJ distributor service does not register in a publicly accessible registry. It simply implements 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 messaging they will do. They have agreed to do request-response messaging using the SAAJ API (the `javax.xml.soap` package).

The Coffee Break servers send two kinds of messages:

- Requests for current wholesale coffee prices
- Customer orders for coffee

The SAAJ 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 servers and the SAAJ distributor are specified by the following DTDs:

- `request-prices.dtd`
- `price-list.dtd`
- `coffee-order.dtd`
- `confirm.dtd`

These DTDs may be found at `<INSTALL>/jwstutorial13/examples/cb/saaj/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 SAAJ API.

Code for the client and server applications is in the directory:

```
<INSTALL>/jwstutorial13/examples/cb/saaj/src/
```

SAAJ Client

The Coffee Break servers, which are the SAAJ clients in this scenario, send requests to their SAAJ distributor. The SAAJ client application uses 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. The `SOAPEnvelope` object 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 SAAJ 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 SAAJ coffee supplier, `http://localhost:8080/saaj-coffee-supplier/getPriceList`. The `SOAPConnection` object `con` is used to send the message, and because it is no longer needed, it is closed.

```
URL endpoint = new URL(url);
SOAPMessage response = con.call(msg, endpoint);
con.close();
```

When the `call` method is executed, Tomcat executes the servlet `PriceListServlet`. This servlet creates and returns a `SOAPMessage` object whose content is the SAAJ distributor's price list. (`PriceListServlet` is discussed in Returning the Price List, page 1021.) Tomcat knows to execute `PriceListServlet` because we map 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 SAAJ 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);
Calendar todayCal = new GregorianCalendar();
todayCal.setTime(today);
Calendar cal = new GregorianCalendar();
cal.setTime(endDate);
plb = new PriceListBean();
plb.setStartDate(todayCal);
plb.setPriceItems(priceItems);
plb.setEndDate(cal);
```

Ordering Coffee

The other kind of message that the Coffee Break servers can send to the SAAJ 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 to the endpoint `http://localhost:8080/saaj-coffee-supplier/orderCoffee` and closes the connection.

```
URL endpoint = new URL(url);
SOAPMessage reply = con.call(msg, endpoint);
con.close();
```

Because we map the given endpoint to `ConfirmationServlet`, Tomcat executes that servlet (discussed in Returning the Order Confirmation, page 1026) 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);
Calendar cal = new GregorianCalendar();
cal.setTime(date);
cb = new ConfirmationBean(id, cal);
```

SAAJ Service

The SAAJ coffee distributor, the SAAJ server in this scenario, provides the response part of the request-response paradigm. When SAAJ 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.

```
public class PriceListServlet extends HttpServlet {
  static MessageFactory fac = null;

  static {
    try {
      fac = MessageFactory.newInstance();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  };
```

Every servlet has an `init` method. This `init` method initializes the servlet with the configuration information that Tomcat passed to it.

```
public void init(ServletConfig servletConfig)
    throws ServletException {
  super.init(servletConfig);
}
```

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 = fac.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( "SAAJ 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 `putHead-`

ers 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()) {
        MimeHeader header = (MimeHeader)it.next();

        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 1026), 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);

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 orderID from 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 orderID test 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 751) to achieve a common look and feel among the HTML pages, and many of the JSTL custom tags discussed in Chapter 17.

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 25–1.

Table 25–1 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 coffee 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 SAAJ 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.

ShoppingCart

`ShoppingCart` is a list of shopping cart items. A `ShoppingCartItem` contains a retail price item, the number of pounds of that item, and the total price for that item.

OrderConfirmations

`OrderConfirmations` is a list of order confirmation objects. An `OrderConfirmation` contains order and confirmation objects, already discussed in Service Interface (page 1003).

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(address, customer,
               orderId, lis, total);

            String SAAJOrderURL =
               URLHelper.getSaajURL() + "/orderCoffee";
            if (distributor.equals(SAAJOrderURL)) {
               OrderRequest or = new OrderRequest(SAAJOrderURL);
               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.

JavaServer Faces Version of Coffee Break Server

JavaServer Faces is designed to provide a clean separation of the presentation layer and the model layer so that you can readily add JavaServer Faces functionality to existing applications. In fact almost all of the original Coffee Break Server back-end code remains the same in the JavaServer Faces technology version of the server.

This section provides some details on how the JavaServer Faces version of the Coffee Break server is different from the non-GUI framework version. Like the non-GUI framework version of the Coffee Break Server implementation, the JavaServer Faces Coffee Break server is organized along the Model-View-Controller design pattern. Instead of the `Dispatcher` servlet examining the request URL, creating and initializing model JavaBeans components, and dispatching requests to view JSP pages, the `FacesServlet`, included with the JavaServer Faces API, performs these tasks. As a result, the `Dispatcher` servlet has been removed from the JavaServer Faces version of the Coffee Break server. Some of the code from the `Dispatcher` has been moved to beans. This will be explained later in this section.

As with the non-GUI framework version of the Coffee Break Server, the JavaServer Faces Coffee Break Server includes JavaBeans components that 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 25–2.

Table 25–2 Model and View Components

Function	JSP Page	JavaBeans Component
Update order data	`orderForm`	`CoffeeBreakBean`, `ShoppingCart`
Update delivery and billing data	`checkoutForm`	`CheckoutFormBean`
Display order confirmation	`checkoutAck`	`OrderConfirmations`

JSP Pages

orderForm

Like in the non-GUI framework version of the Coffee Break server, the `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 coffee amounts and clicks the Update button, the request is posted back to `orderForm`.

The CoffeeBreakBean bean component 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 button.

