The Java™ Web Services Tutorial

For Java Web Services Developer's Pack, v2.0

February 17, 2006

Contents

About This Tutorial. ix

Who Should Use This Tutorial ix
Prerequisites ix
How to Use This Tutorial x
About the Examples x
Required Software xi
Building the Examples xi
How to Print This Tutorial xiii
Typographical Conventions xiii
Feedback xiv

Chapter 1: Building Web Services with JAX-WS xv

Setting the Port xvi
Creating a Simple Web Service and Client with JAX-WS xvi
Requirements of a JAX-WS Endpoint xviii
Coding the Service Endpoint Implementation Class xix
Building the Service xix
Packaging and Deploying the Service xx
A Simple JAX-WS Client xxi
Types Supported by JAX-WS xxiii
Web Services Interoperability and JAX-WS xxiii
Further Information xxiv

Chapter 2: Binding between XML Schema and Java Classes . 1

JAXB Architecture 1
Architectural Overview 2
The JAXB Binding Process 3
More About Unmarshalling 4
More About Marshalling 4

More About Validation 4
Representing XML Content 5
Java Representation of XML Schema 5
Binding XML Schemas 5
Simple Type Definitions 5
Default Data Type Bindings 6
Customizing JAXB Bindings 8
Schema-to-Java 8
Java-to-Schema 9

Chapter 3: Using JAXB 11

General Usage Instructions 12
Description 12
Using the Examples 15
Configuring and Running the Samples 15
JAXB Compiler Options 16
JAXB Schema Generator Options 18
About the Schema-to-Java Bindings 19
Schema-Derived JAXB Classes 22
Basic Examples 29
Unmarshal Read Example 29
Modify Marshal Example 31
Unmarshal Validate Example 33
Customizing JAXB Bindings 35
Why Customize? 35
Customization Overview 36
Customize Inline Example 49
Datatype Converter Example 55
External Customize Example 56
Fix Collides Example 59
Bind Choice Example 64
Java-toSchema Examples 66
j2s-create-marshal Example 66
j2s-xmlAccessorOrder Example 66
j2s-xmlAdapter-field Example 69
j2s-xmlAttribute-field Example 72
j2s-xmlRootElement Example 73
j2s-xmlSchemaType-class Example 73
j2s-xmlType Example 74

Chapter 4: Streaming API for XML 77

Why StAX? 77
Streaming Versus DOM 78
Pull Parsing Versus Push Parsing 79
StAX Use Cases 79
Comparing StAX to Other JAXP APIs 80
StAX API 81
Cursor API 81
Iterator API 82
Choosing Between Cursor and Iterator APIs 87
Using StAX 89
StAX Factory Classes 89
Resources, Namespaces, and Errors 91
Reading XML Streams 92
Writing XML Streams 95
Sun's Streaming Parser Implementation 97
Reporting CDATA Events 97
SJSXP Factories Implementation 98
Sample Code 99
Sample Code Organization 99
Configuring Your Environment for Running the Samples 100
Running the Samples 101
Sample XML Document 102
cursor Sample – CursorParse.java 103
cursor2event Sample – CursorApproachEventObject.java 105
event Sample – EventParse.java 106
filter Sample – MyStreamFilter.java 109
readnwrite Sample – EventProducerConsumer.java 111
writer Sample – CursorWriter.java 114
Further Information 116

Chapter 5: SOAP with Attachments API for Java 117

Overview of SAAJ 118
Messages 118
Connections 122
Tutorial 123
Creating and Sending a Simple Message 124
Adding Content to the Header 133
Adding Content to the SOAPPart Object 134
Adding a Document to the SOAP Body 136

Manipulating Message Content Using SAAJ or DOM APIs 136
Adding Attachments 137
Adding Attributes 139
Using SOAP Faults 145
Code Examples 151
Request.java 152
MyUddiPing.java 153
HeaderExample.java 160
DOMExample.java and DOMSrcExample.java 162
Attachments.java 166
SOAPFaultTest.java 168
Further Information 170

Chapter 6: Java API for XML Registries 171

Overview of JAXR 171
What Is a Registry? 171
What Is JAXR? 172
JAXR Architecture 173
Implementing a JAXR Client 174
Establishing a Connection 175
Querying a Registry 181
Managing Registry Data 186
Using Taxonomies in JAXR Clients 194
Running the Client Examples 199
Before You Compile the Examples 200
Compiling the Examples 201
Running the Examples 201
Using JAXR Clients in Java EE Applications 206
Coding the Application Client: MyAppClient.java 207
Coding the PubQuery Session Bean 207
Editing the Properties File 207
Compiling the Source Files 208
Starting the Application Server 208
Creating JAXR Resources 208
Packaging the Application 209
Deploying the Application 210
Running the Application Client 210
Further Information 210

Chapter 7: Java XML Digital Signature API 213

How XWS-Security and XML Digital Signature API Are Related 214
XML Security Stack 215
Package Hierarchy 216
Service Providers 217
Introduction to XML Signatures 218
Example of an XML Signature 218
XML Digital Signature API Examples 221
validate Example 221
genenveloped Example 226

Chapter 8: Securing Web Services 231

Securing Web Service Endpoints 232
Overview of Message Security 232
Advantages of Message Security 233
Message Security Mechanisms 235
Web Services Security Initiatives and Organizations 236
W3C Specifications 236
OASIS Specifications 237
JCP Specifications 238
WS-I Specifications 239
Using Message Security with Java EE 241
Using the Application Server Message Security Implementation 242
Using the Java WSDP XWSS Security Implementation 247
Further Information 251

Index 253

About This Tutorial

THE Java™ Web Services Tutorial is a guide to developing Web applications with 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) of the Java WSDP, v2.0 with the Sun Java System Application Server Platform Edition 9 (hereafter called the Application Server). 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 who are interested in developing and deploying Web services and Web applications on the Sun Java System Application Server Platform Edition 9.

Prerequisites

Before proceeding with this tutorial you should have a good knowledge of the Java programming language. A good way to get to that point is to work through all the basic and some of the specialized trails in *The Java™ Tutorial*, Mary Campione et al., (Addison-Wesley, 2000). In particular, you should be familiar with relational database and security features described in the trails listed in Table 1.

Table 1 Prerequisite Trails in *The Java™ Tutorial*

Trail	**URL**
JDBC	`http://java.sun.com/docs/books/tutorial/jdbc`
Security	`http://java.sun.com/docs/books/tutorial/security1.2`

How to Use This Tutorial

The *Java Web Services Tutorial* addresses the following technology areas:

- The Java Architecture for XML Web Services (JAX-WS)
- The Java Architecture for XML Binding (JAXB)
- The StAX APIs and the Sun Java Streaming XML Parser implementation
- SOAP with Attachments API for Java (SAAJ)
- The Java Architecture for XML Registries
- XML Digital Signature
- Security in the Web Tier

All of the examples for this tutorial (except for the XML Digital Signature examples) are installed with the Java Web Services Tutorial for Java WSDP 2.0 bundle and can be found in the subdirectories of the `<jwstutorial20>/samples/<`*`technology`*`>` directory, where `<jwstutorial20>`.

About the Examples

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

Required Software

Java Web Services Tutorial Bundle

The example source for the technologies in this tutorial is contained in the Java Web Services Tutorial bundle. If you are viewing this online, you need to download the tutorial bundle from:

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

Application Server

Sun Java System Application Server Platform Edition 9 is the build and runtime environment for the tutorial examples. To build, deploy, and run the examples, you need a copy of the Application Server and the J2SE 5.0.

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. Most of the tutorial examples are distributed with a configuration file for `asant`, a portable build tool contained in the Application Server. This tool is an extension of the Ant tool developed by the Apache Software Foundation (`http://ant.apache.org`). The `asant` utility contains additional tasks that invoke the Application Server administration utility `asadmin`. Directions for building the examples are provided in each chapter.

Build properties and targets common to all the examples are specified in the files `<`*`INSTALL`*`>/jwstutorial13javaeetutorial5/examples/common/build.properties` and `<`*`INSTALL`*`>/jwstutorial13javaeetutorial5/examples/common/targets.xml`. Build properties and targets common to a particular technology are specified in the files `<`*`INSTALL`*`>/jwstutorial13javaeetutorial5/examples/`*`tech`*`/common/build.properties` and `<`*`INSTALL`*`>/jwstutorial13javaeetutorial5/examples/`*`tech`*`/common/targets.xml`.

To run the `asant` scripts, you must set common build properties in the file `<INSTALL>/javaeetutorial5/examples/common/build.properties` as follows:

- Set the `javaee.home` property to the location of your Application Server installation. The build process uses the `javaee.home` property to include the libraries in `<JAVAEE_HOME>/lib/` in the classpath. All examples that run on the Application Server include the Java EE library archive—`<JAVAEE_HOME>/lib/javaee.jar`—in the build classpath. Some examples use additional libraries in `<JAVAEE_HOME>/lib/`; the required libraries are enumerated in the individual technology chapters. `<JAVAEE_HOME>` refers to the directory where you have installed the Application Server.

Note: On Windows, you must escape any backslashes in the `javaee.home` property with another backslash or use forward slashes as a path separator. So, if your Application Server installation is `C:\Sun\AppServer`, you must set `javaee.home` as follows:

```
javaee.home = C:\\Sun\\AppServer
```

or

```
javaee.home=C:/Sun/AppServer
```

- Set the `javaee.tutorial.home` property to the location of your tutorial. This property is used for `asant` deployment and undeployment.

  For example, on UNIX:

  ```
  javaee.tutorial.home=/home/username/javaeetutorial5
  ```

  On Windows:

  ```
  javaee.tutorial.home=C:/javaeetutorial5
  ```

Do not install the tutorial to a location with spaces in the path.

- If you did not use the default value (`admin`) for the admin user, set the `admin.user` property to the value you specified when you installed the Application Server.
- If you did not use port 8080, set the `domain.resources.port` property to the value specified when you installed the Application Server.
- Set the admin user's password in `<INSTALL>/javaeetutorial5/examples/common/admin-password.txt` to the value you specified when you installed the Application Server. The format of this file is `AS_ADMIN_PASSWORD=`*`password`*. For example:

```
AS_ADMIN_PASSWORD=mypassword
```

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 2 lists the typographical conventions used in this tutorial.

Table 2 Typographical Conventions

Font Style	**Uses**
italic	Emphasis, titles, first occurrence of terms
`monospace`	URLs, code examples, file names, path names, tool names, application names, programming language keywords, tag, interface, class, method, and field names, properties
`italic monospace`	Variables in code, file paths, and URLs

Table 2 Typographical Conventions

Font Style	**Uses**
`<`*`italic monospace`*`>`	User-selected file path components

Feedback

Please send comments, broken link reports, errors, suggestions, and questions about this tutorial to the tutorial team at `users@jwsdp.dev.java.net`.

1

Building Web Services with JAX-WS

JAX-WS stands for Java API for XML Web Services. JAX-WS is a technology for building web services and clients that communicate using XML. JAX-WS allows developers to write message-oriented as well as RPC-oriented web services.

In JAX-WS, a remote procedure call is represented by an XML-based protocol such as SOAP. The SOAP specification defines the envelope structure, encoding rules, and conventions 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-WS 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-WS, the developer does not generate or parse SOAP messages. It is the JAX-WS runtime system that converts the API calls and responses to and from SOAP messages.

With JAX-WS, clients and web services have a big advantage: the platform independence of the Java programming language. In addition, JAX-WS is not restrictive: a JAX-WS client can access a web service that is not running on the Java platform, and vice versa. This flexibility is possible because JAX-WS 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.

Setting the Port

Several files in the JAX-WS examples depend on the port that you specified when you installed the Application Server. 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 JAX-WS examples:

- `<INSTALL>/javaeetutorial5/examples/jaxws/simpleclient/HelloClient.java`

Creating a Simple Web Service and Client with JAX-WS

This section shows how to build and deploy a simple web service and client. The source code for the service is in `<INSTALL>/javaeetutorial5/examples/jaxws/helloservice/` and the client is in `<INSTALL>/javaeetutorial5/examples/jaxws/simpleclient/`.

Figure 1–1 illustrates how JAX-WS technology manages communication between a web service and client.

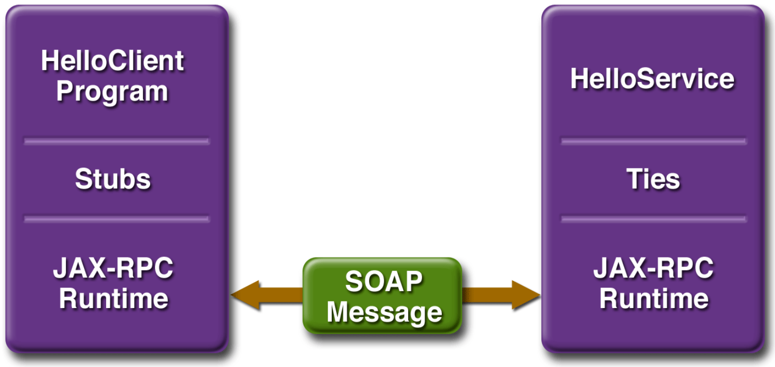

Figure 1–1 Communication Between a JAX-WS Web Service and a Client

The starting point for developing a JAX-WS web service is a Java class annotated with the `javax.jws.WebService` annotation. The `WebService` annotation defines the class as a web service endpoint.

A *service endpoint interface* (SEI) is a Java interface that declares the methods that a client can invoke on the service. An SEI is not required when building a JAX-WS endpoint. The web service implementation class implicitly defines a SEI.

You may specify an explicit SEI by adding the `endpointInterface` element to the `WebService` annotation in the implementation class. You must then provide a SEI that defines the public methods made available in the endpoint implementation class.

You use the endpoint implementation class and the `wsgen` tool to generate the web service artifacts and the stubs that connect a web service client to the JAX-WS runtime. For reference documentation on `wsgen`, see the Application Server man pages at `http://docs.sun.com/db/doc/817-6092`.

Together, the `wsgen` tool and the Application Server provide the Application Server's implementation of JAX-WS.

These are the basic steps for creating the web service and client:

1. Code the implementation class.
2. Compile the implementation class.
3. Use `wsgen` to generate the artifacts required to deploy the service.
4. Package the files into a WAR file.

5. Deploy the WAR file. The tie classes (which are used to communicate with clients) are generated by the Application Server during deployment.
6. Code the client class.
7. Use `wsimport` to generate and compile the stub files.
8. Compile the client class.
9. Run the client.

The sections that follow cover these steps in greater detail.

Requirements of a JAX-WS Endpoint

JAX-WS endpoints must follow these requirements:

- The implementing class must be annotated with either the `javax.jws.WebService` or `javax.jws.WebServiceProvider` annotation.
- The implementing class may explicitly reference an SEI through the `endpointInterface` element of the `@WebService` annotation, but is not required to do so. If no endpointInterface is not specified in `@WebService`, an SEI is implicityly defined for the implementing class.
- The business methods of the implementing class must be public, and must not be declared `static` or `final`.
- Business methods that are exposed to web service clients must be annotated with `javax.jws.WebMethod`.
- Business methods that are exposed to web service clients must have JAX-B-compatible parameters and return types. See Default Data Type Bindings (page 6).
- The implementing class must not be declared `final` and must not be `abstract`.
- The implementing class must have a default public constructor.
- The implementing class must not define the `finalize` method.
- The implementing class may use the `javax.annotation.PostConstruct` or `javax.annotation.PreDestroy` annotations on its methods for lifecycle event callbacks.

  The `@PostConstruct` method is called by the container before the implementing class begins responding to web service clients.

The `@PreDestroy` method is called by the container before the endpoint is removed from operation.

Coding the Service Endpoint Implementation Class

In this example, the implementation class, `Hello`, is annotated as a web service endpoint using the `@WebService` annotation. `Hello` declares a single method named `sayHello`, annotated with the `@WebMethod` annotation. `@WebMethod` exposes the annotated method to web service clients. `sayHello` returns a greeting to the client, using the name passed to `sayHello` to compose the greeting. The implementation class also must define a default, public, no-argument constructor.

```
package helloservice.endpoint;

import javax.jws.WebService;

@WebService()
public class Hello {
  private String message = new String("Hello, ");

  public void Hello() {}

  @WebMethod()
  public String sayHello(String name) {
     return message + name + ".";
  }
}
```

Building the Service

To build `HelloService`, in a terminal window go to the `<INSTALL>/javaeetutorial5/examples/jaxws/helloservice/` directory and type the following:

```
asant build
```

The `build` task command executes these `asant` subtasks:

- `compile-service`

The compile-service Task

This `asant` task compiles `Hello.java`, writing the class files to the `build` subdirectory. It then calls the `wsgen` tool to generate JAX-WS portable artifacts used by the web service. The equivalent command-line command is as follows:

```
wsgen -d build -s build -classpath build
     helloservice.endpoint.Hello
```

The `-d` flag specifies the output location of generated class files. The `-s` flag specifies the output location of generated source files. The `-classpath` flag specifies the location of the input files, in this case the endpoint implmentation class, `helloservice.endpoint.Hello`.

Packaging and Deploying the Service

You package and deploy the service using `asant`.

Upon deployment, the Application Server and the JAX-WS runtime generate any additional artifacts required for web service invocation, including the WSDL file.

Packaging and Deploying the Service with asant

To package and deploy the `helloservice` example, follow these steps:

1. In a terminal window, go to `<`*`INSTALL`*`>/javaeetutorial5/examples/jaxws/helloservice/`.
2. Run `asant create-war`.
3. Make sure the Application Server is started.
4. Set your admin username and password in `<`*`INSTALL`*`>/javaeetutorial5/examples/common/build.properties`.
5. Run `asant deploy`.

You can view the WSDL file of the deployed service by requesting the URL `http://localhost:8080/helloservice/hello?wsdl` in a web browser. Now you are ready to create a client that accesses this service.

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:

```
asant undeploy
```

Testing the Service Without a Client

The Application Server Admin Console allows you to test the methods of a web service endpoint. To test the `sayHello` method of `HelloService`, do the following:

1. Open the Admin Console by opening the following URL in a web browser:

   http://localhost:4848/
2. Enter the admin username and password to log in to the Admin Console.
3. Click Web Services in the left pane of the Admin Console.
4. Click `Hello`.
5. Click Test.
6. Under Methods, enter a name as the parameter to the `sayHello` method.
7. Click the `sayHello` button.

   This will take you to the `sayHello` Method invocation page.
8. Under Method returned, you'll see the response from the endpoint.

A Simple JAX-WS Client

`HelloClient` is a stand-alone Java program that accesses the `sayHello` method of `HelloService`. It makes this call through a stub, a local object that acts as a proxy for the remote service. The stub is created at development time by the wsimport tool, which generates JAX-WS portable artifacts based on a WSDL file.

Coding the Client

When invoking the remote methods on the stub, the client performs these steps:

1. Uses the `javax.xml.ws.WebServiceRef` annotation to declare a reference to a web service. `WebServiceRef` uses the `wsdlLocation` element to specify the URI of the deployed service's WSDL file.

```
@WebServiceRef(wsdlLocation="http://localhost:8080/
     helloservice/hello?wsdl")
static HelloService service;
```

2. Retrieves a proxy to the service, also known as a port, by invoking `getHelloPort` on the service.

   ```
   Hello port = service.getHelloPort();
   ```

   The port implements the SEI defined by the service.

3. Invokes the port's sayHello method, passing to the service a name.

   ```
   String response = port.sayHello(name);
   ```

Here's the full source of HelloClient, located in the `<INSTALL>/javaeetutorial5/examples/jaxws/simpleclient/src/` directory.

```
package simpleclient;

import javax.xml.ws.WebServiceRef;
import helloservice.endpoint.HelloService;
import helloservice.endpoint.Hello;

public class HelloClient {
  @WebServiceRef(wsdlLocation="http://localhost:8080/
       helloservice/hello?wsdl")
  static HelloService service;

  public static void main(String[] args) {
    try {
      HelloClient client = new HelloClient();
      client.doTest(args);
    } catch(Exception e) {
      e.printStackTrace();
    }
  }

  public void doTest(String[] args) {
    try {
      System.out.println("Retrieving the port from
           the following service: " + service);
      Hello port = service.getHelloPort();
      System.out.println("Invoking the sayHello operation
           on the port.");

      String name;
      if (args.length > 0) {
        name = args[0];
      } else {
```

```
                name = "No Name";
            }

            String response = port.sayHello(name);
            System.out.println(response);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Building and Running the Client

To build the client, you must first have deployed `HelloServiceApp`, as described in "Packaging and Deploying the Service with asant (page xx)." Then navigate to `<`*`JAVA_EE_HOME`*`>/examples/jaxws/simpleclient/` and do the following:

```
asant build
```

The run the client, do the following:

```
asant run
```

Types Supported by JAX-WS

JAX-WS delegates the mapping of Java programming language types to and from XML definitions to JAXB. Application developers don't need to know the details of these mappings, but they should be aware that not every class in the Java language can be used as a method parameter or return type in JAX-WS. For information on which types are supported by JAXB, see Default Data Type Bindings (page 6).

Web Services Interoperability and JAX-WS

JAX-WS 2.0 supports the Web Services Interoperability (WS-I) Basic Profile Version 1.1. 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 xxiv).

To support WS-I Basic Profile Version 1.1, JAX-WS has the following features:

- The JAX-WS runtime supports doc/literal and rpc/literal encodings for services, static stubs, dynamic proxies, and DII.

Further Information

For more information about JAX-WS and related technologies, refer to the following:

- Java API for XML Web Services 2.0 specification
  `https://jax-ws.dev.java.net/spec-download.html`
- JAX-WS home
  `https://jax-ws.dev.java.net/`
- Simple Object Access Protocol (SOAP) 1.2 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 1.1
  `http://www.ws-i.org`

2

Binding between XML Schema and Java Classes

THE Java™ Architecture for XML Binding (JAXB) provides a fast and convenient way to bind between XML schemas and 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. JAXB also provides a way generate XML schema from Java objects.

This chapter describes the JAXB architecture, functions, and core concepts. You should read this chapter before proceeding to Chapter 3, 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.

Architectural Overview

Figure 2–1 shows the components that make up a JAXB implementation.

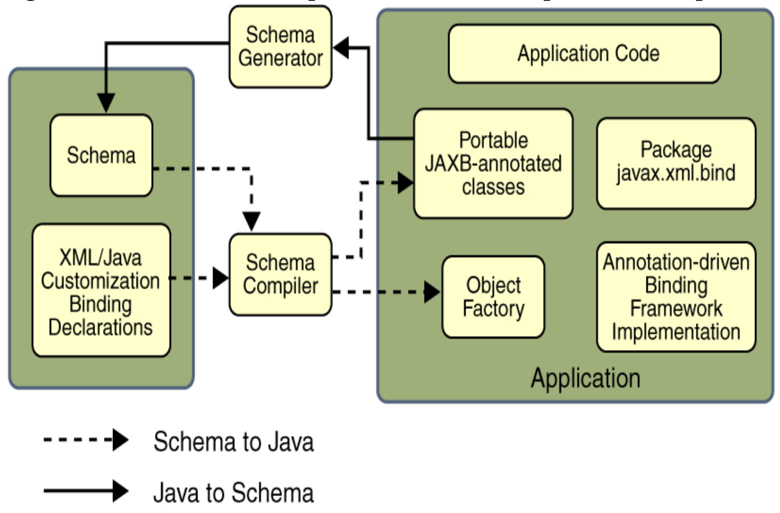

Figure 2–1 JAXB Architectural Overview

A JAXB implementation consists of the following architectural components:

- **schema compiler**: binds a source schema to a set of schema derived program elements. The binding is described by an XML-based binding language.
- **schema generator**: maps a set of existing program elements to a derived schema. The mapping is described by program annotations.
- **binding runtime framework**: provides unmarshalling (reading) and marshalling (writing) operations for accessing, manipulating and validating XML content using either schema-derived or existing program elements.

The JAXB Binding Process

Figure 2–2 shows what occurs during the JAXB binding process.

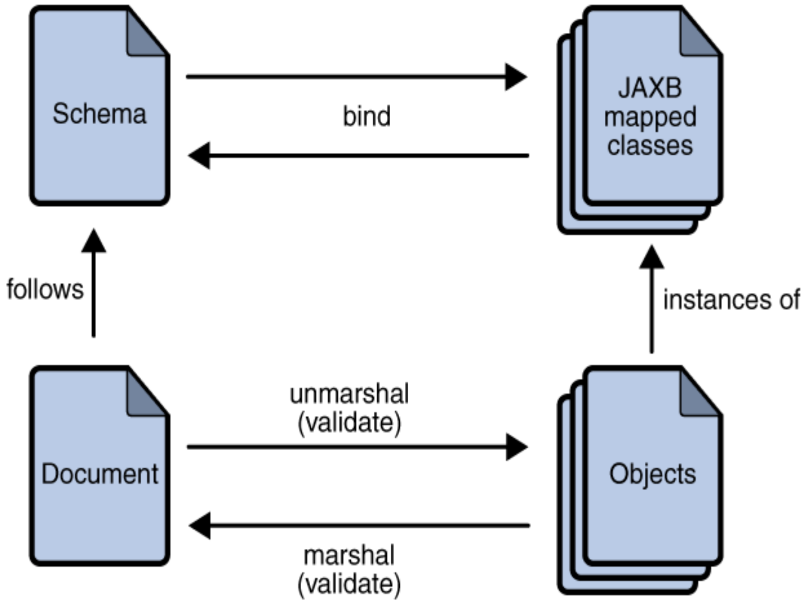

Figure 2–2 Steps in the JAXB Binding Process

Take steps from 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.

More About Unmarshalling

Unmarshlling provides a client application the ability to convert XML data into JAXB-derived Java objects.

More About Marshalling

Marshalling provides a client application the ability to convert a JAXB-derived Java object tree back into XML data.

By default, the `Marshaller` uses UTF-8 encoding when generating XML data.

Client applications are not required to validate the Java content tree before marshalling. 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.

More About Validation

Validation is the process of verifying that an XML document meets all the constraints expressed in the schema. JAXB 1.0 provided validation at unmarshal time and also enabled on-demand validation on a JAXB content tree. JAXB 2.0 only allows validation at unmarshal and marshal time. A web service processing model is to be lax in reading in data and strict on writing it out. To meet that model, validation was added to marshal time so one could confirm that they did not invalidate the XML document when modifying the document in JAXB form.

Representing XML Content

This section describes how JAXB represents XML content as Java objects.

Java Representation of XML Schema

JAXB supports the grouping of generated classes in Java packages. A package comprises:

- A Java class name is derived from the XML element name, or specified by a binding customization.
- An `ObjectFactory` class is a factory that is used to return instances of a bound Java class.

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. 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

Schema-to-Java

The Java language provides a richer set of data type than XML schema. Table 2–1 lists the mapping of XML data types to Java data types in JAXB.

Table 2–1 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`	`javax.xml.datatype.XMLGregorianCalendar`
`xsd:base64Binary`	`byte[]`
`xsd:hexBinary`	`byte[]`
`xsd:unsignedInt`	`long`
`xsd:unsignedShort`	`int`
`xsd:unsignedByte`	`short`
`xsd:time`	`javax.xml.datatype.XMLGregorianCalendar`

Table 2–1 JAXB Mapping of XML Schema Built-in Data Types (Continued)

XML Schema Type	Java Data Type
`xsd:date`	`javax.xml.datatype.XMLGregorianCalendar`
`xsd:g`	`javax.xml.datatype.XMLGregorianCalendar`
`xsd:anySimpleType`	`java.lang.Object`
`xsd:anySimpleType`	`java.lang.String`
`xsd:duration`	`javax.xml.datatype.Duration`
`xsd:NOTATION`	`javax.xml.namespace.QName`

JAXBElement

When XML element information can not be inferred by the derived Java representation of the XML content, a JAXBElement object is provided. This object has methods for getting and setting the object name and object value.

Java-to-Schema

Table 2–2 shows the default mapping of Java classes to XML data types.

Table 2–2 JAXB Mapping of XML Data Types to Java classes.

Java Class	XML Data Type
`java.lang.String`	`xs:string`
`java.math.BigInteger`	`xs:integer`
`java.math.BigDecimal`	`xs:decimal`
`java.util.Calendar`	`xs:dateTime`
`java.util.Date`	`xs:dateTime`

Table 2–2 JAXB Mapping of XML Data Types to Java classes. (Continued)

Java Class	XML Data Type
`javax.xml.namespace.QName`	`xs:QName`
`java.net.URI`	`xs:string`
`javax.xml.datatype.XMLGregorian-Calendar`	`xs:anySimpleType`
`javax.xml.datatype.Duration`	`xs:duration`
`java.lang.Object`	`xs:anyType`
`java.awt.Image`	`xs:base64Binary`
`javax.activa-tion.DataHandler`	`xs:base64Binary`
`javax.xml.trans-form.Source`	`xs:base64Binary`
`java.util.UUID`	`xs:string`

Customizing JAXB Bindings

Schema-to-Java

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.

JAXB provides two ways to customize an XML schema:

- As inline annotations in a source XML schema
- As declarations in an external binding customizations file that is passed to the JAXB binding compiler

Code examples showing how to customize JAXB bindings are provided in Chapter 3.

Java-to-Schema

XML schema that is generated from Java objects can be customized with JAXB annotations.

3

Using JAXB

THIS chapter provides instructions for using the sample Java applications that are included in the `<javaee.tutorial.home>/examples/jaxb` directory. 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 35) for instructions on using Customize examples that demonstrate how to modify the default JAXB bindings. Finally, the Java-to-Schema examples show how to use annotations to map Java classes to XML schema.

Note: The Purchase Order schema, `po.xsd`, and the Purchase Order XML file, `po.xml`, used in the Basic and Customize 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 using the Ant build tool, and provides details about the default schema-to-JAXB bindings used in these examples.

Description

This chapter describes three sets of examples:

- The Basic examples (Unmarshal Read, Modify Marshal, Unmarshal Validate, Pull Parser) demonstrate basic JAXB concepts like ummarshalling, marshalling, validating XML content, and parsing XML data.
- The Customize examples (Customize Inline, Datatype Converter, External Customize, Fix Collides) demonstrate various ways of customizing the binding of XML schemas to Java objects.
- The Java-to-Schema examples show how to use annotations to map Java classes to XML schema.

The Basic and Customize examples are based on a *Purchase Order* scenario. With the exception of the Fix Collides example, each uses an XML document, `po.xml`, written against an XML schema, `po.xsd`.

Table 3–1 briefly describes the Basic examples.

Table 3–1 Basic JAXB Examples

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.
Unmarshal Validate Example	Demonstrates how to enable validation during unmarshalling.
Pull Parser Example	Demonstrates how to use the StAX pull parser to parse a portion of an XML document.

Table 3–2 briefly describes the Customize examples.

Table 3–2 Customize JAXB Examples

Example Name	Description
Customize Inline Example	Demonstrates how to customize the default JAXB bindings by using inline annotations in an XML schema.
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. You should first move `binding.xjb`, the binding file, out of 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 `asant` alone uses the binding customizations to resolve the name conflicts while compiling the schema.

Table 3–3 briefly describes the Java-to-Schema examples.

Table 3–3 Java-toSchema JAXB Examples

Example Name	**Description**
j2s-create-marshal	Illustrates how to marshal and unmarshal JAXB-annotated classes to XML schema. The example also shows how to enable JAXP 1.3 validation at unmarshal time using a schema file that was generated from the JAXB mapped classes.
j2s-xmlAccessorOrder	Illustrates how to use the @XmlAccessorOrder and @XmlType.propOrder mapping annotations in Java classes to control the order in which XML content is marshalled/unmarshaled by a Java type.
j2s-xmlAdapter-field	Illustrates how to use the interface `XmlAdapter` and the annotation @XmlJavaTypeAdapter to provide a a custom mapping of XML content into and out of a HashMap (field) that uses an “int” as the key and a “string” as the value.
j2s-xmlAttribute-field	Illustrates how to use the annotation @XmlAttribute to define a property or field to be handled as an XML attribute.
j2s-xmlRootElement	Illustrates how to use the annotation @XmlRootElement to define an XML element name for the XML schema type of the corresponding class.
j2s-xmlSchemaType-class	Illustrates how to use the annotation @XmlSchemaType to customize the mapping of a property or field to an XML built-in type.
j2s-xmlType	Illustrates how to use the annotation @XmlType to map a class or enum type to an XML schema type.

Each Basic and Customize 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 Fix Collides example uses `example.xsd` rather than `po.xsd`.
- `po.xml` is the *Purchase Order* XML file containing sample XML content, and is the file you will unmarshal into a Java content tree in each example. This file is almost exactly the same in each example, with minor content

differences to highlight different JAXB concepts. Note that the Fix Collides example uses `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. Use Ant to generate, compile, and run the schema-derived JAXB 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 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 perform these steps by using `asant` with the `build.xml` project file included in each example directory.

Configuring and Running the Samples

The `build.xml` file included in each example directory is an asant project file that, when run, automatically performs the following steps:

1. Updates your `CLASSPATH` to include the necessary schema-derived JAXB classes.
2. For the Basic and Customize examples, 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`. For the Java-to-Schema examples runs `schemagen`, the schema generator, to generate XML schema from the annotated Java classes.

3. Compiles the schema-derived JAXB classes or the annotated Java code.
4. Runs the `Main` class for the 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 19). The methods used for building and processing the Java content tree are described in Basic Examples (page 29).

JAXB Compiler Options

The JAXB XJC schema binding compiler transforms, or binds, a source XML schema to a set of JAXB content classes in the Java programming language. The compiler, `xjc`, is provided in two flavors in the Application Server: `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 3–4.

Table 3–4 `xjc` Command-Line Options

Option or Argument	Description
`-nv`	Do not perform strict validation of the input schema(s). By default, `xjc` performs strict validation of the source schema before processing. Note that this does not mean the binding compiler will not perform any validation; it simply means that it will perform less-strict validation.
`-extension`	By default, the XJC binding compiler strictly enforces the rules outlined in the Compatibility chapter of the JAXB Specification. 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 will be allowed to use the JAXB Vendor Extensions.

Table 3–4 `xjc` Command-Line Options (Continued)

Option or Argument	Description
`-b` *`file`*	Specify one or more external binding files to process. (Each binding file must have its 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. In addition, the ordering of the 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` *`package`*	Specify an alternate output directory. By default, the XJC binding compiler will generate the Java content classes in the current directory. The output directory must already exist; the XJC binding compiler will not create it for you.
`-proxy` *`proxy`*	Specify the HTTP/HTTPS proxy. The format is `[user[:password]@]proxyHost[:proxyPort]`. The old `-host` and `-port` options are still supported by the Reference Implementation for backwards compatibility, but they have been deprecated.
`-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. For more information, please read the XML Entity and URI Resolvers document or examine the catalog-resolver sample application.
`-readOnly`	Force the XJC binding compiler to mark the generated Java sources read-only. By default, the XJC binding compiler does not write-protect the Java source files it generates.
`-npa`	Supress the generation of package level annotations into `**/package-info.java`. Using this switch causes the generated code to internalize those annotations into the other generated classes.
`-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.

Table 3–4 `xjc` Command-Line Options (Continued)

Option or Argument	Description
`-quiet`	Suppress compiler output, such as progress information and warnings.
`-help`	Display a brief summary of the compiler switches.
`-version`	Display the compiler version information.
`-Xlocator`	Enable source location support for generated code.
`-Xsync-methods`	Generate accessor methods with the `synchronized` keyword.
`-mark-generated`	Mark the generated code with the `-@javax.annotation.Generated` annotation.

JAXB Schema Generator Options

The JAXB Schema Generator, `schemagen`, creates a schema file for each namespace referenced in your Java classes. The schema generator can be launched using the appropriate schemagen shell script in the `bin` directory for your platform. The schema generator processes Java source files only. If your Java sources reference other classes, those sources must be accessible from your system CLASSPATH environment variable or errors will occur when the schema is generated. There is no way to control the name of the generated schema files.

You can display quick usage instructions by invoking the scripts without any options, or with the `-help` switch. The syntax is as follows:

```
schemagen [-options ...] [java_source_files]
```

The `schemagen` command-line options are listed in Table 3–5.

Table 3–5 `schemagen` Command-Line Options

Option or Argument	Description
`-d` *`path`*	Specifies the location of the processor- and javac generated class files.

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, Unmarshal Validate), 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 3–6 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`

Table 3–6 Schema-Derived JAXB Classes in the Basic Examples (Continued)

Class	Description
`primer/po/impl/PurchaseOrderType-Impl.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 3–7 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`

Table 3–7 Schema-to-Java Bindings for the Basic Examples (Continued)

XML Schema	**JAXB Binding**
`<xsd:complexType name="USAddress">`	
`  <xsd:sequence>`	
`    <xsd:element name="name" type="xsd:string"/>`	
`    <xsd:element name="street" type="xsd:string"/>`	
`    <xsd:element name="city" type="xsd:string"/>`	
`    <xsd:element name="state" type="xsd:string"/>`	`USAddress.java`
`    <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>`	`Items.java`
`    <xsd:element name="item" minOccurs="1" maxOc-`	
`curs="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>`	`Items.ItemType`
`           </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>`	

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 the Basic examples (Unmarshal Read, Modify Marshal, Unmarshal Validate) 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>/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>/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>
```

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 4).

1. The `<INSTALL>/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
```

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 29), 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 16).

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 3–7.

Figure 3–1 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.

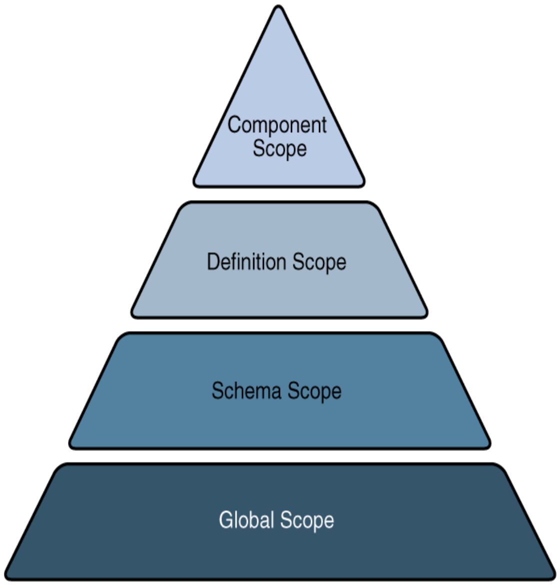

Figure 3–1 Customization Scope Inheritance and Precedence

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 47) 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 45) for more information.

`<globalBindings>` declarations are only valid in the `annotation` element of the top-level `schema` element. There can only be a single instance of a `<globalBindings>` declaration in any given schema or binding declarations file. If one source schema includes or imports a second source schema, the `<globalBindings>` declaration must be declared in the first source schema.

Schema Binding Declarations

Schema scope customizations are declared with `<schemaBindings>`. The syntax for schema scope customizations is:

```
<schemaBindings>
  [ <package> package </package> ]
  [ <nameXmlTransform> ... </nameXmlTransform> ]*
</schemaBindings>

<package [ name = "packageName" ]
  [ <javadoc> ... </javadoc> ]
</package>

<nameXmlTransform>
  [ <typeName [ suffix="suffix" ]
              [ prefix="prefix" ] /> ]
  [ <elementName [ suffix="suffix" ]
                 [ prefix="prefix" ] /> ]
  [ <modelGroupName [ suffix="suffix" ]
                    [ prefix="prefix" ] /> ]
  [ <anonymousTypeName [ suffix="suffix" ]
                       [ prefix="prefix" ] /> ]
</nameXmlTransform>
```

As shown above, `<schemaBinding>` declarations include two subcomponents:

- `<package>...</package>` specifies the name of the package and, if desired, the location of the API documentation for the schema-derived classes.

- `<nameXmlTransform>...</nameXmlTransform>` specifies customizations to be applied.

Class Binding Declarations

The `<class>` binding declaration enables you to customize the binding of a schema element to a Java content interface or a Java `Element` interface. `<class>` declarations can be used to customize:

- A name for a schema-derived Java interface
- An implementation class for a schema-derived Java content interface.

The syntax for `<class>` customizations is:

```
<class [ name = "className"]
   [ implClass= "implClass" ] >
   [ <javadoc> ... </javadoc> ]
</class>
```

- `name` is the name of the derived Java interface. It must be a legal Java interface name and must not contain a package prefix. The package prefix is inherited from the current value of package.
- `implClass` is the name of the implementation class for `className` and must include the complete package name.
- The `<javadoc>` element specifies the Javadoc tool annotations for the schema-derived Java interface. The string entered here must use `CDATA` or < to escape embedded HTML tags.

Property Binding Declarations

The `<property>` binding declaration enables you to customize the binding of an XML schema element to its Java representation as a property. The scope of customization can either be at the definition level or component level depending upon where the `<property>` binding declaration is specified.

The syntax for `<property>` customizations is:

```
<property[ name = "propertyName"]
   [ collectionType = "propertyCollectionType" ]
   [ fixedAttributeAsConstantProperty = "true" | "false" | "1" | "0" ]
   [ generateIsSetMethod = "true" | "false" | "1" | "0" ]
   [ enableFailFastCheck ="true" | "false" | "1" | "0" ]
   [ <baseType> ... </baseType> ]
   [ <javadoc> ... </javadoc> ]
</property>
```

```
<baseType>
   <javaType> ... </javaType>
</baseType>
```

- `name` defines the customization value `propertyName`; it must be a legal Java identifier.
- `collectionType` defines the customization value `propertyCollection-Type`, which is the collection type for the property. `propertyCollection-Type` if specified, can be either `indexed` or any fully-qualified class name that implements `java.util.List`.
- `fixedAttributeAsConstantProperty` defines the customization value `fixedAttributeAsConstantProperty`. The value can be either `true`, `false`, `1`, or `0`.
- `generateIsSetMethod` defines the customization value of `generateIs-SetMethod`. The value can be either `true`, `false`, `1`, or `0`.
- `enableFailFastCheck` defines the customization value `enableFail-FastCheck`. The value can be either `true`, `false`, `1`, or `0`. Please note that the JAXB implementation does not support failfast validation.
- `<javadoc>` customizes the Javadoc tool annotations for the property's getter method.

<javaType> Binding Declarations

The `<javaType>` declaration provides a way to customize the translation of XML datatypes to and from Java datatypes. XML provides more datatypes than Java, and so the `<javaType>` declaration lets you specify custom datatype bindings when the default JAXB binding cannot sufficiently represent your schema.

The target Java datatype can be a Java built-in datatype or an application-specific Java datatype. If an application-specific datatype is used as the target, your implementation must also provide parse and print methods for unmarshalling and marshalling data. To this end, the JAXB specification supports a `parseMethod` and `printMethod`:

- The `parseMethod` is called during unmarshalling to convert a string from the input document into a value of the target Java datatype.
- The `printMethod` is called during marshalling to convert a value of the target type into a lexical representation.

If you prefer to define your own datatype conversions, JAXB defines a static class, DatatypeConverter, to assist in the parsing and printing of valid lexical representations of the XML Schema built-in datatypes.

The syntax for the <javaType> customization is:

```
<javaType name= "javaType"
      [ xmlType= "xmlType" ]
      [ hasNsContext = “true” | “false” ]
      [ parseMethod= "parseMethod" ]
      [ printMethod= "printMethod" ]>
```

- name is the Java datatype to which xmlType is to be bound.
- xmlType is the name of the XML Schema datatype to which javaType is to bound; this attribute is required when the parent of the <javaType> declaration is <globalBindings>.
- parseMethod is the name of the parse method to be called during unmarshalling.
- printMethod is the name of the print method to be called during marshalling.
- hasNsContext allows a namespace context to be specified as a second parameter to a print or a parse method; can be either true, false, 1, or 0. By default, this attribute is false, and in most cases you will not need to change it.

The <javaType> declaration can be used in:

- A <globalBindings> declaration
- An annotation element for simple type definitions, GlobalBindings, and <basetype> declarations.
- A <property> declaration.

See MyDatatypeConverter Class (page 54) 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 42).

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 40).

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 `<`*`JAVA_HOME`*`>/jaxb/samples/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 `enable-FailFastCheck` 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 &lt;b>Purchase Order&lt;/b> consists of addresses and items.`" The `&lt;` is used to escape the opening bracket on the `<b>` HTML tags.

Note: When a `<class>` customization is specified in the `appinfo` element of a `complexType` definition, as it is here, the `complexType` definition is bound to a Java content interface.

Later in `po.xsd`, another `<javadoc>` customization is declared at this class level, but this time the HTML string is escaped with `CDATA`:

```
<xsd:annotation>
 <xsd:appinfo>
    <jxb:class>
      <jxb:javadoc>
     <![CDATA[ First line of documentation for a
<b>USAddress</b>.]]>
```

```
          </jxb:javadoc>
        </jxb:class>
      </xsd:appinfo>
      </xsd:annotation>
```

Note: If you want to include HTML markup tags in a `<jaxb:javadoc>` customization, you must enclose the data within a `CDATA` section or escape all left angle brackets using `&lt;`. See *XML 1.0 2nd Edition* for more information (`http://www.w3.org/TR/2000/REC-xml-20001006#sec-cdata-sect`).

Property Binding Declarations

Of particular interest here is the `generateIsSetMethod` customization, which causes two additional property methods, `isSetQuantity` and `unsetQuantity`, to be generated. These methods enable a client application to distinguish between schema default values and values occurring explicitly within an instance document.

For example, in `po.xsd`:

```
<xsd:complexType name="Items">
   <xsd:sequence>
      <xsd:element name="item" minOccurs="1"
maxOccurs="unbounded">
         <xsd:complexType>
            <xsd:sequence>
           <xsd:element name="productName" type="xsd:string"/>
            <xsd:element name="quantity" default="10">
            <xsd:annotation>
               <xsd:appinfo>
                  <jxb:property generateIsSetMethod="true"/>
               </xsd:appinfo>
            </xsd:annotation>
         .
         .
         .
         </xsd:complexType>
      </xsd:element>
   </xsd:sequence>
</xsd:complexType>
```

The `@generateIsSetMethod` applies to the `quantity` element, which is bound to a property within the `Items.ItemType` interface. `unsetQuantity` and `isSetQuantity` methods are generated in the `Items.ItemType` interface.

MyDatatypeConverter Class

The *<INSTALL>*/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>/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 56), 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 56) 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 remove the `binding` parameter to the `xjc` task in the `build.xml` file in the `<INSTALL>/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 Running the ant Task Without Using a Binding Declarations File
- The binding.xjb Declarations File
- Resolving the Conflicts in example.xsd

The example.xsd Schema

The XML schema, `<INSTALL>/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 Running the ant Task Without Using a Binding Declarations File

Here is the output that is returned if you run the `ant` task in the *<INSTALL>*`/examples/jaxb/fix-collides` directory without specifying the `binding` parameter to the `xjc` task in the `build.xml` file:

```
[echo] Compiling the schema w/o external binding file
(name collision errors expected)...
[xjc] Compiling file:/C:/javaeetutorial5/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>*`/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>
```

If you add the binding parameter you removed back to the xjc task in the build.xml file and then run ant in the <*INSTALL*>/examples/jaxb/fix-collides directory, the customizations in binding.xjb will be passed 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`*`>/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`*`>/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`.

Java-toSchema Examples

The Java-to-Schema examples show how to use annotations to map Java classes to XML schema.

j2s-create-marshal Example

The j2s-create-marhal example illustrates Java to schema databinding. It demonstrates marshalling and unmarshalling of JAXB annotated classes. The example also shows how to enable JAXP 1.3 validation at unmarshal time using a schema file that was generated from the JAXB mapped classes.

The schema file, `bc.xsd`, was generated with the following commands:

```
% schemagen src/cardfile/*.java
% cp schema1.xsd bc.xsd
```

Note that `schema1.xsd`, was copied to `bc.xsd`; `schemagen` does not allow you to specify a schema name of your choice.

j2s-xmlAccessorOrder Example

The j2s-xmlAccessorOrder example shows how to use the @XmlAccessorOrder and @XmlType.propOrder annotations to dictate the order in which XML content is marshalled/unmarshalled by a Java type.

Java-to-Schema maps a JavaBean's properties and fields to an XML Schema type. The class elements are mapped to either an XML Schema complex type or an XML Schema simple type. The default element order for a generated schema type is currently unspecified because Java reflection does not impose a return order. The lack of reliable element ordering negatively impacts application portability. You can use two annotations, @XmlAccessorOrder and @XmlType.propOrder, to define schema element ordering for applications that need to be portable across JAXB Providers.

The @XmlAccessorOrder annotation imposes one of two element ordering algorithms, AccessorOrder.UNDEFINED or AccessorOrder.ALPHABETICAL. AccessorOrder.UNDEFINED is the default setting. The order is dependent on the system's reflection implementation. AccessorOrder.ALPHABETICAL orders the elements in lexicographic order as determined by `java.lang.String.CompareTo(String anotherString)`.

You can define the @XmlAccessorOrder annotation for annotation type ElementType.PACKAGE on a class object. When the @XmlAccessorOrder annotation is defined on a package, the scope of the formatting rule is active for every class in the package.

When defined on a class, the rule is active on the contents of that class.

There can be multiple @XmlAccessorOrder annotations within a package. The order of precedence is the innermost (class) annotation takes precedence over the outer annotation. For example, if @XmlAccessorOrder(AccessorOrder.ALPHABETICAL) is defined on a package and @XmlAccessorOrder(AccessorOrder.UNDEFINED) is defined on a class in that package, the contents of the generated schema type for the class would be in an unspecified order and the contents of the generated schema type for evey other class in the package would be alphabetical order.

The @XmlType annotation can be defined for a class. The annotation element propOrder() in the @XmlType annotation allows you to specify the content order in the generated schema type. When you use the @XmlType.propOrder annotation on a class to specify content order, all public properties and public fields in the class must be specified in the parameter list. Any public property or field that you want to keep out of the parameter list must be annotated with @XmlAttribute or @XmlTransient.

The default content order for @XmlType.propOrder is {} or {""}, not active. In such cases, the active @XmlAccessorOrder annotation takes precedence. When class content order is specified by the @XmlType.propOrder annotation, it takes precedence over any active @XmlAccessorOrder annotation on the class or package. If the @XmlAccessorOrder and @XmlType.propOrder(A, B, ...) annotations are specified on a class, the propOrder always takes precedence regardless of the order of the annontation statements. For example, in the code below, the @XmlAccessorOrder annotation precedes the @XmlType.propOrder annotation.

```
@XmlAccessorOrder(AccessorOrder.ALPHABETICAL)
@XmlType(propOrder={"name", "city"})
public class USAddress {
        :
    public String getCity() {return city;}
    public void setCity(String city) {this.city = city;}

    public String getName() {return name;}
    public void setName(String name) {this.name = name;}
        :
}
```

In the code below, the @XmlType.propOrder annotation precedes the @XmlAccessorOrder annotation.

```
@XmlType(propOrder={"name", "city"})
@XmlAccessorOrder(AccessorOrder.ALPHABETICAL)
public class USAddress {
        :
    public String getCity() {return city;}
    public void setCity(String city) {this.city = city;}

    public String getName() {return name;}
    public void setName(String name) {this.name = name;}
        :
}
```

In both scenarios, propOrder takes precedence and the identical schema content shown below will be generated.