The table of coffees displayed on the orderForm is rendered using one of the JavaServer Faces component tags, data_table. Here is part of the data_table tag from orderForm:

```
<h:data_table id="table"
  columnClasses="list-column-center,list-column-right,
    list-column-center, list-column-right"
  headerClass="list-header" rowClasses="list-row"
  styleClass="list-background"
  value="#{CoffeeBreakBean.cart.items}" var="sci">
  <f:facet name="header">
    <h:output_text  value="#{CBMessages.OrderForm}"/>
  </f:facet>
  <h:column>
    <f:facet name="header">
      <h:output_text  value="Coffee"/>
    </f:facet>
    <h:output_text id="coffeeName"
      value="#{sci.item.coffeeName}"/>
  </h:column>
  ...
</h:data_table>
```

When this tag is processed, a UIData component and a Table renderer are created on the server side. The UIData component supports a data binding to a collection of data objects. The Table renderer takes care of generating the HTML markup. The UIData component iterates through the list of Coffees, and the Table renderer renders each row in the table.

This example is a classic use case for a UIData component because the number of coffees might not be known to the application developer or the page author at the time the application is developed. Also the UIData component can dynamically adjust the number of rows in the table to accommodate the underlying data.

For more information on UIData, please see section The UIData Component (page 821).

checkoutForm

checkoutForm is used to collect delivery and billing information for the customer. When the Submit button is clicked, an ActionEvent is generated. This event is first handled by the submit method of the checkoutFormBean. This method acts as a listener for the event because the tag corresponding to the submit button references the submit method with its action attribute:

```
<h:command_button value="#{CBMessages.Submit}"
  action="#{checkoutFormBean.submit}"/>
```

The submit method submits the suborders to each distributor and stores the result in the request-scoped OrderConfirmations bean.

The checkoutForm page has standard validators on several components and a custom validator on the email component. Here is the tag corresponding to the firstName component, which holds the customer's first name:

```
<h:input_text id="firstName"
  value="#{checkoutFormBean.firstName}"
  size="15" maxlength="20" required="true"/>
```

With the required attribute set to true, the JavaServer Faces implementation will check if the user entered something in the First Name field.

The email component has a custom validator registered on it. Here is the tag corresponding to the email component:

```
<h:input_text id="email" value="#{checkoutFormBean.email}"
  size="25" maxlength="125"
  validator="#{checkoutFormBean.validateEmail}"/>
```

The validator attribute refers to the validateEmail method on the CheckoutFormBean class. This method ensures that the value the user enters in the email field has a "@" character in it.

If the validation does not succeed, the checkoutForm is re-rendered, 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 the checkoutAck page.

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. This page also uses a UIData component. Again, the number of coffees the customer ordered is not known prior to runtime. the `UIData` component dynamically adds rows to accommodate the order.

JavaBeans Components

The JavaBeans components in the JavaServer Faces version of the Coffee Break server are almost the same as those in the original version. This section highlights what has changed and describes the new components.

CheckoutFormBean

The `validate` method of the original version of the `CheckoutFormBean` checks the completeness of information entered into `checkoutForm`. Because JavaServer Faces technology automatically validates certain kinds of data when the appropriate validator is registered on a component, the `validate` method of `checkoutFormBean` is not necessary in the JavaServer Faces version of that bean.

Several of the tags on the `checkoutForm` page have their `required` attributes set to `true`. This will cause the implementation to check if the user enters values in these fields. The tag corresponding to the `email` component registers a custom validator on the `email` component, as explained in checkoutForm (page 1033). The code that performs the validation is the `validateEmail` method:

```
public void validateEmail(FacesContext context,
  UIInput toValidate) {
  String message = "";
  String email = (String) toValidate.getValue();
  if (email.indexOf('@') == -1) {
     toValidate.setValid(false);
     message = CoffeeBreakBean.loadErrorMessage(context,
        CoffeeBreakBean.CB_RESOURCE_BUNDLE_NAME,
           "EMailError");
     context.addMessage(toValidate.getClientId(context),
        new FacesMessage(message, ""));
  } else {
      toValidate.setValid(true);
  }
}
```

As in the non-GUI framework version of Coffee Break server, if the information is incomplete or invalid, the page is re-rendered to display 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.

CoffeeBreakBean

CoffeeBreakBean is exclusive to the JavaServer Faces technology version of the Coffee Break server. It acts as the backing bean to the JSP pages. See Backing Bean Management (page 799) for more information on backing beans. It creates the ShoppingCart object, which defines the model data for the components on the orderForm page that hold the data about each coffee. The CoffeeBreakBean also loads the RetailPriceList object. In addition, it provides the methods that are invoked when the buttons on the orderForm and checkoutAck are clicked. For example, the checkout method is invoked when the Checkout button is clicked because the tag corresponding to the Checkout button refers to the checkout method with its action attribute, shown here:

```
<h:command_button id="checkoutLink"
  value="#{CBMessages.Checkout}"
  action="#{CoffeeBreakBean.checkout}" />
```

The checkout method returns a String, which the JavaServer Faces page navigation system matches against a set of navigation rules to determine what page to access next. The navigation rules are defined in a separate XML file, described in the next section.

Resource Configuration

A JavaServer Faces application usually includes an XML file that configures resources for the application. These resources include JavaBeans components, navigation rules, and others.

Two of the resources configured for the JavaServer Faces version of the Coffee Break server are the CheckoutForm bean and navigation rules for the orderForm page, shown here:

```
<managed-bean>
  <managed-bean-name>checkoutFormBean</managed-bean-name>
  <managed-bean-class>
    com.sun.cb.CheckoutFormBean
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
    <managed-property>
```

```
            <property-name>firstName</property-name>
            <value>Coffee</value>
         </managed-property>
         <managed-property>
            <property-name>lastName</property-name>
            <value>Lover</value>
         </managed-property>
         <managed-property>
            <property-name>email</property-name>
            <value>jane@home</value>
         </managed-property>
         ...
      </managed-bean>

   <navigation-rule>
      <from-view-id>/orderForm.jsp</from-view-id>
      <navigation-case>
         <from-outcome>checkout</from-outcome>
         <to-view-id>/checkoutForm.jsp</to-view-id>
      </navigation-case>
   </navigation-rule>
```

As shown in the managed-bean element, the `checkoutForm` bean properties are initialized with the values for the user, Coffee Lover. This way, the hyperlink tag from `orderForm` is not required to submit these values in the request parameters.

As shown in the navigation-rule element, when the `String`, `checkout`, is returned from a method referred to by a component's `action` attribute, the `checkoutForm` page displays.

Building, Packaging, Installing, and Running the Application

The source code for the Coffee Break application is located in the directory `<INSTALL>/jwstutorial13/examples/cb/`. Within the `cb` directory are subdirectories for each Web application—`saaj`, `jaxrpc`, `server`, `server-jsf`—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.

Setting the Port

Several files in the Coffee Break depend on the port that you specified when you installed the Java WSDP. The tutorial examples assume that the server runs on the default port, 8080. If you have changed the port, you must update the port number in the following files before building and running the examples:

- `<INSTALL>/jwstutorial13/examples/cb/common/src/com/sun/cb/CoffeeBreak.properties`
- `<INSTALL>/jwstutorial13/examples/cb/jaxrpc/config-wsdl.xml`

Building the Common Classes

The Coffee Break applications share a number of common utility classes. To build the common classes:

1. In a terminal window, go to `<INSTALL>/jwstutorial13/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 `<INSTALL>/jwstutorial13/examples/cb/jaxrpc/`.
2. Run `ant build-service`. This taskthe JAR file containing the JAXR routines and run `wscompile` and creates the WAR file of the JAX-RPC service.

   If you get an error, make sure you edited the file `<INSTALL>/jwstutorial13/examples/common/build.properties` as described in Building the Examples (page xxvi).
3. Start Tomcat.
4. Run ant `set-up-service`. This task registers the service with the Registry Server and installs it into Tomcat. The registration process can take some time, so wait until you see output like the following before proceeding to the next step:

```
run-jaxr-publish:
  [echo] Running OrgPublisher.
  [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. Run `ant build-client`. This task creates the JAR file that contains the classes needed by JAX-RPC clients. The `build-client` task runs `wscompile` to generate the stubs and JavaBeans components.
6. Test that the JAX-RPC service has been installed correctly by running the test programs `ant run-test-order` and `ant run-test-price`

   Here is what you should see when you run `ant run-test-price`:

```
run-test-price:
run-test-client:
   [java] 07/21/03 08/20/03
   [java] Kona 6.50
   [java] French Roast 5.00
   [java] Wake Up Call 5.50
   [java] Mocca 4.00
```

Building and Installing the SAAJ Service

To build the SAAJ service and client library and install the SAAJ service:

1. In a terminal window, go to `<INSTALL>/jwstutorial13/examples/cb/saaj/`.
2. Run `ant build`. This task creates the client library and compiles the server classes into the correct location for installation.
3. Make sure Tomcat is started.
4. Run `ant install`.
5. Test that the SAAJ service has been installed correctly by running one or both of the test programs `ant run-test-price` and `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 `<INSTALL>/jwstutorial13/examples/cb/server/`.
2. Run `ant build`. This task compiles the server classes and copies the classes, JSP pages, and tag libraries into the correct location for packaging.
3. Make sure Tomcat is started.
4. Run `ant install`.

Building and Installing the JavaServer Faces Technology Version of the Coffee Break Server

To build and install the JavaServer Faces technology version of the Coffee Break server:

1. Download the beta release of the JavaServer Faces reference implementation.
2. Follow the Installation Instructions—a document located in the `docs` directory of the JavaServer Faces reference implementation download and available online—to remove the JavaServer Faces reference implementation included with Java WSDP 1.3 and replace it with the beta version.

1. In a terminal window, go to `<INSTALL>/jwstutorial13/examples/cb/server-jsf/`.
2. Open the `build.properties` file and set the `jsf-api.jar` and `jsf-impl.jar` properties to point to these JAR files, included in the JavaServer Faces technology download.
3. Run `ant build`. This task compiles the server classes and copies the classes, JSP pages, tag libraries, and other necessary files into the correct location for packaging.
4. Make sure Tomcat is started.
5. Run `ant install`.

Running the Coffee Break Client

After you have installed all the Web applications, check that all the applications are running by opening the URL `http://localhost:8080/manager/html` in a browser and entering your username and password in the dialog that appears. You should see `cbserver`, `jaxrpc-coffee-supplier`, and `saaj-coffee-supplier` in the list of applications.

If you have installed the non-GUI framework version of the Coffee Break server, you can run the Coffee Break client by opening the Coffee Break server URL in a Web browser:

```
http://localhost:8080/cbserver/orderForm
```

If you have installed the JavaServer Faces technology version of the Coffee Break server, you can run the JavaServer Faces version of the Coffee Break client by opening this URL in a Web browser:

```
http://localhost:8080/cbserver/faces/orderForm.jsp
```

You should see a page something like the one shown in Figure 25–2.

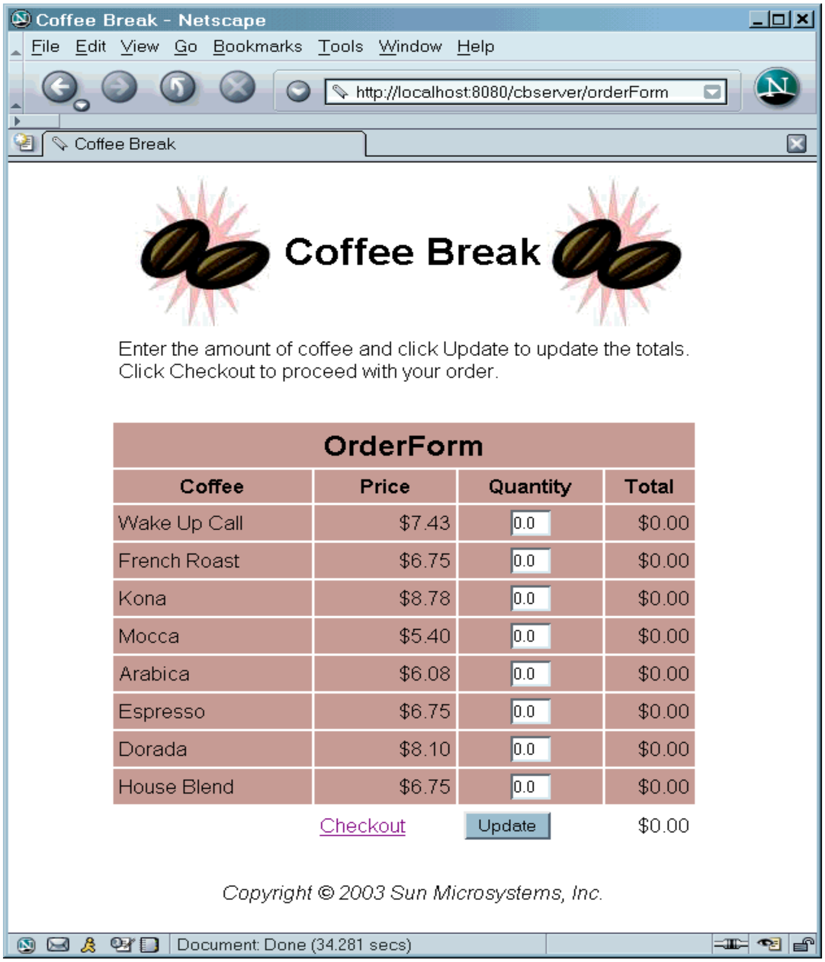


Figure 25–2 Order Form

After you have gone through the application screens, you will get an order confirmation that looks like the one shown in Figure 25–3.

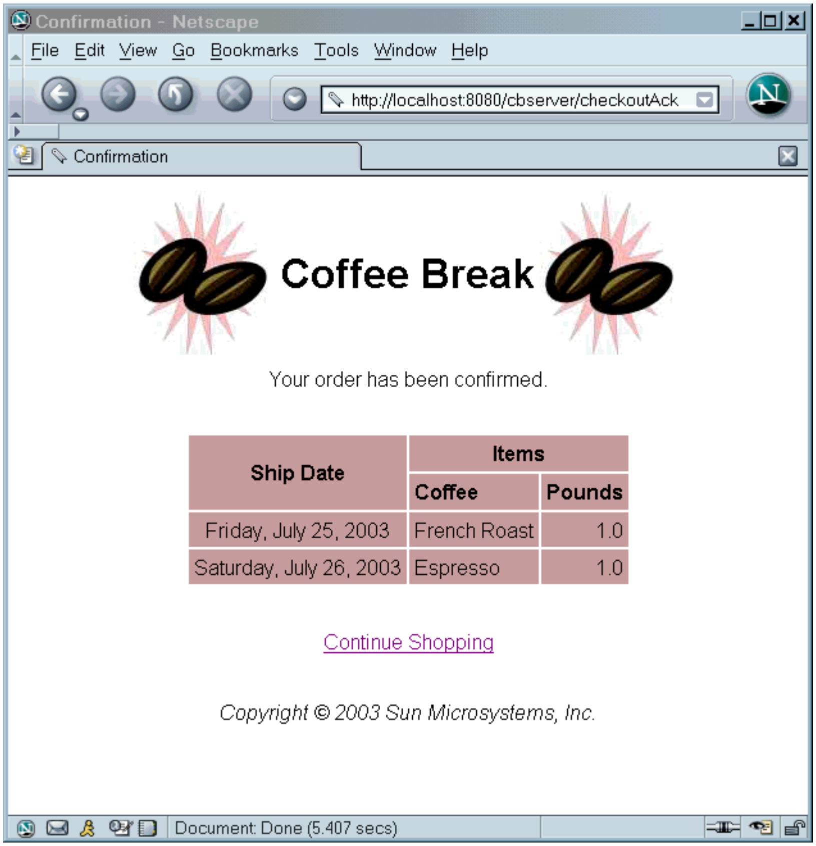


Figure 25–3 Order Confirmation

Removing the Coffee Break Application

To remove the Coffee Break application, perform the following steps:

1. If you have installed the non-GUI framework version of the Coffee Break server, in a terminal window, go to `<INSTALL>/jwstutorial13/examples/cb/server/`. Otherwise, if you have installed the JavaServer Faces version of the Coffee Break server, go to `<INSTALL>/jwstutorial13/examples/cb/server-jsf/`
2. Run `ant remove`.
3. Go to `<INSTALL>/jwstutorial13/examples/cb/saaj/`.
4. Run `ant remove`.

5. Go to `<INSTALL>/jwstutorial13/examples/cb/jaxrpc/`.
6. Run `ant take-down-service`.
7. Stop Tomcat.

If you want to remove the `build` and `dist` directories, run `ant clean` in each directory, including `<INSTALL>/jwstutorial13/examples/cb/common/`.

A

Tomcat Web Server Administration Tool

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 Tomcat 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 as follows:
   - On the Unix platform, type the following at a terminal window:

     <*JWSDP_HOME*>`/bin/startup.sh`

- On the Microsoft Windows platform, start Tomcat from the Start menu by following this chain: Start→Programs→Java Web Services Developer Pack 1.3→Start Tomcat.

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 a user name and password combination that has been assigned the role of `admin`, such as the user name and password that you entered when you installed the Java WSDP, or a user name and password combination that you entered subsequent to installation and to which you have assigned the role of admin.

   For security purposes, `admintool` 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.

   If you've forgotten this 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" />
   ```

When a successful user name and password combination is entered, the `admintool` Web application displays in the Web browser window:

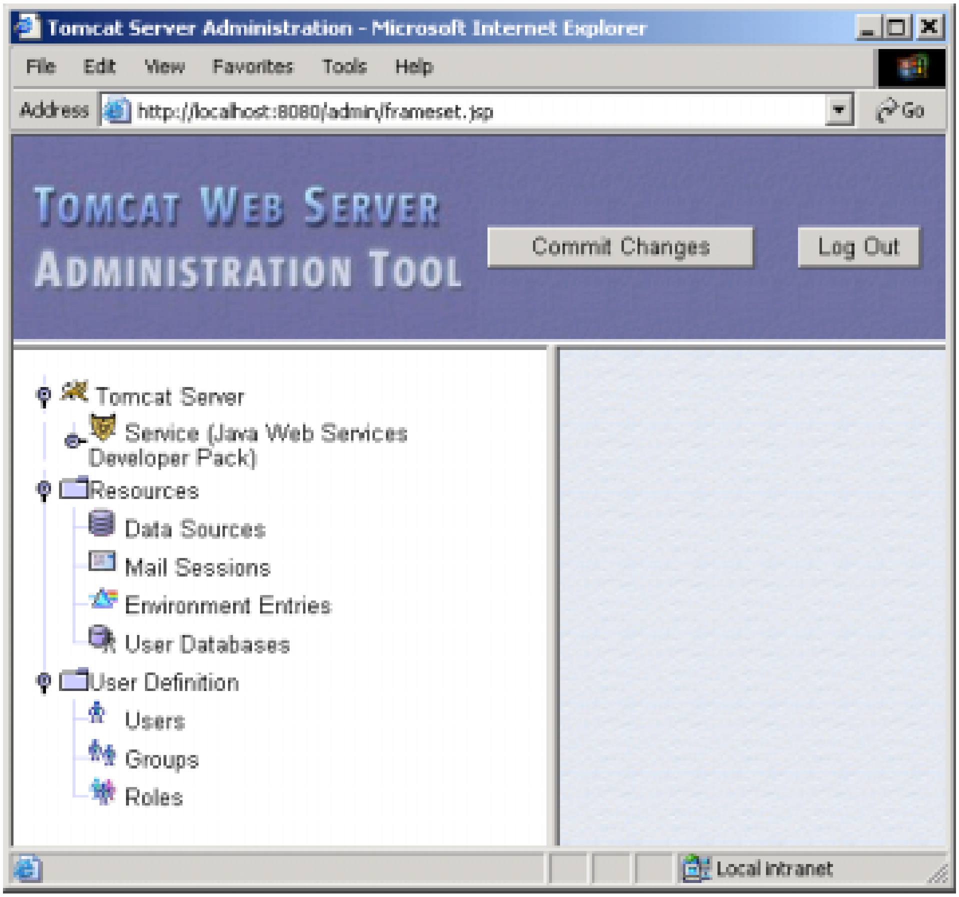


Figure A–1 The Tomcat Server Administration Tool

Once `admintool` is running, you can perform any of the Tomcat Web Server administration tasks listed in the rest of this appendix. After you have made changes to Tomcat, select the Save button on that page to save the attributes for the current Tomcat process.

If you want the changes to the Tomcat server to be available when Tomcat is restarted, select the Commit Changes button. This writes the changes to the `<JWSDP_HOME>/conf/server.xml` file. 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.2003-10-15.12-11-54`. To restore a previous configuration, shut down Tomcat, rename the file to `server.xml`, and restart Tomcat.

Log out of `admintool` by selecting Log Out when you are finished.

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 `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/index.html`.

This appendix 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 1084).

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

Table A–1 Server Properties (Continued)

Property	**Description**
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 a Java Web Services Developer Pack 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 1062).

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 1048) 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 1054) 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 1062) for more information.
- User **Realm** elements represent a database of user names, passwords, and roles assigned to those users. See Configuring Realm Elements (page 1065) 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 1072) 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 1050) 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 `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/index.html` or the document titled *JK 2 Connectors* at `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/index.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 970) contains detailed instructions on setting up an HTTPS connector.

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
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).
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.
Secure	Set this attribute to True if you wish to have calls to `request.isSecure()` return True for requests received by this Connector (you would want this on an SSL Connector). The default value is False.
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.

Table A–2 Common Connector Attributes (Continued)

Attribute	Description
Compression	The Connector may use HTTP/1.1 GZIP compression in an attempt to save server bandwidth. The acceptable values for the parameter are `Off` (disable compression), `On` (allow compression, which causes text data to be compressed), `Force` (forces compression in all cases), or a numerical integer value (which is equivalent to `On`, but specifies the minimum amount of data before the output is compressed). If the content-length is not known and compression is set to `On` or more aggressive, the output will also be compressed. If not specified, this attribute is set to `False`.
Connection Linger	The number of milliseconds during which the sockets used by this Connector will linger when they are closed. The default value is -1 (socket linger is disabled).
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).
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.
Disable Upload Timeout	This flag allows the servlet container to use a different, longer connection timeout while a servlet is being executed, which in the end allows either the servlet a longer amount of time to complete its execution, or a longer timeout during data upload. If not specified, this attribute is set to `False`.
Max Keep Alive Requests	The maximum number of HTTP requests which can be pipelined until the connection is closed by the server. Setting this attribute to 1 will disable HTTP/1.0 keep-alive, as well as HTTP/1.1 keep-alive and pipelining. If not specified, this attribute is set to 100.
Max Spare Threads	The maximum number of unused request processing threads that will be allowed to exist until the thread pool starts stopping the unnecessary threads. The default value is 50.

Table A–2 Common Connector Attributes (Continued)

Attribute	Description
Max Threads	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 200.
Min. Spare Threads	The number of request processing threads that will be created when this Connector is first started. The connector will also make sure it has the specified number of idle processing threads available. This attribute should be set to a value smaller than that set for Max Threads. The default value is 4.
TCP No Delay	If set to `True`, the `TCP_NO_DELAY` option will be set on the server socket, which improves performance under most circumstances. This is set to `True` by default.
X Powered By	A boolean value that indicates if the X-Powered-By response header is enabled or disabled.
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. The default is -1.
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–3 may also be set.

Table A–3 HTTPS Attributes

Attribute	Description
Algorithm	The certificate encoding algorithm to be used. If not specified, the default value is SunX509.
Ciphers	A comma separated list of the encryption ciphers that may be used. If not specified, any available cipher may be used.
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`.
Keystore Type	The type of keystore file to be used for the server certificate. If not specified, the default value is JKS.
SSL Protocol	The version of the SSL protocol to use. If not specified, the default is TLS.

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 970) 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 1057).

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 1058).
- Logger Elements, which are discussed in Configuring Logger Elements (page 1062).
- Valve Elements, which are discussed in Configuring Valve Elements (page 1072).
- Host Aliases, which are discussed in Host Name Aliases (page 1057).

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 1055) 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 `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/config/host.html`.

Host Attributes

The attributes shown in Table A–4 may be viewed, set, or modified for a Host.

Table A–4 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.

Table A–4 Host Attributes (Continued)

Attribute	Description
Auto Deploy	This flag value indicates if Web applications from this host should be automatically deployed by the host configurator. The flag's value defaults to True. See Automatic Application Deployment at `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/config/host.html#Automatic%20Application%20Deployment` for more information.
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).
Deploy on Startup	This flag value indicates if Web applications from this host should be automatically deployed by the host configurator. The flag's value defaults to True. See Automatic Application Deployment at `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/config/host.html#Automatic%20Application%20Deployment` for more information.
Deploy XML	Set to False if you want to disable deploying applications from using a Context XML configuration file. This also disables the ability to install web application directories or WAR files with the `manager` application that are not located in the Host appBase directory. Applications are deployed with the security permissions of `catalina`, for security this may need to be set to False if untrusted users can manage web applications. The flag's value defaults to True.
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`.

Table A–4 Host Attributes (Continued)

Attribute	Description
XML Namespace Aware	In addition to the automatic deployment that occurs at startup time, you can also request that new XML configuration files that are dropped in the appBase (or $CATALINA_HOME/conf/[engine_name]/[host_name] in the case of an XML configuration file) directory while Tomcat is running will be automatically deployed, according to the rules described in `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/config/host.html#Automatic%20Application%20Deployment`.
XML Validation	A boolean value used to turn XML validation on and off.

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.

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 1059) 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 `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/config/context.html`.

Context Attributes

The Context element page contains three types of properties:

- Context Properties, described in Table A–5.
- Loader Properties, described in Table A–6.
- Session Manager Properties, described in Table A–7.

The attributes shown in Table A–5 may be viewed, set, or modified for Context properties.

Table A–5 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).

Table A–5 Context Properties

Attribute	Description
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.
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.

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–6 may be viewed, set, or modified for Loader properties.

Table A–6 Loader Properties

Attribute	**Description**
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–7 may be viewed, set, or modified for Session Manager properties.

Table A–7 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).

Table A–7 Session Manager Properties

Attribute	**Description**
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 1064) 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 `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/config/logger.html`.

Logger Attributes

Common attributes for all of the Logger types are outlined in Table A–8.

Table A–8 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–9.

Table A–9 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–9 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 (default) 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 1067).

- `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 1068).

- `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 1071).

- `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 1070).

To learn more about Realms, read the document titled *Realm Component* at `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/config/realm.html` or *Realm Configuration How To* at `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/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–10.

Table A–10 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 user's 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–10 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–11.

Table A–11 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–11 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 1074).
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 Element	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–11 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–12.

Table A–12 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. 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–13.

Table A–13 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 1072) 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 `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/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–14.

Table A–14 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 1074) 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.
Rotatable	Default True. Flag to determine if log rotation should occur. If set to False, this file is never rotated and `fileDateFormat` is ignored. Use with caution!
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–15.

Table A–15 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–16.

Table A–16 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–16 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 http://jakarta.apache.org/tomcat/tomcat-5.0-doc/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. Select the Data Source node.
2. Select Create New Data Source from the Available Actions list.
3. Enter the preferred values for the new Data Source. See Table A–17 for details on the attributes.
4. Select Save to save the changes for this session. Select Commit Changes to save the changes for when Tomcat is restarted.

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–17 may be viewed, set, or modified for a Data Source.

Table A–17 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.
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 1079) 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–18.

Table A–18 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`.

Table A–18 Environment Entries Attributes (Continued)

Attribute	Description
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 1081) 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–19.

Table A–19 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

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

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` and `manager`.

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 finished. 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="manager"/>
  <role rolename="admin"/>
  <user username="your_name" password="your_password"
    roles="admin,manager"/>
</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 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.
3. Select Save when finished.

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 finished. 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="manager"/>
  <role rolename="admin"/>
  <user username="your_name" password="your_password"
    roles="admin,manager"/>
  <user username="Anil" password="13345" fullName=""
    roles="customer"/>
</tomcat-users>
```

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 `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/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 `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/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 `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/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 `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/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 `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/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 `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/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 970). The Tomcat documentation at

`http://jakarta.apache.org/tomcat/tomcat-5.0-doc/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

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`

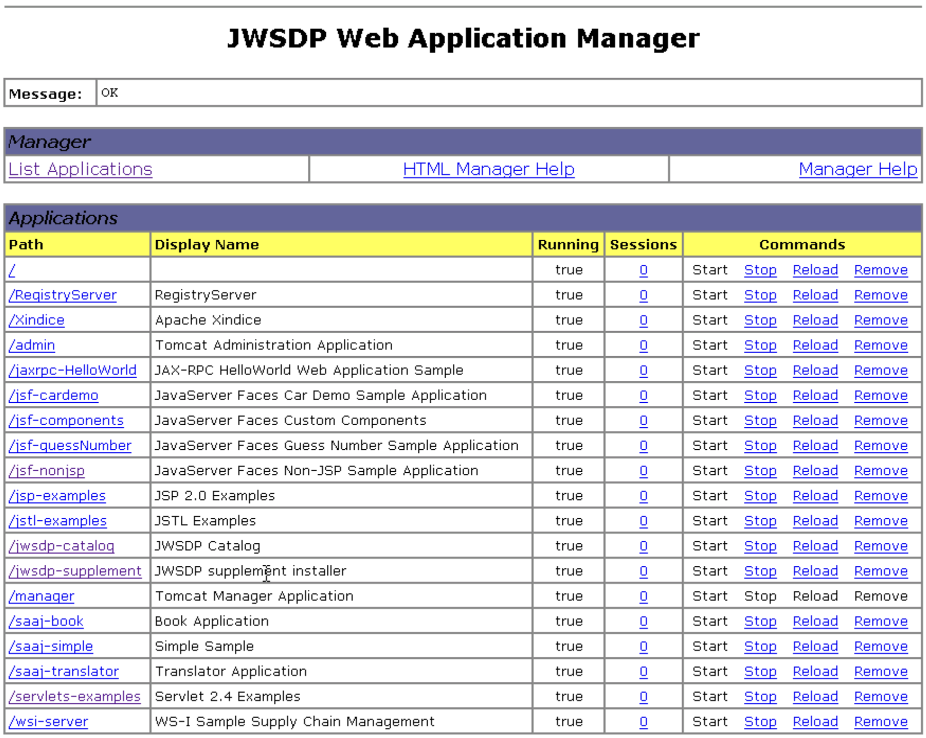
Table B–1 Tomcat Web Application Manager Commands

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*, at `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/manager-howto.html`, contains reference information about the manager application.

There is also a GUI for the manager application. It is accessed via the URL http://<*host*>:8080/manager/html. As shown in Figure B–1, when you first start Tomcat, the manager application displays all the Web applications distributed with the Java WSDP.



JWSDP Web Application Manager

Message:	OK

Manager		
List Applications	HTML Manager Help	Manager Help

Applications				
Path	Display Name	Running	Sessions	Commands
/		true	0	Start Stop Reload Remove
/RegistryServer	RegistryServer	true	0	Start Stop Reload Remove
/Xindice	Apache Xindice	true	0	Start Stop Reload Remove
/admin	Tomcat Administration Application	true	0	Start Stop Reload Remove
/jaxrpc-HelloWorld	JAX-RPC HelloWorld Web Application Sample	true	0	Start Stop Reload Remove
/jsf-cardemo	JavaServer Faces Car Demo Sample Application	true	0	Start Stop Reload Remove
/jsf-components	JavaServer Faces Custom Components	true	0	Start Stop Reload Remove
/jsf-guessNumber	JavaServer Faces Guess Number Sample Application	true	0	Start Stop Reload Remove
/jsf-nonjsp	JavaServer Faces Non-JSP Sample Application	true	0	Start Stop Reload Remove
/jsp-examples	JSP 2.0 Examples	true	0	Start Stop Reload Remove
/jstl-examples	JSTL Examples	true	0	Start Stop Reload Remove
/jwsdp-catalog	JWSDP Catalog	true	0	Start Stop Reload Remove
/jwsdp-supplement	JWSDP supplement installer	true	0	Start Stop Reload Remove
/manager	Tomcat Manager Application	true	0	Start Stop Reload Remove
/saaj-book	Book Application	true	0	Start Stop Reload Remove
/saaj-simple	Simple Sample	true	0	Start Stop Reload Remove
/saaj-translator	Translator Application	true	0	Start Stop Reload Remove
/servlets-examples	Servlet 2.4 Examples	true	0	Start Stop Reload Remove
/wsi-server	WS-I Sample Supply Chain Management	true	0	Start Stop Reload Remove

Figure B–1 Tomcat 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 the directory <*INSTALL*>/examples/common 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.

The Ant build files import these properties with the following element:

```
<property file="../common/build.properties"/>
```

The document *Manager App HOW-TO*, at `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/manager-howto.html`, contains reference information about the Ant tasks.

C

The Java WSDP Registry Server

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 Java API for XML Registries (page 567). 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 1095) 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 add and delete Registry Server users by means of a script.

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.

To start Tomcat on Windows, choose Java Web Services Developer Pack 1.*x*→Start Tomcat from the Start menu.

To start Tomcat on a UNIX system, use the following command:

```
<JWSDP_HOME>/bin/startup.sh
```

To stop Tomcat on Windows, choose Java Web Services Developer Pack 1.*x*→Stop Tomcat from the Start menu.

To stop Tomcat on a UNIX system, use the following command:

```
<JWSDP_HOME>/bin/shutdown.sh
```

Changing the Port for the Registry Server

Normally you run Tomcat on port 8080. If another application uses this port, you can change the Tomcat port by editing the `<JWSDP_HOME>/conf/server.xml` file. Open the file in a text editor and find the comment “Define a non-SSL Coyote HTTP/1.1 Connector on port 8080.” In the Connector element that follows, change this attribute to some other port value, such as 8082 or 8083:

```
port=”8080”
```

In order to run the Registry Server on a changed Tomcat port, you must also edit the file `<JWSDP_HOME>/jwsdp-shared/bin/launcher.xml`. Find the following lines (they are all on one line):

```
<sysproperty key="org.apache.xindice.host"
value="desired Xindice host"/>
<sysproperty key="org.apache.xindice.port"
value="desired Xindice port"/>
```

Make the host and port the same as those for Tomcat. Uncomment these properties before you save the file.

Using JAXR to Access the Registry Server

You can access the Registry Server by using the sample programs in the *<INSTALL>*/jwstutorial13/examples/jaxr/ directory. For details on how these examples work and how to run them, see Running the Client Examples (page 595).

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 Tomcat is using a nondefault port, change `8080` to the correct value for your system.

   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.

Adding and Deleting Users

To add a new user to the Registry Server database, you use the script `registry-server-test.bat` (Windows) or `registry-server-test.sh` (UNIX), in the directory <*JWSDP_HOME*>`/registry-server/samples/`. This script uses files in the directory <*JWSDP_HOME*>`/registry-server/samples/xml/`. You use the same script to delete a user.

Adding a New User to the Registry

To add a new user to the Registry Server database, you first use thescript to generate a hash password for the user. Then you use the file `UserInfo.xml` in the `xml` subdirectory. Perform the following steps:

1. Go to the directory <*JWSDP_HOME*>`/registry-server/samples/`.
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 password 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. Type 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

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/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 and the Registry Server (see The Java WSDP Registry Server, page 1089).

Starting the Browser

To start the browser, go to the directory `<JWSDP_HOME>/jaxr/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 1090) 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 type

```
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, type 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 and the default Registry Server URL:

```
http://localhost:8080/RegistryServer
```

If you are accessing the Registry Server on a remote system, replace `localhost` with the fully qualified hostname of the system where the Registry Server is running. If Tomcat is running on a nondefault port, replace `8080` with the correct port number. 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. Type 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 type 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: Type the name of the organization.
- Id: You cannot type or modify data in this field; the ID value is returned by the registry when you submit the data.
- Description: Type a description of the organization.

Use the Primary Contact Information fields as follows:

- Name: Type the name of the primary contact person for the organization.
- Phone: Type the primary contact's phone number.
- Email: Type 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 1100).

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: Type a name for the service.
- Id field: You cannot type or modify data in this field for a level 0 JAXR provider.
- Description field: Type 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 1100) 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: Type a description of the service binding.
- Access URI: Type 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, type 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, type 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

HTTP Overview

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.

F

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

The character set and encoding names recognized by Internet authorities are listed in the IANA charset registry:

```
http://www.iana.org/assignments/character-sets
```

The Java programming language represents characters internally using the Unicode character set, which provides support for most languages. For storage and transmission over networks, however, many other character encodings are used. The Java 2 platform therefore also supports character conversion to and from other character encodings. Any Java runtime must support the Unicode transformations UTF-8, UTF-16BE, and UTF-16LE as well as the ISO-8859-1 character encoding, but most implementations support many more. For a complete list of the encodings that can be supported by the Java 2 platform, see:

```
http://java.sun.com/j2se/1.4/docs/guide/intl/encoding.doc.html
```

Glossary

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.

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.

business logic

The code that implements the functionality of an application.

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. *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 architecture.

Connector

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 declarations 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 with 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

In general, an XML structure in which one or more elements contains text intermixed with subelements. See also *data*.

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 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 or remote interfaces or both. 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, "<", 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 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 < 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.

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 or content or both of a request or response. Filters differ from *Web components* in that they usually do not themselves create responses but rather 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.

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.

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.

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.

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 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.

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 schema 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 e-mail.

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 that 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.

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*.

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.

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 data types and namespaces to prevent naming conflicts.

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 (OS) on which the Web services platform is executing.

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.

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 a different address space or on a different machine from the clients that access it. Note: An *enterprise information system* is referred to as a 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 or principals or both, recognized by the container to security roles specified in the *deployment descriptor*. Security roles have to be mapped by the deployer 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 or both.

security constraint

A declarative way to annotate the intended protection of Web content. A security constraint consists of a *Web resource collection*, an *authorization constraint*, and a *user data constraint*.

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 or protocols 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 detect errors in static SQL statements at program development time, rather than at execution time as with a JDBC driver.

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 upon or tampered with. Servers are always authenticated and clients are optionally authenticated.

standalone client

A client that does not use a messaging provider and does not run in a container.

system administrator

The person responsible for configuring and administering the enterprise's computers, networks, and software systems.

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 transformation.

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 represents 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: `proto-`

`col://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 of which the servlet is a part. 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 that 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.

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.

W3C

World Wide Web Consortium. The international body that governs Internet standards.

WAR file

Web application archive file. 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*. 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*.

Xalan

An interpreting version of XSLT.

XHTML

An XML look-a-like 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 document type definition (*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 schemas 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.

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`.

JavaServer Faces Technology, JavaServer Pages Technology

Jennifer Ball is a staff writer at Sun Microsystems, where she documents JavaServer Faces technology. Previously she has documented the Java2D API, the J2EE SDK deploytool, and JAXB. She holds an M.A. degree in Interdisciplinary Computer Science from Mills College.

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 enterprise application development methods, object-oriented databases, application servers, and Web technologies. 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, Security, Administration Tool

Debbie Bode Carson is a staff writer with Sun Microsystems, where she documents the J2EE, J2SE, and Java Web Services platforms. In previous positions she documented creating database applications using C++ and Java technologies and creating distributed applications using Java technology. In addition to the chapters in this book, she also currently writes about the CORBA technologies Java IDL and Java Remote Method Invocation over Internet InterORB Protocol (RMI-IIOP), various topics in Web services, and several Java platform tools. Besides Java technology and writing, she loves surfing, sailing, and traveling.

, Introduction to Web ServicesJava Architecture 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 maintains Sun's XML Web site. In addition to his Web work, Scott has written many developer guides, administrator guides, user guides, specifications, white papers, 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*.

Ian Evans is an editor and staff writer at Sun Microsystems, where he edits the J2EE platform specifications and documents the J2EE platform. In previous positions he has documented programming tools, CORBA middleware, and Java application servers, and taught classes on UNIX, web programming, and server-side Java development.

Java API for XML Registries, SOAP with Attachments API for Java

Kim Haase is a staff writer with Sun Microsystems, where she documents the J2EE platform and Java Web Services. 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, and SOAP with Attachments API for Java. She is a co-author of *Java™ Message Service API Tutorial and Reference*.

Security

Eric Jendrock is a staff writer with Sun Microsystems, where he documents the J2EE platform and Java Web Services. Previously, he documented middleware products and standards. Currently, he writes about the Java Web Services Developer Pack, Java Architecture for XML Binding, and J2EE platform and Web security.

Index