```
<xs:complexType name="usAddress">
   <xs:sequence>
      <xs:element name="name" type="xs:string" minOccurs="0"/>
      <xs:element name="city" type="xs:string" minOccurs="0"/>
   </xs:sequence>
</xs:complexType>
```

The purchase order code example demonstrates the affects of schema content ordering using the @XmlAccessorOrder annotation at the package and class level, and the @XmlType.propOrder annotation on a class.

Class `package-info.java` defines @XmlAccessorOrder to be ALPHABETICAL for the package. The public fields `shipTo` and `billTo` in class `Purchase-`

`OrderType` will be affected in the generated schema content order by this rule. Class `USAddress` defines the @XmlType.propOrder annotation on the class. This demonstates user-defined property order superseding ALPHABETICAL order in the generated schema.

The generated schema file can be found in directory `schemas`.

j2s-xmlAdapter-field Example

The j2s-xmlAdapter-field example demonstrates how to use the `XmlAdapter` interface and the @XmlJavaTypeAdapter annotation to provide a custom mapping of XML content into and out of a HashMap (field) that uses an "int" as the key and a "string" as the value.

Interface `XmlAdapter` and annotation @XmlJavaTypeAdapter are used for special processing of datatypes during unmarshalling/marshalling. There are a variety of XML datatypes for which the representation does not map easily into Java (for example, `xs:DateTime` and `xs:Duration`), and Java types which do not map conveniently into XML representations, for example implementations of `java.util.Collection` (such as `List`) and `java.util.Map` (such as `HashMap`) or for non-JavaBean classes. It is for these cases that

The `XmlAdapter` interface and the @XmlJavaTypeAdapter annotation are provided for cases such as these. This combination provides a portable mechanism for reading/writing XML content into and out of Java applications.

The `XmlAdapter` interface defines the methods for data reading/writing.

```
/*
 *  ValueType - Java class that provides an XML representation
 *             of the data. It is the object that is used for
 *              marshalling and unmarshalling.
 *
 *  BoundType - Java class that is used to process XML content.
 */

public abstract class XmlAdapter<ValueType,BoundType> {
    // Do-nothing constructor for the derived classes.
    protected XmlAdapter() {}

    // Convert a value type to a bound type.
    public abstract BoundType unmarshal(ValueType v);

    // Convert a bound type to a value type.
    public abstract ValueType marshal(BoundType v);
}
```

You can use the @XmlJavaTypeAdapter annotation to associate a particular `XmlAdapter` implementation with a `Target` type, PACKAGE, FIELD, METHOD, TYPE, or PARAMETER.

The j2s-xmlAdapter-field example demonstrates an XmlAdapter for mapping XML content into and out of a (custom) `HashMap`. The `HashMap` object, `basket`, in class `KitchenWorldBasket`, uses a key of type “int” and a value of type “String”. We want these datatypes to be reflected in the XML content that is read and written. The XML content should look like this.

```
<basket>
     <entry key="9027">glasstop stove in black</entry>
     <entry key="288">wooden spoon</entry>
</basket>
```

The default schema generated for Java type `HashMap` does not reflect the desired format.

```
<xs:element name="basket">
   <xs:complexType>
     <xs:sequence>
       <xs:element name="entry" minOccurs="0"
           maxOccurs="unbounded">
         <xs:complexType>
           <xs:sequence>
```

```
                <xs:element name="key" minOccurs="0"
                    type="xs:anyType"/>
                <xs:element name="value" minOccurs="0"
                    type="xs:anyType"/>
             </xs:sequence>
           </xs:complexType>
         </xs:element>
       </xs:sequence>
     </xs:complexType>
   </xs:element>
```

In the default HashMap schema, key and value are both elements and are of datatype `anyType`. The XML content will look like this:

```
<basket>
     <entry>
        <key>9027</>
        <value>glasstop stove in black</>
     </entry>
     <entry>
        <key>288</>
        <value>wooden spoon</>
    </entry>
</basket>
```

To resolve this issue, we wrote two Java classes, `PurchaseList` and `PartEntry`, that reflect the needed schema format for unmarshalling/marshalling the content. The XML schema generated for these classes is as follows:

```
<xs:complexType name="PurchaseListType">
    <xs:sequence>
        <xs:element name="entry" type="partEntry"
            nillable="true" maxOccurs="unbounded"
            minOccurs="0"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="partEntry">
    <xs:simpleContent>
        <xs:extension base="xs:string">
            <xs:attribute name="key" type="xs:int"
                use="required"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>
```

Class AdapterPurchaseListToHashMap implements the XmlAdapter interface. In class KitchenWorldBasket, the @XmlJavaTypeAdapter annotation is used to pair AdapterPurchaseListToHashMap with field HashMap basket. This pairing will cause the marshal/unmarshal method of AdapterPurchaseListToHashMap to be called for any corresponding marshal/unmarshal action on KitchenWorldBasket.

j2s-xmlAttribute-field Example

The j2s-xmlAttribute-field example shows how to use the @XmlAttribute annotation to define a property or field to be treated as an XML attribute.

The @XmlAttribute annotation maps a field or JavaBean property to an XML attribute. The following rules are imposed:

- A static final field is mapped to a XML fixed attribute.
- When the field or property is a collection type, the items of the collection type must map to a schema simple type.
- When the field or property is other than a collection type, the type must map to a schema simple type.

When following the JavaBean programming paradigm, a property is defined by a “get” and “set” prefix on a field name.

```
int zip;
public int getZip(){return zip;}
public void setZip(int z){zip=z;}
```

Within a bean class, you have the choice of setting the @XmlAttribute annotation on one of three components: the field, the setter method, or the getter method. If you set the @XmlAttribute annotation on the field, the setter method will need to be renamed or there will be a naming conflict at compile time. If you set the @XmlAttribute annotationt on one of the methods, it must be set on either the setter or getter method, but not on both.

The j2s-xmlAttribute-field example shows how to use the @XmlAttribute annotationd on a static final field, on a field rather than on one of the corresponding bean methods, on a bean property (method), and on a field that is other than a collection type. In class USAddress, fields, country, and zip are tagged as attributes. The setZip method was disabled to avoid the compile error. Property state was tagged as an attribute on the setter method. You could have used the getter method instead. In class PurchaseOrderType, field cCardVendor is a non-collection type. It meets the requirment of being a simple type; it is an enum type.

j2s-xmlRootElement Example

The j2s-xmlRootElement example demonstrates the use of the @XmlRootElement annotation to define an XML element name for the XML schema type of the corresponding class.

The @XmlRootElement annotation maps a class or an enum type to an XML element. At least one element definition is needed for each top-level Java type used for unmarshalling/marshalling. If there is no element definition, there is no starting location for XML content processing.

The @XmlRootElement annotation uses the class name as the default element name. You can change the default name by using the annotation attribute `name`. If you do, the specified name will then be used as the element name and the type name. It is common schema practice for the element and type names to be different. You can use the @XmlType annotation to set the element type name.

The namespace attribute of the @XmlRootElement annotation is used to define a namespace for the element.

j2s-xmlSchemaType-class Example

The j2s-XmlSchemaType-class example demonstrates the use of the annotation @XmlSchemaType to customize the mapping of a property or field to an XML built-in type.

The @XmlSchemaType annotation can be used to map a Java type to one of the XML built-in types. This annotation is most useful in mapping a Java type to one of the nine date/time primitive datatypes.

When the @XmlSchemaType annotation is defined at the package level, the identification requires both the XML built-in type name and the corresponding Java type class. A @XmlSchemaType definition on a field or property takes precedence over a package definition.

The j2s-XmlSchemaType-clasexample shows how to use the @XmlSchemaType annotation at the package level, on a field and on a property. File `TrackingOrder` has two fields, `orderDate` and `deliveryDate`, which are defined to be of type `XMLGregorianCalendar`. The generated schema will define these elements to be of XML built-in type `gMonthDay`. This relationship was defined on the package in the file `package-info.java`. Field `shipDate` in file `TrackingOrder` is also defined to be of type `XMLGregorianCalendar`, but the @XmlSchemaType annotation statements override the package definition and specify the field to be of type `date`. Property method `getTrackingDuration` defines the schema element to be defined as primitive type `duration` and not Java type `String`.

j2s-xmlType Example

The j2s-xmlType example demonstrates the use of annotation @XmlType. Annotation @XmlType maps a class or an enum type to a XML Schema type.

A class must have either a public zero arg constructor or a static zero arg factory method in order to be mapped by this annotation. One of these methods is used during unmarshalling to create an instance of the class. The factory method may reside within in a factory class or the existing class. There is an order of presedence as to which method is used for unmarshalling.

- If a factory class is identified in the annotation, a corresponding factory method in that class must also be identified and that method will be used.
- If a factory method is identified in the annotation but no factory class is identified, the factory method must reside in the current class. The factory method is used even if there is a public zero arg constructor method present.
- If no factory method is identified in the annotation, the class must contain a public zero arg constructor method.

In this example a factory class provides zero arg factory methods for several classes. The @XmlType annotation on class `OrderContext` references the factory class. The unmarshaller will use the identified factory method in this class.

```
public class OrderFormsFactory {
    public OrderContext newOrderInstance() {
        return new OrderContext()
    }

    public PurchaseOrderType newPurchaseOrderType() {
        return new newPurchaseOrderType();
    }
}

@XmlType(name="oContext", factoryClass="OrderFormsFactory",
factoryMethod="newOrderInstance")
public class OrderContext {
    public OrderContext(){ ..... }
}
```

In this example, a factory method is defined in a class, which also contains a standard class constructure. Because the `factoryMethod` value is defined and no `factoryClass` is defined, the factory method `newOrderInstance` is used during unmarshalling.

```
@XmlType(name="oContext", factoryMethod="newOrderInstance")
 public class OrderContext {

    public OrderContext(){ ..... }

    public OrderContext newOrderInstance() {
        return new OrderContext();
    }
}
```

4 Streaming API for XML

This chapter focuses on the Streaming API for XML (StAX), a streaming Java-based, event-driven, pull-parsing API for reading and writing XML documents. StAX enables you to create bidrectional XML parsers that are fast, relatively easy to program, and have a light memory footprint.

StAX provides is the latest API in the JAXP family, and provides an alternative to SAX, DOM, TrAX, and DOM for developers looking to do high-performance stream filtering, processing, and modification, particularly with low memory and limited extensibility requirements.

Note: To synopsize, StAX provides a standard, bidirectional *pull parser* interface for streaming XML processing, offering a simpler programming model than SAX and more efficient memory management than DOM. StAX enables developers to parse and modify XML streams as events, and to extend XML information models to allow application-specific additions. More detailed comparisons of StAX with several alternative APIs are provided below, in “Comparing StAX to Other JAXP APIs.”

Why StAX?

The StAX project was spearheaded by BEA with support from Sun Microsystems, and the JSR 173 specification passed the Java Community Process final approval ballot in March, 2004 (`http://jcp.org/en/jsr/detail?id=173`). The primary goal of the StAX API is to give “parsing control to the programmer by exposing a simple iterator based API. This allows the programmer to ask for the next event (pull the event) and allows state to be stored in procedural fashion." StAX was created to address limitations in the two most prevalent parsing APIs, SAX and DOM.

Streaming Versus DOM

Generally speaking, there are two programming models for working with XML infosets: document *streaming* and the *document object model* (DOM).

The *DOM* model involves creating in-memory objects representing an entire document tree and the complete infoset state for an XML document. Once in memory, DOM trees can be navigated freely and parsed arbitrarily, and as such provide maximum flexibility for developers. However the cost of this flexibility is a potentially large memory footprint and significant processor requirements, as the entire representation of the document must be held in memory as objects for the duration of the document processing. This may not be an issue when working with small documents, but memory and processor requirements can escalate quickly with document size.

Streaming refers to a programming model in which XML infosets are transmitted and parsed serially at application runtime, often in real time, and often from dynamic sources whose contents are not precisely known beforehand. Moreover, stream-based parsers can start generating output immediately, and infoset elements can be discarded and garbage collected immediately after they are used. While providing a smaller memory footprint, reduced processor requirements, and higher performance in certain situations, the primary trade-off with stream processing is that you can only see the infoset state at one location at a time in the document. You are essentially limited to the "cardboard tube" view of a document, the implication being that you need to know what processing you want to do before reading the XML document.

Streaming models for XML processing are particularly useful when your application has strict memory limitations, as with a cellphone running J2ME, or when your application needs to simultaneously process several requests, as with an application server. In fact, it can be argued that the majority of XML business logic can benefit from stream processing, and does not require the in-memory maintenance of entire DOM trees.

Pull Parsing Versus Push Parsing

Streaming *pull parsing* refers to a programming model in which a client application calls methods on an XML parsing library when it needs to interact with an XML infoset—that is, the client only gets (pulls) XML data when it explicitly asks for it.

Streaming *push parsing* refers to a programming model in which an XML parser sends (pushes) XML data to the client as the parser encounters elements in an XML infoset—that is, the parser sends the data whether or not the client is ready to use it at that time.

Pull parsing provides several advantages over push parsing when working with XML streams:

- With pull parsing, the client controls the application thread, and can call methods on the parser when needed. By contrast, with push processing, the parser controls the application thread, and the client can only accept invocations from the parser.
- Pull parsing libraries can be much smaller and the client code to interact with those libraries much simpler than with push libraries, even for more complex documents.
- Pull clients can read multiple documents at one time with a single thread.
- A StAX pull parser can filter XML documents such that elements unnecessary to the client can be ignored, and it can support XML views of non-XML data.

StAX Use Cases

The StAX specification defines a number of uses cases for the API:

- **Data binding**
  - Unmarshalling an XML document
  - Marshalling an XML document
  - Parallel document processing
  - Wireless communication
- **SOAP message processing**
  - Parsing simple predictable structures
  - Parsing graph representations with forward references

    - Parsing WSDL
- **Virtual data sources**
    - Viewing as XML data stored in databases
    - Viewing data in Java objects created by XML data binding
    - Navigating a DOM tree as a stream of events
- **Parsing specific XML vocabularies**
- **Pipelined XML processing**

A complete discussion of all these use cases is beyond the scope of this chapter. Please refer to the StAX specification for further information.

Comparing StAX to Other JAXP APIs

As an API in the JAXP family, StAX can be compared, among other APIs, to SAX, TrAX, and JDOM. Of the latter two, StAX is not as powerful or flexible as TrAX or JDOM, but neither does it require as much memory or processor load to be useful, and StAX can, in many cases, outperform the DOM-based APIs. The same arguments outlined above, weighing the cost/benefits of the DOM model versus the streaming model, apply here.

With this in mind, the closest comparisons between can be made between StAX and SAX, and it is here that StAX offers features that are beneficial in many cases; some of these include:

- StAX-enabled clients are generally easier to code than SAX clients. While it can be argued that SAX parsers are marginally easier to write, StAX parser code can be smaller and the code necessary for the client to interact with the parser simpler.
- StAX is a bidirectional API, meaning that it can both read and write XML documents. SAX is read only, so another API is needed if you want to write XML documents.
- SAX is a push API, whereas StAX is pull. The trade-offs between push and pull APIs outlined above apply here.

Table 4–1 synopsizes the comparative features of StAX, SAX, DOM, and TrAX (table adapted from "Does StAX Belong in Your XML Toolbox?" (`http://www.developer.com/xml/article.php/3397691`) by Jeff Ryan).

Table 4–1 XML Parser API Feature Summary

Feature	StAX	SAX	DOM	TrAX
API Type	Pull, streaming	Push, streaming	In memory tree	XSLT Rule
Ease of Use	High	Medium	High	Medium
XPath Capability	No	No	Yes	Yes
CPU and Memory Efficiency	Good	Good	Varies	Varies
Forward Only	Yes	Yes	No	No
Read XML	Yes	Yes	Yes	Yes
Write XML	Yes	No	Yes	Yes
Create, Read, Update, Delete	No	No	Yes	No

StAX API

The StAX API exposes methods for iterative, event-based processing of XML documents. XML documents are treated as a filtered series of events, and infoset states can be stored in a procedural fashion. Moreover, unlike SAX, the StAX API is bidirectional, enabling both reading and writing of XML documents.

The StAX API is really two distinct API sets: a *cursor* API and an *iterator* API. These two API sets explained in greater detail later in this chapter, but their main features are briefly described below.

Cursor API

As the name implies, the StAX *cursor* API represents a cursor with which you can walk an XML document from beginning to end. This cursor can point to one thing at a time, and always moves forward, never backward, usually one infoset element at a time.

The two main cursor interfaces are `XMLStreamReader` and `XMLStreamWriter`. `XMLStreamReader` includes accessor methods for all possible information retrievable from the XML Information model, including document encoding, element names, attributes, namespaces, text nodes, start tags, comments, processing instructions, document boundaries, and so forth; for example:

```
public interface XMLStreamReader {
  public int next() throws XMLStreamException;
  public boolean hasNext() throws XMLStreamException;
  public String getText();
  public String getLocalName();
  public String getNamespaceURI();
  // ... other methods not shown
}
```

You can call methods on `XMLStreamReader`, such as `getText` and `getName`, to get data at the current cursor location. `XMLStreamWriter` provides methods that correspond to `StartElement` and `EndElement` event types; for example:

```
public interface XMLStreamWriter {
  public void writeStartElement(String localName) \
    throws XMLStreamException;
  public void writeEndElement() \
    throws XMLStreamException;
  public void writeCharacters(String text) \
    throws XMLStreamException;
// ... other methods not shown
}
```

The cursor API mirrors SAX in many ways. For example, methods are available for directly accessing string and character information, and integer indexes can be used to access attribute and namespace information. As with SAX, the cursor API methods return XML information as strings, which minimizes object allocation requirements.

Iterator API

The StAX *iterator* API represents an XML document stream as a set of discrete event objects. These events are pulled by the application and provided by the parser in the order in which they are read in the source XML document.

The base iterator interface is called `XMLEvent`, and there are subinterfaces for each event type listed in Table 4–2, below. The primary parser interface for reading iterator events is `XMLEventReader`, and the primary interface for writing iterator events is `XMLEventWriter`. The `XMLEventReader` interface contains five methods, the most important of which is `nextEvent()`, which returns the next event in an XML stream. `XMLEventReader` implements `java.util.Iterator`, which means that returns from `XMLEventReader` can be cached or passed into routines that can work with the standard Java Iterator; for example:

```
public interface XMLEventReader extends Iterator {
  public XMLEvent nextEvent() throws XMLStreamException;
  public boolean hasNext();
  public XMLEvent peek() throws XMLStreamException;
  ...
}
```

Similarly, on the output side of the iterator API, you have:

```
public interface XMLEventWriter {
  public void flush() throws XMLStreamException;
  public void close() throws XMLStreamException;
  public void add(XMLEvent e) throws XMLStreamException;
  public void add(Attribute attribute) \
    throws XMLStreamException;
  ...
}
```

Iterator Event Types

Table 4–2 lists the thirteen `XMLEvent` types defined in the event iterator API.

Table 4–2 `XMLEvent` Types

Event Type	Description
`StartDocument`	Reports the beginning of a set of XML events, including encoding, XML version, and standalone properties.
`StartElement`	Reports the start of an element, including any attributes and namespace declarations; also provides access to the prefix, namespace URI, and local name of the start tag.
`EndElement`	Reports the end tag of an element. Namespaces that have gone out of scope can be recalled here if they have been explicitly set on their corresponding `StartElement`.

Table 4–2 `XMLEvent` Types (Continued)

Event Type	**Description**
`Characters`	Corresponds to XML `CData` sections and `CharacterData` entities. Note that ignorable whitespace and significant whitespace are also reported as `Character` events.
`EntityRef-erence`	Character entities can be reported as discrete events, which an application developer can then choose to resolve or pass through unresolved. By default, entities are resolved. Alternatively, if you do not want to report the entity as an event, replacement text can be substituted and reported as `Characters`.
`Processin-gInstruc-tion`	Reports the target and data for an underlying processing instruction.
`Comment`	Returns the text of a comment
`EndDocument`	Reports the end of a set of XML events.
`DTD`	Reports as `java.lang.String` information about the DTD, if any, associated with the stream, and provides a method for returning custom objects found in the DTD.
`Attribute`	Attributes are generally reported as part of a `StartElement` event. However, there are times when it is desirable to return an attribute as a standalone `Attribute` event; for example, when a namespace is returned as the result of an `XQuery` or `XPath` expression.
`Namespace`	As with attributes, namespaces are usually reported as part of a `StartElement`, but there are times when it is desirable to report a namespace as a discrete `Namespace` event.

Note that the `DTD`, `EntityDeclaration`, `EntityReference`, `NotationDeclaration`, and `ProcessingInstruction` events are only created if the document being processed contains a DTD.

Sample Event Mapping

As an example of how the event iterator API maps an XML stream, consider the following XML document:

```
<?xml version="1.0"?>
<BookCatalogue xmlns="http://www.publishing.org">
  <Book>
     <Title>Yogasana Vijnana: the Science of Yoga</Title>
     <ISBN>81-40-34319-4</ISBN>
     <Cost currency="INR">11.50</Cost>
  </Book>
</BookCatalogue>
```

This document would be parsed into eighteen primary and secondary events, as shown below. Note that secondary events, shown in curly braces ({}), are typically accessed from a primary event rather than directly.

Table 4–3 Sample Iterator API Event Mapping

#	Element/Attribute	Event
1	`version="1.0"`	`StartDocument`

2	`isCData = false`
	`data = “\n”`
	`IsWhiteSpace = true`
		`Characters`

3	`qname = BookCatalogue:http://www.publishing.org`
	`attributes = null`
	`namespaces = {BookCatalogue” -> http://www.publishing.org”}`
		`StartElement`

4	`qname = Book`
	`attributes = null`
	`namespaces = null`
		`StartElement`

5	`qname = Title`
	`attributes = null`
	`namespaces = null`
		`StartElement`

6	`isCData = false`
	`data = “Yogasana Vijnana: the Science of Yoga\n\t”`
	`IsWhiteSpace = false`
		`Characters`

7	`qname = Title`
	`namespaces = null`
		`EndElement`

Table 4–3 Sample Iterator API Event Mapping (Continued)

#	Element/Attribute	Event

8	`qname = ISBN`
	`attributes = null`
	`namespaces = null`
		`StartElement`

9	`isCData = false`
	`data = “81-40-34319-4\n\t”`
	`IsWhiteSpace = false`
		`Characters`

10	`qname = ISBN`
	`namespaces = null`
		`EndElement`

11	`qname = Cost`
	`attributes = {“currency” -> INR}`
	`namespaces = null`
		`StartElement`

12	`isCData = false`
	`data = “11.50\n\t”`
	`IsWhiteSpace = false`
		`Characters`

13	`qname = Cost`
	`namespaces = null`
		`EndElement`

14	`isCData = false`
	`data = “\n”`
	`IsWhiteSpace = true`
		`Characters`

15	`qname = Book`
	`namespaces = null`
		`EndElement`

16	`isCData = false`
	`data = “\n”`
	`IsWhiteSpace = true`
		`Characters`

17	`qname = BookCatalogue:http://www.publishing.org`
	`namespaces = {BookCatalogue” -> http://www.publishing.org”}`
		`EndElement`

18		`EndDocument`

There are several important things to note in the above example:

- The events are created in the order in which the corresponding XML elements are encountered in the document, including nesting of elements, opening and closing of elements, attribute order, document start and document end, and so forth.

- As with proper XML syntax, all container elements have corresponding start and end events; for example, every `StartElement` has a corresponding `EndElement`, even for empty elements.
- `Attribute` events are treated as secondary events, and are accessed from their corresponding `StartElement` event.
- Similar to `Attribute` events, `Namespace` events are treated as secondary, but appear twice and are accessible twice in the event stream, first from their corresponding `StartElement` and then from their corresponding `EndElement`.
- `Character` events are specified for all elements, even if those elements have no character data. Similarly, `Character` events can be split across events.
- The StAX parser maintains a namespace stack, which holds information about all XML namespaces defined for the current element and its ancestors. The namespace stack is exposed through the `javax.xml.namespace.NamespaceContext` interface, and can be accessed by namespace prefix or URI.

Choosing Between Cursor and Iterator APIs

It is reasonable to ask at this point, “What API should I choose? Should I create instances of `XMLStreamReader` or `XMLEventReader`? Why are there two kinds of APIs anyway?”

Development Goals

The authors of the StAX specification targeted three types of developers:

- **Library and infrastructure developers** – Create application servers, JAXM, JAXB, JAX-RPC and similar implementations; need highly efficient, low-level APIs with minimal extensibility requirements.
- **J2ME developers** – Need small, simple, pull-parsing libraries, and have minimal extensibility needs.

- **J2EE and J2SE developers** – Need clean, efficient pull-parsing libraries, plus need the flexibility to both read and write XML streams, create new event types, and extend XML document elements and attributes.

Given these wide-ranging development categories, the StAX authors felt it was more useful to define two small, efficient APIs rather than overloading one larger and necessarily more complex API.

Comparing Cursor and Iterator APIs

Before choosing between the cursor and iterator APIs, you should note a few things that you can do with the iterator API that you cannot do with cursor API:

- Objects created from the `XMLEvent` subclasses are immutable, and can be used in arrays, lists, and maps, and can be passed through your applications even after the parser has moved on to subsequent events.
- You can create subtypes of `XMLEvent` that are either completely new information items or extensions of existing items but with additional methods.
- You can add and remove events from an XML event stream in much simpler ways than with the cursor API.

Similarly, keep some general recommendations in mind when making your choice:

- If you are programming for a particularly memory-constrained environment, like J2ME, you can make smaller, more efficient code with the cursor API.
- If performance is your highest priority—for example, when creating low-level libraries or infrastructure—the cursor API is more efficient.
- If you want to create XML processing pipelines, use the iterator API.
- If you want to modify the event stream, use the iterator API.
- If you want to your application to be able to handle pluggable processing of the event stream, use the iterator API.
- In general, if you do not have a strong preference one way or the other, using the iterator API is recommended because it is more flexible and extensible, thereby “future-proofing” your applications.

Using StAX

In general, StAX programmers create XML stream readers, writers, and events by using the `XMLInputFactory`, `XMLOutputFactory` and `XMLEventFactory` classes. Configuration is done by setting properties on the factories, whereby implementation-specific settings can be passed to the underlying implementation using the `setProperty()` method on the factories. Similarly, implementation-specific settings can be queried using the `getProperty()` factory method.

The `XMLInputFactory`, `XMLOutputFactory` and `XMLEventFactory` classes are described below, followed by discussions of resource allocation, namespace and attribute management, error handling, and then finally reading and writing streams using the cursor and iterator APIs.

StAX Factory Classes

XMLInputFactory

The `XMLInputFactory` class lets you configure implementation instances of XML stream reader processors created by the factory. New instances of the abstract class `XMLInputFactory` are created by calling the `newInstance()` method on the class. The static method `XMLInputFactory.newInstance()` is then used to create a new factory instance.

Deriving from JAXP, the `XMLInputFactory.newInstance()` method determines the specific `XMLInputFactory` implementation class to load by using the following lookup procedure:

1. Use the `javax.xml.stream.XMLInputFactory` system property.
2. Use the `lib/xml.stream.properties` file in the JRE directory.
3. Use the Services API, if available, to determine the classname by looking in the `META-INF/services/javax.xml.stream.XMLInputFactory` files in jars available to the JRE.
4. Use the platform default `XMLInputFactory` instance.

After getting a reference to an appropriate XMLInputFactory, an application can use the factory to configure and create stream instances. Table 4–4 lists the properties supported by `XMLInputFactory`. See the StAX specification for a more detailed listing.

Table 4–4 XMLInputFactory Properties

Property	**Description**
javax.xml.stream.isValidating	Turns on implementation specific validation.
javax.xml.stream.isCoalescing	*(Required)* Requires the processor to coalesce adjacent character data.
javax.xml.stream.isNamespaceAware	Turns off namespace support. All implementations must support namespaces supporting non-namespace aware documents is optional.
javax.xml.stream.isReplacingEntityReferences	*(Required)* Requires the processor to replace internal entity references with their replacement value and report them as characters or the set of events that describe the entity.
javax.xml.stream.isSupportingExternalEntities	*(Required)* Requires the processor to resolve external parsed entities.
javax.xml.stream.reporter	*(Required)* Sets and gets the implementation of the XMLReporter
javax.xml.stream.resolver	*(Required)* Sets and gets the implementation of the XMLResolver interface
javax.xml.stream.allocator	*(Required)* Sets/gets the implementation of the XMLEventAllocator interface

XMLOutputFactory

New instances of the abstract class `XMLOutputFactory` are created by calling the `newInstance()` method on the class. The static method `XMLOutputFactory.newInstance()` is then used to create a new factory instance. The algorithm used to obtain the instance is the same as for `XMLInputFactory` but references the `javax.xml.stream.XMLOutputFactory` system property.

`XMLOutputFactory` supports only one property, `javax.xml.stream.isRepairingNamespaces`. This property is required, and its purpose is to create default prefixes and associate them with Namespace URIs. See the StAX specification for a more information.

XMLEventFactory

New instances of the abstract class `XMLEventFactory` are created by calling the `newInstance()` method on the class. The static method `XMLEventFactory.newInstance()` is then used to create a new factory instance. This factory references the `javax.xml.stream.XMLEventFactory` property to instantiate the factory. The algorithm used to obtain the instance is the same as for `XMLInputFactory` and `XMLOutputFactory` but references the `javax.xml.stream.XMLEventFactory` system property.

There are no default properties for `XMLEventFactory`.

Resources, Namespaces, and Errors

The StAX specification handles resource allocation, attributes and namespace, and errors and exceptions as described below.

Resource Resolution

The `XMLResolver` interface provides a means to set the method that resolves resources during XML processing. An application sets the interface on `XMLInputFactory`, which then sets the interface on all processors created by that factory instance.

Attributes and Namespaces

Attributes are reported by a StAX processor using lookup methods and strings in the cursor interface and `Attribute` and `Namespace` events in the iterator interface. Note here that namespaces are treated as attributes, although namespaces are reported separately from attributes in both the cursor and iterator APIs. Note also that namespace processing is optional for StAX processors. See the StAX specification for complete information about namespace binding and optional namespace processing.

Error Reporting and Exception Handling

All fatal errors are reported by way of `javax.xml.stream.XMLStreamException`. All nonfatal errors and warnings are reported using the `javax.xml.stream.XMLReporter` interface.

Reading XML Streams

As described earlier in this chapter, the way you read XML streams with a StAX processor—and more importantly, what you get back—varies significantly depending on whether you are using the StAX cursor API or the event iterator API. The following two sections describe how to read XML streams with each of these APIs.

Using XMLStreamReader

The `XMLStreamReader` interface in the StAX cursor API lets you read XML streams or documents in a forward direction only, one item in the infoset at a time. The following methods are available for pulling data from the stream or skipping unwanted events:

- Get the value of an attribute
- Read XML content
- Determine whether an element has content or is empty
- Get indexed access to a collection of attributes
- Get indexed access to a collection of namespaces
- Get the name of the current event (if applicable)
- Get the content of the current event (if applicable)

Instances of `XMLStreamReader` have at any one time a single current event on which its methods operate. When you create an instance of `XMLStreamReader` on a stream, the initial current event is the `START_DOCUMENT` state.The `XMLStreamReader.next()` method can then be used to step to the next event in the stream.

Reading Properties, Attributes, and Namespaces

The `XMLStreamReader.next()` method loads the properties of the next event in the stream. You can then access those properties by calling the `XMLStreamReader.getLocalName()` and `XMLStreamReader.getText()` methods.

When the `XMLStreamReader` cursor is over a `StartElement` event, it reads the name and any attributes for the event, including the namespace. All attributes for an event can be accessed using an index value, and can also be looked up by namespace URI and local name. Note, however, that only the namespaces declared on the current `StartEvent` are available; previously declared namespaces are not maintained, and redeclared namespaces are not removed.

XMLStreamReader Methods

`XMLStreamReader` provides the following methods for retrieving information about namespaces and attributes:

```
int getAttributeCount();
String getAttributeNamespace(int index);
String getAttributeLocalName(int index);
String getAttributePrefix(int index);
String getAttributeType(int index);
String getAttributeValue(int index);
String getAttributeValue(String namespaceUri,String
localName);
boolean isAttributeSpecified(int index);
```

Namespaces can also be accessed using three additional methods:

```
int getNamespaceCount();
String getNamespacePrefix(int index);
String getNamespaceURI(int index);
```

Instantiating an XMLStreamReader

This example, taken from the StAX specification, shows how to instantiate an input factory, create a reader, and iterate over the elements of an XML stream:

```
XMLInputFactory f = XMLInputFactory.newInstance();
XMLStreamReader r = f.createXMLStreamReader( ... );
while(r.hasNext()) {
  r.next();
}
```

Using XMLEventReader

The `XMLEventReader` API in the StAX event iterator API provides the means to map events in an XML stream to allocated event objects that can be freely reused, and the API itself can be extended to handle custom events.

XMLEventReader provides four methods for iteratively parsing XML streams:

- `next()` – Returns the next event in the stream
- `nextEvent()` – Returns the next typed XMLEvent
- `hasNext()` – Returns true if there are more events to process in the stream
- `peek()` – Returns the event but does not iterate to the next event

For example, the following code snippet illustrates the `XMLEventReader` method declarations:

```
package javax.xml.stream;
import java.util.Iterator;
public interface XMLEventReader extends Iterator {
  public Object next();
  public XMLEvent nextEvent() throws XMLStreamException;
  public boolean hasNext();
  public XMLEvent peek() throws XMLStreamException;
...
}
```

To read all events on a stream and then print them, you could use the following:

```
while(stream.hasNext()) {
XMLEvent event = stream.nextEvent();
System.out.print(event);
}
```

Reading Attributes

You can access attributes from their associated `javax.xml.stream.StartElement`, as follows:

```
public interface StartElement extends XMLEvent {
  public Attribute getAttributeByName(QName name);
  public Iterator getAttributes();
}
```

You can use the `getAttributes()` method on the `StartElement` interface to use an `Iterator` over all the attributes declared on that `StartElement`.

Reading Namespaces

Similar to reading attributes, namespaces are read using an `Iterator` created by calling the `getNamespaces()` method on the `StartElement` interface. Only the namespace for the current `StartElement` is returned, and an application can get the current namespace context by using `StartElement.getNamespaceContext()`.

Writing XML Streams

StAX is a bidirectional API, and both the cursor and event iterator APIs have their own set of interfaces for writing XML streams. As with the interfaces for reading streams, there are significant differences between the writer APIs for cursor and event iterator. The following sections describe how to write XML streams using each of these APIs.

Using XMLStreamWriter

The `XMLStreamWriter` interface in the StAX cursor API lets applications write back to an XML stream or create entirely new streams. XMLStreamWriter has methods that let you:

- Write well-formed XML
- Flush or close the output
- Write qualified names

Note that `XMLStreamWriter` implementations are not required to perform well-formedness or validity checks on input. While some implementations my perform strict error checking, others may not. The rules you choose to implement are set on properties provided by the `XMLOutputFactory` class.

The `writeCharacters(...)` method is used to escape characters such as &, <, >, and “. Binding prefixes can be handled by either passing the actual value for the prefix, by using the `setPrefix()` method, or by setting the property for defaulting namespace declarations.

The following example, taken from the StAX specification, shows how to instantiate an output factory, create a writer and write XML output:

```
XMLOutputFactory output = XMLOutputFactory.newInstance();
XMLStreamWriter writer = output.createXMLStreamWriter( ... );
writer.writeStartDocument();
writer.setPrefix("c","http://c");
writer.setDefaultNamespace("http://c");
writer.writeStartElement("http://c","a");
writer.writeAttribute("b","blah");
writer.writeNamespace("c","http://c");
writer.writeDefaultNamespace("http://c");
writer.setPrefix("d","http://c");
writer.writeEmptyElement("http://c","d");
writer.writeAttribute("http://c","chris","fry");
writer.writeNamespace("d","http://c");
writer.writeCharacters("foo bar foo");
writer.writeEndElement();
writer.flush();
```

This code generates the following XML (new lines are non-normative)

```
<?xml version='1.0' encoding='utf-8'?>
<a b="blah" xmlns:c="http://c" xmlns="http://c">
<d:d d:chris="fry" xmlns:d="http://c"/>foo bar foo</a>
```

Using XMLEventWriter

The `XMLEventWriter` interface in the StAX event iterator API lets applications write back to an XML stream or create entirely new streams. This API can be extended, but the main API is as follows:

```
public interface XMLEventWriter {
  public void flush() throws XMLStreamException;
  public void close() throws XMLStreamException;
  public void add(XMLEvent e) throws XMLStreamException;
  // ... other methods not shown.
}
```

Instances of `XMLEventWriter` are created by an instance of `XMLOutputFactory`. Stream events are added iteratively, and an event cannot be modified after it has been added to an event writer instance.

Attributes, Escaping Characters, Binding Prefixes

StAX implementations are required to buffer the last `StartElement` until an event other than `Attribute` or `Namespace` is added or encountered in the stream. This means that when you add an `Attribute` or a `Namespace` to a stream, it is appended the current `StartElement` event.

You can use the `Characters` method to escape characters like `&`, `<`, `>`, and “.

The `setPrefix(...)` method can be used to explicitly bind a prefix for use during output, and the `getPrefix(...)` method can be used to get the current prefix. Note that by default, `XMLEventWriter` adds namespace bindings to its internal namespace map. Prefixes go out of scope after the corresponding `EndElement` for the event in which they are bound.

Sun's Streaming Parser Implementation

The Sun Java System Application Server (SJSAS) PE 9.0 package includes Sun Microsystem's JSR 173 (StAX) implementation, called the Sun Java Streaming XML Parser (SJSXP). The SJSXP is a high-speed, non-validating, W3C XML 1.0 and Namespace 1.0-compliant streaming XML pull parser built upon the Xerces2 codebase.

In Sun's SJSXP implementation, the Xerces2 lower layers, particularly the Scanner and related classes, have been redesigned to behave in a pull fashion. In addition to the changes the lower layers, the SJSXP includes additional StAX-related functionality and many performance-enhancing improvements. The SJSXP is implemented in `appserv-ws.jar` and `javaee.jar`, both of which are located in the `<javaee.home>/lib` directory.

Included with this J2EE tutorial are StAX code samples, located in the `<javaee.tutorial.home>/examples/stax` directory, that illustrate how Sun's SJSXP implementation works. These samples are described in the Sample Code section, later in this chapter.

Before proceeding with the sample code, there are two important aspects of the SJSXP about which you should be aware:

- Reporting CDATA Events
- SJSXP Factories Implementation

These two topics are discussed below.

Reporting CDATA Events

The `javax.xml.stream.XMLStreamReader` implemented in the SJSXP does not report CDATA events. If you have an application that needs to receive such events, configure the `XMLInputFactory` to set the following implementation-specific "report-cdata-event" property:

```
XMLInputFactory factory = XMLInptuFactory.newInstance();
factory.setProperty("report-cdata-event", Boolean.TRUE);
```

SJSXP Factories Implementation

Most applications do not need to know the factory implementation class name. Just adding the javaee.jar and appserv-ws.jar files to the classpath is sufficient for most applications because these two jars supply the factory implementation classname for various SJSXP properties under the META-INF/services directory—for example, javax.xml.stream.XMLInputFactory, javax.xml.stream.XMLOutputFactory, and javax.xml.stream.XMLEventFactory—which is the third step of a lookup operation when an application asks for the factory instance. See the javadoc for the XMLInputFactory.newInstance() method for more information about the lookup mechanism.

However, there may be scenarios when an application would like to know about the factory implementation class name and set the property explicitly. These scenarios could include cases where there are multiple JSR 173 implementations in the classpath and the application wants to choose one, perhaps one that has superior performance, contains a crucial bug fix, or suchlike.

If an application sets the SystemProperty, it is the first step in a lookup operation, and so obtaining the factory instance would be fast compared to other options; for example:

```
javax.xml.stream.XMLInputFactory -->
com.sun.xml.stream.ZephyrParserFactory
javax.xml.stream.XMLOutputFactory -->
com.sun.xml.stream.ZephyrWriterFactor
javax.xml.stream.XMLEventFactory -->
com.sun.xml.stream.events.ZephyrEventFactory
```

Sample Code

This section steps through the sample StAX code included in the J2EE 1.4 Tutorial bundle. All sample directories used in this section are located off the `<javaee.tutorial.home>/examples/stax` directory.

The topics covered in this section are as follows:

- Sample Code Organization (page 99)
- Configuring Your Environment for Running the Samples (page 100)
- Running the Samples (page 101)
- Sample XML Document (page 102)
- cursor Sample – CursorParse.java (page 103)
- cursor2event Sample – CursorApproachEventObject.java (page 105)
- event Sample – EventParse.java (page 106)
- filter Sample – MyStreamFilter.java (page 109)
- readnwrite Sample – EventProducerConsumer.java (page 111)
- writer Sample – CursorWriter.java (page 114)

Sample Code Organization

There are seven StAX sample directories in `<javaee.tutorial.home>/examples/stax`:

- **common** contains a `build.properties` file and a `target.xml` file that are used commonly by all the StAX tutorial examples. There is also a `data` directory containing a sample XML document, `BookCatalog.xml`, that is used by all the StAX examples. The values in `common/build.properties` as well as all the StAX examples are inherited from a parent `build.properties` file in the `<javaee.tutorial.home>/examples/common` directory. Note that you should not need to modify the `build.properties` file in `stax/common/build.properties`.
- **cursor** contains `CursorParse.java`, which illustrates how to use the `XMLStreamReader` (cursor) API to read an XML file.
- **cursor2event** contains `CursorApproachEventObject.java`, which illustrates how an application can get information as an `XMLEvent` object when using cursor API.

- **event** contains `EventParse.java`, which illustrates how to use the `XMLEventReader` (event iterator) API to read an XML file.
- **filter** contains `MyStreamFilter.java`, which illustrates how to use the StAX Stream Filter APIs. In this example, the filter accepts only `StartElement` and `EndElement` events and filters out the remainder of the events.
- **readnwrite** contains `EventProducerConsumer.java`, which illustrates how the StAX producer/consumer mechanism can be used to simultaneously read and write XML streams.
- **writer** contains `CursorWriter.java`, which illustrates how to use `XMLStreamWriter` to write an XML file programatically.

Configuring Your Environment for Running the Samples

The instructions for configuring your environment are the same as those for running the other J2EE Tutorial samples. Specifically, to configure your environment for running the StAX examples, follow the steps below.

Note: If you are configuring the samples to run in a Microsoft Windows environment, use UNIX-style forward slashes (`/`) rather than Windows-style backslashes (`\`) to separate directory names when specifying paths in the steps below. For example, if your Application Server PE installation is in `c:\Sun\AppServer`, specify `c:/Sun/AppServer` instead.

1. Set the following two properties in `<javaee.tutorial.home>/examples/common/build.properties`:
   - **javaee.home** to the directory in which SJSAS PE 9.0 is installed
   - **javaee.tutorial.home** to the directory in which the J2EE 1.4 Tutorial is installed.
2. Specify the admin password used for your Application Server installation in `<javaee.tutorial.home>/examples/common/admin-password.txt`.

3. You may also need to specify the path to the `asant` command in your `PATH` environment variable; for example, on UNIX/Linux:

   ```
   export PATH=$PATH:/opt/SUNWappserver/bin/
   ```

   or, on Windows:

   ```
   set PATH=%PATH%;c:\Sun\AppServer\bin
   ```

   `asant` is a script wrapper around the implementation of Ant bundled with the J2EE 1.4 Tutorial. Be sure to use this Ant implementation when running the tutorial samples rather than any other Ant implementation you may have installed on your system. Also be sure to use the `asant` wrapper rather than running Ant directly.

4. Finally, note that the `build.xml` files in the various `stax` sample directories include classpath references to `<javaee.home>/lib/javaee.jar` and `<javaee.home>/lib/appserv-ws.jar`. You should not change these values, and if you create your own `build.xml` files, be sure to include these classpath references.

Running the Samples

The samples are run by means of the `asant` Ant wrapper and three build targets, defined in the `<javaee.tutorial.home>/stax/samples/build.xml` file. When you run any of the samples, the compiled class files are placed in a directory named `./build`. This directory is created if it does not exist already.

Note: As mentioned above, be sure to use the implementation of Ant bundled with the J2EE Tutorial rather than any version of Ant you may already have installed on your system. Also be sure to use the `asant` wrapper script rather than running Ant directly.

There is a separate `build.xml` file in each of the `stax` sample directories except `common`, and each `build.xml` file provides the same three targets. Switch to the directory containing the sample you want to run, and then run the desired `build.xml` target from there.

The three Ant targets defined in each of the StAX `build.xml` files are:

- **build** – Compile and run all classes
- **run** – Run the example
- **clean** – Clean all compiled files and sample directories when you are done

For example, to run the `cursor` example:

```
cd <javaee.tutorial.home>/examples/stax/cursor
asant build
asant run
```

Sample XML Document

The sample XML document, `BookCatalogue.xml`, used by most of the StAX sample classes is located in the `<javaee.tutorial.home>/examples/stax/common/data` directory, and is a simple book catalog based on the common `BookCatalogue` namespace. The contents of `BookCatalogue.xml` are listed below:

```
<?xml version="1.0" encoding="UTF-8"?>
<BookCatalogue xmlns="http://www.publishing.org">
  <Book>
    <Title>Yogasana Vijnana: the Science of Yoga</Title>
    <author>Dhirendra Brahmachari</Author>
    <Date>1966</Date>
    <ISBN>81-40-34319-4</ISBN>
    <Publisher>Dhirendra Yoga Publications</Publisher>
    <Cost currency="INR">11.50</Cost>
  </Book>
  <Book>
    <Title>The First and Last Freedom</Title>
    <Author>J. Krishnamurti</Author>
    <Date>1954</Date>
    <ISBN>0-06-064831-7</ISBN>
    <Publisher>Harper &amp; Row</Publisher>
    <Cost currency="USD">2.95</Cost>
  </Book>
</BookCatalogue>
```

cursor Sample – CursorParse.java

Located in the <javaee.tutorial.home>/examples/stax/cursor directory, CursorParse.java demonstrates using the StAX cursor API to read an XML document. In this sample, the application instructs the parser to read the next event in the XML input stream by calling <code>next()</code>.

Note that <code>next()</code> just returns an integer constant corresponding to underlying event where the parser is positioned. The application needs to call the relevant function to get more information related to the underlying event.

You can imagine this approach as a virtual cursor moving across the XML input stream. There are various accessor methods which can be called when that virtual cursor is at particular event.

Stepping Through Events

In this example, the client application pulls the next event in the XML stream by calling the next() method on the parser; for example:

```
try
  {
     for(int i =0 ; i< count ; i++)
       {
          //pass the file name.. allrelativeentity
          //references will be resolved againstthis as
          //base URI.
          XMLStreamReader xmlr=
xmlif.createXMLStreamReader(filename, new
FileInputStream(filename));
          //when XMLStreamReader is created, it is positioned
at START_DOCUMENT event.
          int eventType = xmlr.getEventType();
          //printEventType(eventType);
          printStartDocument(xmlr);
          //check if there aremore eventsinthe input stream
          while(xmlr.hasNext())
            {
               eventType =xmlr.next();
               //printEventType(eventType);
               //these functionsprints the information about
theparticular event by calling relevant function
               printStartElement(xmlr);
               printEndElement(xmlr);
               printText(xmlr);
```

```
                    printPIData(xmlr);
                    printComment(xmlr);
                }
            }
```

Note that `next()` just returns an integer constant corresponding to the event underlying the current cursor location. The application calls the relevant function to get more information related to the underlying event. There are various accessor methods which can be called when the cursor is at particular event.

Returning String Representations

Because the `next()` method only returns integers corresponding to underlying event types, you typically need to map these integers to string representations of the events; for example:

```
public final staticString getEventTypeString(inteventType)
{
  switch(eventType)
    {
        case XMLEvent.START_ELEMENT:
          return "START_ELEMENT";
        case XMLEvent.END_ELEMENT:
          return "END_ELEMENT";
        case XMLEvent.PROCESSING_INSTRUCTION:
          return "PROCESSING_INSTRUCTION";
        case XMLEvent.CHARACTERS:
          return "CHARACTERS";
        case XMLEvent.COMMENT:
          return "COMMENT";
        case XMLEvent.START_DOCUMENT:
          return "START_DOCUMENT";
        case XMLEvent.END_DOCUMENT:
          return "END_DOCUMENT";
        case XMLEvent.ENTITY_REFERENCE:
          return "ENTITY_REFERENCE";
        case XMLEvent.ATTRIBUTE:
          return "ATTRIBUTE";
        case XMLEvent.DTD:
          return "DTD";
        case XMLEvent.CDATA:
          return "CDATA";
        case XMLEvent.SPACE:
          return "SPACE";
    }
  return"UNKNOWN_EVENT_TYPE , "+ eventType;
}
```

Running the Sample

When you run the CursorParse sample, the class is compiled, and the XML stream is parsed and returned to STDOUT.

cursor2event Sample – CursorApproachEventObject.java

Located in the <javaee.tutorial.home>/examples/stax/cursor2event directory, CursorApproachEventObject.java demonstrates how to get information returned by an XMLEvent object even when using the cursor API.

The idea here is that the cursor API's XMLStreamReader returns integer constants corresponding to particular events, where as the event iterator API's XMLEventReader returns immutable and persistent event objects. XMLStreamReader is more efficient, but XMLEventReader is easier to use, as all the information related to a particular event is encapsulated in a returned XMLEvent object. However, the disadvantage of event approach is the extra overhead of creating objects for every event, which consumes both time and memory.

With this mind, XMLEventAllocator can be used to get event information as an XMLEvent object, even when using the cursor API.

Instantiating an XMLEventAllocator

The first step is to create a new XMLInputFactory and instantiate an XMLEventAllocator:

```
XMLInputFactory xmlif = XMLInputFactory.newInstance();
System.out.println("FACTORY: " + xmlif);
xmlif.setEventAllocator(new XMLEventAllocatorImpl());
allocator = xmlif.getEventAllocator();
XMLStreamReader xmlr = xmlif.createXMLStreamReader(filename,
new FileInputStream(filename));
```

Creating an Event Iterator

The next step is to create an event iterator:

```
int eventType = xmlr.getEventType();
while(xmlr.hasNext()){
  eventType = xmlr.next();
  //Get all "Book" elements as XMLEvent object
  if(eventType == XMLStreamConstants.START_ELEMENT &&
xmlr.getLocalName().equals("Book")){
     //get immutable XMLEvent
     StartElement event = getXMLEvent(xmlr).asStartElement();
     System.out.println("EVENT: " + event.toString());
  }
}
```

Creating the Allocator Method

The final step is to create the `XMLEventAllocator` method:

```
private static XMLEvent getXMLEvent(XMLStreamReader reader)
throws XMLStreamException{
  return allocator.allocate(reader);
}
```

Running the Sample

When you run the `CursorApproachEventObject` sample, the class is compiled, and the XML stream is parsed and returned to `STDOUT`. Note how the `Book` events are returned as strings.

event Sample – EventParse.java

Located in the `<javaee.tutorial.home>/examples/stax/event` directory, `EventParse.java` demonstrates how to use the StAX event API to read an XML document.

Creating an Input Factory

The first step is to create a new instance of XMLInputFactory:

```
XMLInputFactory factory = XMLInputFactory.newInstance();
System.out.println("FACTORY: " + factory);
```

Creating an Event Reader

The next step is to create an instance of XMLEventReader:

```
XMLEventReader r = factory.createXMLEventReader(filename, new
FileInputStream(filename));
```

Creating an Event Iterator

The third step is to create an event iterator:

```
XMLEventReader r = factory.createXMLEventReader(filename, new
FileInputStream(filename));
while(r.hasNext()) {
  XMLEvent e = r.nextEvent();
  System.out.println(e.toString());
}
```

Getting the Event Stream

The final step is to get the underlying event stream:

```
public final static String getEventTypeString(int eventType)
{
  switch (eventType)
    {
      case XMLEvent.START_ELEMENT:
        return "START_ELEMENT";
      case XMLEvent.END_ELEMENT:
        return "END_ELEMENT";
      case XMLEvent.PROCESSING_INSTRUCTION:
        return "PROCESSING_INSTRUCTION";
      case XMLEvent.CHARACTERS:
        return "CHARACTERS";
      case XMLEvent.COMMENT:
        return "COMMENT";
      case XMLEvent.START_DOCUMENT:
            return "START_DOCUMENT";
         case XMLEvent.END_DOCUMENT:
            return "END_DOCUMENT";
         case XMLEvent.ENTITY_REFERENCE:
            return "ENTITY_REFERENCE";
         case XMLEvent.ATTRIBUTE:
            return "ATTRIBUTE";
         case XMLEvent.DTD:
            return "DTD";
         case XMLEvent.CDATA:
            return "CDATA";
         case XMLEvent.SPACE:
            return "SPACE";
      }
    return "UNKNOWN_EVENT_TYPE " + "," + eventType;
  }
```

Running the Sample

When you run the EventParse sample, the class is compiled, and the XML stream is parsed as events and returned to STDOUT. For example, an instance of the Author element is returned as:

```
<['http://www.publishing.org']::Author>
Dhirendra Brahmachari
</['http://www.publishing.org']::Author>
```

Note in this example that the event comprises an opening and closing tag, both of which include the namespace. The content of the element is returned as a string within the tags.

Similarly, an instance of the Cost element is returned as:

```
<['http://www.publishing.org']::Cost currency='INR'>
11.50
</['http://www.publishing.org']::Cost>
```

In this case, the currency attribute and value are returned in the opening tag for the event.

See earlier in this chapter, in the “Iterator API” and “Reading XML Streams” sections, for a more detailed discussion of StAX event parsing.

filter Sample – MyStreamFilter.java

Located in the <javaee.tutorial.home>/examples/stax/filter directory, MyStreamFilter.java demonstrates how to use the StAX stream filter API to filter out events not needed by your application. In this example, the parser filters out all events except StartElement and EndElement.

Implementing the StreamFilter Class

The MyStreamFilter implements javax.xml.stream.StreamFilter:

```
public class MyStreamFilter implements
javax.xml.stream.StreamFilter{
```

Creating an Input Factory

The next step is to create an instance of XMLInputFactory. In this case, various properties are also set on the factory:

```
XMLInputFactory xmlif = null ;
try{
xmlif = XMLInputFactory.newInstance();
xmlif.setProperty(XMLInputFactory.IS_REPLACING_ENTITY_REFERENC
ES,Boolean.TRUE);
xmlif.setProperty(XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTIT
IES,Boolean.FALSE);
xmlif.setProperty(XMLInputFactory.IS_NAMESPACE_AWARE ,
Boolean.TRUE);
xmlif.setProperty(XMLInputFactory.IS_COALESCING ,
Boolean.TRUE);
}catch(Exception ex){
  ex.printStackTrace();
}
System.out.println("FACTORY: " + xmlif);
System.out.println("filename = "+ filename);
```

Creating the Filter

The next step is to instantiate a file input stream and create the stream filter:

```
FileInputStream fis = new FileInputStream(filename);

XMLStreamReader xmlr =
xmlif.createFilteredReader(xmlif.createXMLStreamReader(fis),
new MyStreamFilter());

int eventType = xmlr.getEventType();
printEventType(eventType);
while(xmlr.hasNext()){
  eventType = xmlr.next();
  printEventType(eventType);
  printName(xmlr,eventType);
  printText(xmlr);
  if(xmlr.isStartElement()){
    printAttributes(xmlr);
  }
  printPIData(xmlr);
  System.out.println("-----------------------------");
}
```

Capturing the Event Stream

The next step is to capture the event stream. This is done in basically the same way as in the `event Sample - EventParse.java` sample.

Filtering the Stream

The final step is the filter the stream:

```
public boolean accept(XMLStreamReader reader) {
  if(!reader.isStartElement() && !reader.isEndElement())
    return false;
  else
    return true;
}
```

Running the Sample

When you run the `MyStreamFilter` sample, the class is compiled, and the XML stream is parsed as events and returned to `STDOUT`. For example an `Author` event is returned as follows:

```
EVENT TYPE(1):START_ELEMENT
HAS NAME: Author
HAS NO TEXT
HAS NO ATTRIBUTES
-----------------------------
EVENT TYPE(2):END_ELEMENT
HAS NAME: Author
HAS NO TEXT
-----------------------------
```

Similarly, a `Cost` event is returned as follows:

```
EVENT TYPE(1):START_ELEMENT
HAS NAME: Cost
HAS NO TEXT

HAS ATTRIBUTES:
ATTRIBUTE-PREFIX:
ATTRIBUTE-NAMESP: null
ATTRIBUTE-NAME:   currency
ATTRIBUTE-VALUE: USD
ATTRIBUTE-TYPE:  CDATA

-----------------------------
EVENT TYPE(2):END_ELEMENT
HAS NAME: Cost
HAS NO TEXT
-----------------------------
```

See earlier in this chapter, in the “Iterator API” and “Reading XML Streams” sections, for a more detailed discussion of StAX event parsing.

readnwrite Sample – EventProducerConsumer.java

Located in the `<javaee.tutorial.home>/examples/stax/`readnwrite directory, `EventProducerConsumer.java` demonstrates how to use a StAX parser simultaneously as both a producer and a consumer.

The StAX `XMLEventWriter` API extends from the `XMLEventConsumer` interface, and is referred to as an *event consumer*. By contrast, `XMLEventReader` is an *event producer*. StAX supports simultaneous reading and writing, such that it is possible to read from one XML stream sequentially and simultaneously write to another stream.

This sample shows how the StAX producer/consumer mechanism can be used to read and write simultaneously. This sample also shows how a stream can be modified, and new events can be added dynamically and then written to different stream.

Creating an Event Producer/Consumer

The first step is to instantiate an event factory and then create an instance of an event producer/consumer:

```
XMLEventFactory m_eventFactory=XMLEventFactory.newInstance();
public EventProducerConsumer() {
}
.
.
.
try{
  EventProducerConsumer ms = new EventProducerConsumer();

  XMLEventReader reader =
XMLInputFactory.newInstance().createXMLEventReader(new
java.io.FileInputStream(args[0]));
  XMLEventWriter writer =
XMLOutputFactory.newInstance().createXMLEventWriter(System.out
);
```

Creating an Iterator

The next step is to create an iterator to parse the stream:

```
while(reader.hasNext())
  {
    XMLEvent event = (XMLEvent)reader.next();
    if(event.getEventType() == event.CHARACTERS)
      {

writer.add(ms.getNewCharactersEvent(event.asCharacters()));
      }
```

```
        else
          {
            writer.add(event);
          }
      }
    writer.flush();
```

Creating a Writer

The final step is to create a stream writer in the form of a new `Character` event:

```
    Characters getNewCharactersEvent(Characters event){
      if(event.getData().equalsIgnoreCase("Name1")){
        return
    m_eventFactory.createCharacters(Calendar.getInstance().getTime
    ().toString());

      }
      //else return the same event
      else return event;
    }
```

Running the Sample

When you run the `EventProducerConsumer` sample, the class is compiled, and the XML stream is parsed as events and written back to `STDOUT`:

```
    <?xml version="1.0" encoding="UTF-8"?>
    <BookCatalogue xmlns="http://www.publishing.org">
      <Book>
        <Title>Yogasana Vijnana: the Science of Yoga</Title>
        <Author>Dhirendra Brahmachari</Author>
        <Date>1966</Date>
        <ISBN>81-40-34319-4</ISBN>
        <Publisher>Dhirendra Yoga Publications</Publisher>
        <Cost currency="INR">11.50</Cost>
      </Book>

      <Book>
        <Title>The First and Last Freedom</Title>
        <Author>J. Krishnamurti</Author>
        <Date>1954</Date>
        <ISBN>0-06-064831-7</ISBN>
        <Publisher>Harper &amp; Row</Publisher>
        <Cost currency="USD">2.95</Cost>
      </Book>
    </BookCatalogue>
```

writer Sample – CursorWriter.java

Located in the `<javaee.tutorial.home>/examples/stax/writer` directory, `CursorWriter.java` demonstrates how to use the StAX cursor API to write an XML stream.

Creating the Output Factory

The first step is to create an instance of `XMLOutputFactory`:

```
XMLOutputFactory xof =  XMLOutputFactory.newInstance();
```

Creating a Stream Writer

The next step is to create an instance of `XMLStreamWriter`:

```
XMLStreamWriter xtw = null;
```

Writing the Stream

The final step is to write the XML stream. Note that the stream is flushed and closed after the final `EndDocument` is written:

```
xtw = xof.createXMLStreamWriter(new FileWriter(fileName));
xtw.writeComment("all elements here are explicitly in the HTML
namespace");
xtw.writeStartDocument("utf-8","1.0");
xtw.setPrefix("html", "http://www.w3.org/TR/REC-html40");
xtw.writeStartElement("http://www.w3.org/TR/REC-
html40","html");
xtw.writeNamespace("html", "http://www.w3.org/TR/REC-html40");
xtw.writeStartElement("http://www.w3.org/TR/REC-
html40","head");
xtw.writeStartElement("http://www.w3.org/TR/REC-
html40","title");
xtw.writeCharacters("Frobnostication");
xtw.writeEndElement();
xtw.writeEndElement();
xtw.writeStartElement("http://www.w3.org/TR/REC-
html40","body");
xtw.writeStartElement("http://www.w3.org/TR/REC-html40","p");
xtw.writeCharacters("Moved to");
xtw.writeStartElement("http://www.w3.org/TR/REC-html40","a");
xtw.writeAttribute("href","http://frob.com");
xtw.writeCharacters("here");
xtw.writeEndElement();
xtw.writeEndElement();
xtw.writeEndElement();
xtw.writeEndElement();
xtw.writeEndDocument();
xtw.flush();
xtw.close();
```

Running the Sample

When you run the `CursorWriter` sample, the class is compiled, and the XML stream is parsed as events and written to a file named `CursorWriter-Output`:

```
<!--all elements here are explicitly in the HTML namespace-->
<?xml version="1.0" encoding="utf-8"?>
<html:html xmlns:html="http://www.w3.org/TR/REC-html40">
<html:head>
<html:title>Frobnostication</html:title></html:head>
<html:body>
<html:p>Moved to <html:a href="http://frob.com">here</html:a>
</html:p>
</html:body>
</html:html>
```

Note that in the actual `CursorWriter-Output` file, this stream is written without any linebreaks; the breaks have been added here to make the listing easier to read. In this example, as with the object stream in the `event Sample – Event-Parse.java` sample, the namespace prefix is added to both the opening and closing HTML tags. This is not required by the StAX specification, but it is good practice when the final scope of the output stream is not definitively known.

Further Information

For more information about StAX, see:

- Java Community Process page:
  `http://jcp.org/en/jsr/detail?id=173`.
- W3C Recommendation “Extensible Markup Language (XML) 1.0”:
  `http://www.w3.org/TR/REC-xml`
- XML Information Set:
  `http://www.w3.org/TR/xml-infoset/`
- JAXB specification:
  `http://java.sun.com/xml/jaxb`
- JAX-RPC specification:
  `http//java.sun.com/xml/jaxrpc`
- W3C Recommendation “Document Object Model”:
  `http://www.w3.org/DOM/`
- SAX “Simple API for XML”:
  `http://www.saxproject.org/`
- DOM “Document Object Model”:
  `http://www.w3.org/TR/2002/WD-DOM-Level-3-Core-20020409/core.html#ID-B63ED1A3`
- W3C Recommendation “Namespaces in XML”:
  `http://www.w3.org/TR/REC-xml-names/`

For some useful articles about working with StAX, see:

- Jeff Ryan, “Does StAX Belong in Your XML Toolbox?”:
  `http://www.developer.com/xml/article.php/3397691`
- Elliotte Rusty Harold, “An Introduction to StAX”:
  `http://www.xml.com/pub/a/2003/09/17/stax.html`
- “More efficient XML parsing with the Streaming API for XML”:
  `http://www-106.ibm.com/developerworks/xml/library/x-tipstx/`

5 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-WS handlers and JAXR implementations. Secondarily, it is an API that developers can use when they choose to write SOAP messaging applications directly rather than use JAX-WS. The SAAJ API allows you to do XML messaging from the Java platform: By simply making method calls using the SAAJ API, you can read and write SOAP-based XML messages, and you can optionally send and receive such messages over the Internet (some implementations may not support sending and receiving). 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 and 1.2 specifications and the SOAP with Attachments specification. The SAAJ 1.3 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 122.)

Note: The `javax.xml.messaging` package, defined in the Java API for XML Messaging (JAXM) 1.1 specification, is not part of the Java EE platform 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, giving 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 commonly used features. The code examples in the last part of the tutorial show you how to build an application.

Overview of SAAJ

This section 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 and 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 or 1.2 specification and to the WS-I Basic Profile 1.1 specification simply by making Java API calls.

The Structure of an XML Document

An XML document has a hierarchical structure made up of 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 5–1 illustrates the structure of a SOAP message with no attachments.

Note: Many SAAJ API interfaces extend DOM interfaces. In a SAAJ message, the SOAPPart class is also a DOM document. See SAAJ and DOM (page 122) for details.

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 can include one or more headers that contain metadata about the message (for example, information about the sending and receiving parties). The SOAPBody object, which always follows the SOAPHeader object if there is one, contains the message content. If there is a SOAPFault object (see Using SOAP Faults, page 145), it must be in the SOAPBody object.

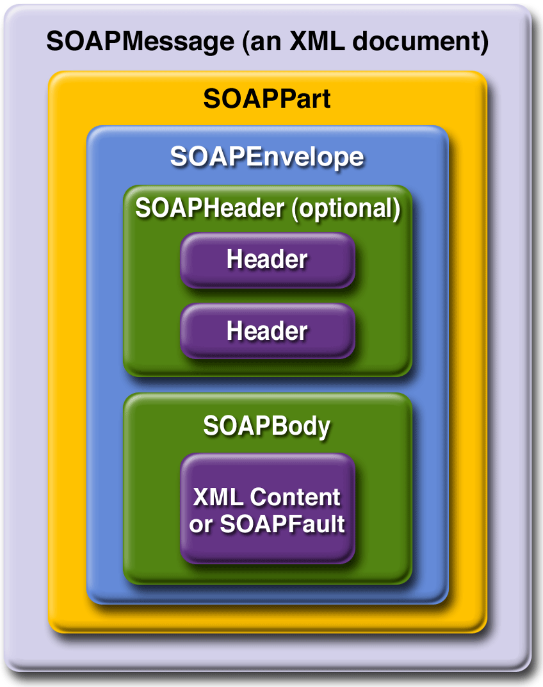

Figure 5–1 SOAPMessage Object with No Attachments

Messages with Attachments

A SOAP message may include one or more attachment parts in addition to the SOAP part. The SOAP part must 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 5–2 shows the high-level structure of a SOAP message that has two attachments.

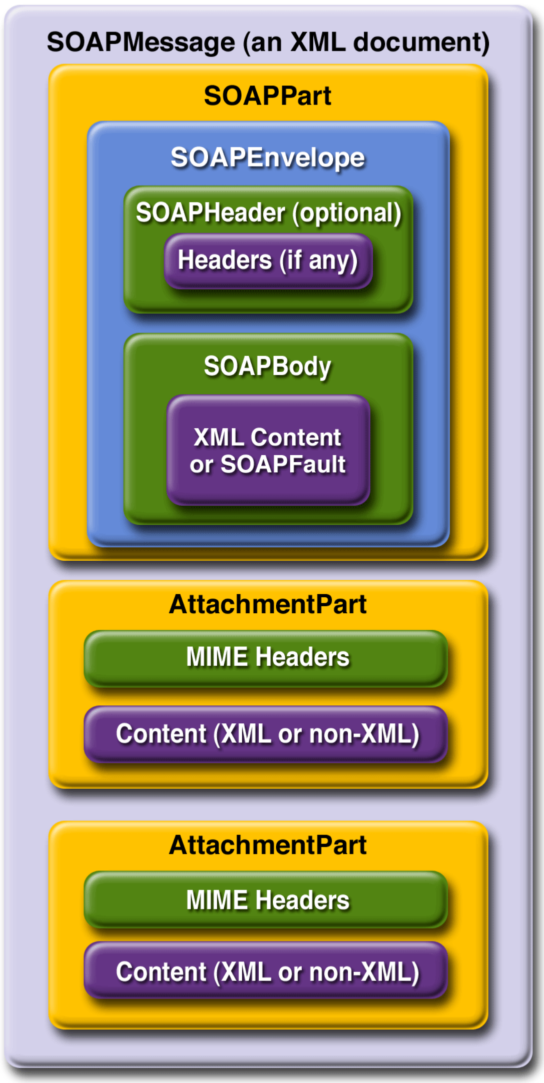

Figure 5–2 `SOAPMessage` Object with Two `AttachmentPart` Objects

The SAAJ API provides the `AttachmentPart` class to represent an 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 must create and add them yourself. The tutorial section walks 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. These headers 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

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. For details on how to use DOM documents with the SAAJ API, see Adding Content to the SOAPPart Object (page 134) and Adding a Document to the SOAP Body (page 136).

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 call method, 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 call method, which both sends the message and blocks until it receives the response. Thus, the return value for the `call` method is the `SOAPMessage` object that is the response to the message that was sent. The `request` parameter 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 `call` method, 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 acknowledgment that the update was received. Such an acknowledgment 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 walks 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 `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 151), you will see the code fragments from earlier parts of the tutorial in runnable applications, which you can test yourself.

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 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`.

If you specify no arguments to the `newInstance` method, it creates a message factory for SOAP 1.1 messages. To create a message factory that allows you to create and process SOAP 1.2 messages, use the following method call:

```
MessageFactory factory =
MessageFactory.newInstance(SOAPConstants.SOAP_1_2_PROTOCOL);
```

To create a message factory that can create either SOAP 1.1 or SOAP 1.2 messages, use the following method call:

```
MessageFactory factory =
MessageFactory.newInstance(SOAPConstants.DYNAMIC_SOAP_PROTOCOL
);
```

This kind of factory enables you to process an incoming message that might be of either type.

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. When you call `createMessage` with no arguments, the message that is created 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 can be deleted if it is not needed. However, if there is one, it must precede the `SOAPBody` object. The `SOAPBody` object can hold either the content of the message or a *fault* message that contains status information or details about a problem with the message. The section Using SOAP Faults (page 145) 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 preceding 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

The `SOAPBody` object contains either content or a fault. To add content to the body, you normally create one or more `SOAPBodyElement` objects to hold the content. You can also add subelements to the `SOAPBodyElement` objects by using the `addChildElement` method. For each element or child element, you add content by using the `addTextNode` method.

When you create any new element, you also need to create an associated `javax.xml.namespace.QName` object so that it is uniquely identified.

Note: You can use `Name` objects instead of `QName` objects. `Name` objects are specific to the SAAJ API, and you create them using either `SOAPEnvelope` methods or `SOAPFactory` methods. However, the `Name` interface may be deprecated at a future release.

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, and this 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 Overview of SOAP Faults (page 146) and Creating and Populating a SOAPFault Object (page 147).

`QName` objects associated with `SOAPBodyElement` or `SOAPHeaderElement` objects must be fully qualified; that is, they must be created with a namespace URI, a local part, and a namespace prefix. Specifying a namespace for an element makes clear which one is meant if more than one element has the same local name.

The following code fragment retrieves the `SOAPBody` object `body` from `message`, constructs a `QName` object for the element to be added, and adds a new `SOAPBodyElement` object to `body`.

```
SOAPBody body = message.getSOAPBody();
QName bodyName = new QName("http://wombat.ztrade.com",
  "GetLastTradePrice", "m");
SOAPBodyElement bodyElement = body.addBodyElement(bodyName);
```

At this point, `body` contains a `SOAPBodyElement` object identified by the `QName` 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 addChildElement method. Then you need to give it the stock symbol using the addTextNode method. The QName 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.

```
QName name = new QName("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 and 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.

Here is the SAAJ code:

```
SOAPMessage message = messageFactory.createMessage();
SOAPHeader header = message.getSOAPHeader();
SOAPBody body = message.getSOAPBody();
```

Here is the 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`. Note that `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.

Here is the SAAJ code:

```
QName bodyName = new QName("http://wombat.ztrade.com",
  "GetLastTradePrice", "m");
SOAPBodyElement bodyElement = body.addBodyElement(bodyName);
```

Here is the 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.

Here is the SAAJ code:

```
QName name = new QName("symbol");
SOAPElement symbol = bodyElement.addChildElement(name);
symbol.addTextNode("SUNW");
```

Here is the 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.

The following example shows how to add multiple `SOAPElement` objects and add 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 `QName` object for it has a namespace URI, a local name, and a namespace prefix. 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, can have `Name` objects with only the local name.

```
SOAPBody body = message.getSOAPBody();
QName bodyName = new QName("http://sonata.fruitsgalore.com",
  "PurchaseLineItems", "PO");
SOAPBodyElement purchaseLineItems =
  body.addBodyElement(bodyName);

QName childName = new QName("Order");
SOAPElement order =
  purchaseLineItems.addChildElement(childName);

childName = new QName("Product");
SOAPElement product = order.addChildElement(childName);
product.addTextNode("Apple");

childName = new QName("Price");
SOAPElement price = order.addChildElement(childName);
price.addTextNode("1.56");

childName = new QName("Order");
SOAPElement order2 =
  purchaseLineItems.addChildElement(childName);

childName = new QName("Product");
SOAPElement product2 = order2.addChildElement(childName);
```

```
product2.addTextNode("Peach");

childName = soapFactory.new QName("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://sonata.fruitsgalore.com">
  <Order>
    <Product>Apple</Product>
    <Price>1.56</Price>
  </Order>

  <Order>
    <Product>Peach</Product>
    <Price>1.48</Price>
  </Order>
</PO:PurchaseLineItems>
```

Getting a SOAPConnection Object

The SAAJ API is focused primarily on reading and writing messages. After you have written a message, you can send it using various mechanisms (such as JMS or JAXM). The SAAJ API does, however, provide a simple mechanism for request-response messaging.

To send a message, a SAAJ client can use a `SOAPConnection` object. A `SOAP-Connection` 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 `SOAP-ConnectionFactory` class with a default implementation. You can get an instance of this implementation using 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 finished 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 to add content or to retrieve 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 `getValue` method is called on `bodyElement`, the element on which the `addTextNode` method was called.

To access bodyElement, you call the getChildElements method on soapBody. Passing bodyName to getChildElements returns a java.util.Iterator object that contains all the child elements identified by the Name object bodyName. You already know that there is only one, so calling the next method on it will return the SOAPBodyElement you want. Note that the Iterator.next method returns a Java Object, so you need 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 more than one element had the name bodyName, you would have to use a while loop using the Iterator.hasNext method 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 create a SOAPHeaderElement object. As with all new elements, it must have an associated QName 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();
QName headerName = new QName(
  "http://ws-i.org/schemas/conformanceClaim/",
  "Claim", "wsi");
SOAPHeaderElement headerElement =
  header.addHeaderElement(headerName);
headerElement.addAttribute(new QName("conformsTo"),
  "http://ws-i.org/profiles/basic/1.1/");
```

At this point, header contains the SOAPHeaderElement object headerElement identified by the QName 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
    xmlns:wsi="http://ws-i.org/schemas/conformanceClaim/"
    conformsTo="http://ws-i.org/profiles/basic/1.1/"/>
</SOAP-ENV:Header>
```

For more information about creating SOAP messages that conform to WS-I, see the Conformance Claim Attachment Mechanisms document described in the Conformance 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 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 preceding 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 an XML document in text form. `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 `SOAPPart.setContent` method 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:

```
<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 136) and Adding Attachments (page 137).

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. After 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 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 after you start manipulating it using DOM, you should no longer use SAAJ APIs.

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 must 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 by using the AttachmentPart method setContent. This method takes two parameters: a Java Object for the content, and a String object for the MIME content type that is used to encode the object. Content in the SOAPBody part of a message automatically has a Content-Type header with the value "text/xml" because the content must be in XML. In contrast, the type of content in an AttachmentPart object must be specified because it can be any type.

Each AttachmentPart object has one or more MIME headers associated with it. When you specify a type to the setContent method, that type is used for the header Content-Type. Note that 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 can call the SOAPMessage method getAttachments and pass it a MIMEHeaders object containing the MIME headers you are interested in.

The following code fragment shows one of the ways to use the method setContent. The Java Object in the first parameter can 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 must add it to the SOAPMessage object, something that 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 attachment variable 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 can 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 this code fragment. First, it sets a header for Content-ID using 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, something that is possible because one of the things a DataHandler object does is to 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 need to access the attachment. The SOAPMessage class provides two versions of the getAttachments method 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, getAttachments 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 the AttachmentPart objects in the SOAPMessage object message. Then it prints the content ID, the content type, and the 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.equals("text/plain")) {
     Object content = attachment.getContent();
     System.out.println("Attachment contains:\n" + content);
  }
}
```

Adding Attributes

An XML element can have one or more attributes that give information about that element. An attribute consists of a name for the attribute followed immediately by an equal 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 SOAP-

BodyElement object or SOAPHeaderElement object, it is legal for its QName object to contain only a local name.

```
QName attributeName = new QName("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 preceding line of code would have returned only the value for the attribute named id. If you wanted to retrieve the values for all the attributes for person, you would use the method getAllAttributes, which returns an iterator over all the values. The following lines of code retrieve and print each value on a separate line until there are no more attribute values. Note that the Iterator.next method returns a Java Object, which is cast to a QName object so that it can be assigned to the QName object attributeName. (The examples in DOMExample.java and DOMSrcExample.java (page 162) use code similar to this.)

```
Iterator iterator = person.getAllAttributesAsQNames();
while (iterator.hasNext()){
  QName attributeName = (QName) iterator.next();
  System.out.println("Attribute name is " +
     attributeName.toString());
  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 have seen how to add, retrieve, and remove attributes. This information is general in that it applies to any element. The next section discusses attributes that can 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, whereas 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 SOAP 1.2 specification defines three such attributes: `role` (a new name for `actor`), `mustUnderstand`, and `relay`.

The next sections discuss these attributes.

See HeaderExample.java (page 160) for an example that uses the code shown in this section.

The Actor Attribute

The `actor` attribute 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 can 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";

QName order =
  new QName(nameSpaceURI, "orderDesk", nameSpace);
SOAPHeaderElement orderHeader =
  header.addHeaderElement(order);
orderHeader.setActor("http://gizmos.com/orders");

QName shipping =
  new QName(nameSpaceURI, "shippingDesk", nameSpace);
SOAPHeaderElement shippingHeader =
  header.addHeaderElement(shipping);
shippingHeader.setActor("http://gizmos.com/shipping");

QName confirmation =
  new QName(nameSpaceURI, "confirmationDesk", nameSpace);
SOAPHeaderElement confirmationHeader =
  header.addHeaderElement(confirmation);
confirmationHeader.setActor(
  "http://gizmos.com/confirmations");

QName billing =
  new QName(nameSpaceURI, "billingDesk", nameSpace);
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 the SOAPHeaderElement objects that have an actor that matches the specified actor. The first method, examineHeaderElements, returns an iterator over all the elements that have the specified actor.

```
java.util.Iterator headerElements =
  header.examineHeaderElements("http://gizmos.com/orders");
```

The second method, extractHeaderElements, not only returns an iterator over all the SOAPHeaderElement objects that have the specified actor attribute but also detaches them from the SOAPHeader object. So, for example, after the order desk application did its work, it would call extractHeaderElements to remove all the SOAPHeaderElement objects that applied to it.

```
java.util.Iterator headerElements =
  header.extractHeaderElements("http://gizmos.com/orders");
```

Each SOAPHeaderElement object can have only one actor attribute, but the same actor can 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();
  QName headerName =
    headerElement.getElementQName();
  System.out.println("\nHeader name is " +
    headerName.toString());
  System.out.println("Actor is " +
    headerElement.getActor());
}
```

The role Attribute

The role attribute is the name used by the SOAP 1.2 specification for the SOAP 1.2 actor attribute. The SOAPHeaderElement methods setRole and getRole perform the same functions as the setActor and getActor methods.

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 a 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 Using SOAP Faults, page 145.) 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.

For example, you could set the mustUnderstand attribute to true for the confirmationHeader in the code fragment in The Actor Attribute (page 141):

```
QName confirmation =
  new QName(nameSpaceURI, "confirmationDesk", nameSpace);
SOAPHeaderElement confirmationHeader =
  header.addHeaderElement(confirmation);
confirmationHeader.setActor(
  "http://gizmos.com/confirmations");
confirmationHeader.setMustUnderstand(true);
```

This fragment produces the following XML:

```
<ns:confirmationDesk
  xmlns:ns="http://gizmos.com/NSURI"
  SOAP-ENV:actor="http://gizmos.com/confirmations"
  SOAP-ENV:mustUnderstand="1"/>
```

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 preceding section:

```
System.out.println("mustUnderstand is " +
  headerElement.getMustUnderstand());
```

The relay Attribute

The SOAP 1.2 specification adds a third attribute to a `SOAPHeaderElement`, `relay`. This attribute, like `mustUnderstand`, is a boolean value. If it is set to `true`, it indicates that the SOAP header block must not be processed by any node that is targeted by the header block, but must only be passed on to the next targeted node. This attribute is ignored on header blocks whose `mustUnderstand` attribute is set to true or that are targeted at the ultimate receiver (which is the default). The default value of this attribute is `false`.

For example, you could set the `relay` element to `true` for the `billingHeader` in the code fragment in The Actor Attribute (page 141) (also changing `setActor` to `setRole`):

```
QName billing =
  new QName(nameSpaceURI, "billingDesk", nameSpace);
SOAPHeaderElement billingHeader =
  header.addHeaderElement(billing);
billingHeader.setRole("http://gizmos.com/billing");
billingHeader.setRelay(true);
```

This fragment produces the following XML:

```
<ns:billingDesk
  xmlns:ns="http://gizmos.com/NSURI"
  env:relay="true"
  env:role="http://gizmos.com/billing"/>
```

To display the value of the attribute, call `getRelay`:

```
System.out.println("relay is " + headerElement.getRelay());
```

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, which 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. Furthermore, 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.

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, as 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 139), 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: it must contain a prefix followed by a local name. The SOAP specifications define a set of fault code local name values, which a developer can extend to cover other problems. (These are defined in section 4.4.1 of the SOAP 1.1 specification and in section 5.4.6 of the SOAP 1.2 specification.) Table 5–1 on page 149 lists and describes the default fault code local names defined in the specifications.

A SOAP 1.2 fault code can optionally have a hierarchy of one or more subcodes.

- 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 Actor Attribute, page 141.
- 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 seen how to add content to a `SOAPBody` object; this section walks 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 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 SOAP 1.1 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.

```
QName faultName =
  new QName(SOAPConstants.URI_NS_SOAP_ENVELOPE, "Server");
fault.setFaultCode(faultName);
fault.setFaultActor("http://gizmos.com/orders");
fault.setFaultString("Server not responding");
```

The SOAP 1.2 code would look like this:

```
QName faultName =
  new QName(SOAPConstants.URI_NS_SOAP_1_2_ENVELOPE,
    "Receiver");
fault.setFaultCode(faultName);
fault.setFaultRole("http://gizmos.com/order");
fault.addFaultReasonText("Server not responding", Locale.US);
```

To add one or more subcodes to the fault code, call the method fault.append-FaultSubcode, which takes a QName argument.

The SOAPFault object fault, created in the preceding 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 to set a fault actor. Also note that fault does not have a Detail object because it does not relate to the SOAPBody object. (If you use SOAP 1.2, you can use the setFault-Role method instead of setFaultActor.)

The following SOAP 1.1 code fragment creates a SOAPFault object that includes a Detail object. Note that a SOAPFault object can have only one Detail object, which is simply a container for DetailEntry objects, but the Detail object can have multiple DetailEntry objects. The Detail object in the following lines of code has two DetailEntry objects added to it.

```
SOAPFault fault = body.addFault();

QName faultName =
  new QName(SOAPConstants.URI_NS_SOAP_ENVELOPE, "Client");
fault.setFaultCode(faultName);
fault.setFaultString("Message does not have necessary info");

Detail detail = fault.addDetail();

QName entryName =
  new QName("http://gizmos.com/orders/", "order", "PO");
```

```
DetailEntry entry = detail.addDetailEntry(entryName);
entry.addTextNode("Quantity element does not have a value");

QName entryName2 =
  new QName("http://gizmos.com/orders/", "order", "PO");
DetailEntry entry2 = detail.addDetailEntry(entryName2);
entry2.addTextNode("Incomplete address: no zip code");
```

See SOAPFaultTest.java (page 168) for an example that uses code like that shown in this section.

The SOAP 1.1 and 1.2 specifications define slightly different values for a fault code. Table 5–1 lists and describes these values.

Table 5–1 SOAP Fault Code Values

SOAP 1.1	SOAP 1.2	Description
VersionMismatch	VersionMismatch	The namespace or local name for a `SOAPEnvelope` object was invalid.
MustUnderstand	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	Sender	The `SOAPMessage` object was not formed correctly or did not contain the information needed to succeed.
Server	Receiver	The `SOAPMessage` object could not be processed because of a processing error, not because of a problem with the message itself.
N/A	DataEncodingUnknown	A SOAP header block or SOAP body child element information item targeted at the faulting SOAP node is scoped with a data encoding that the faulting node does not support.

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 whether the SOAPBody object contains a SOAPFault object. If it does, 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();
  QName code = newFault.getFaultCodeAsQName();
  String string = newFault.getFaultString();
  String actor = newFault.getFaultActor();
```

To retrieve subcodes from a SOAP 1.2 fault, call the method newFault.getFaultSubcodes.

Next the code prints the values it has just retrieved. Not all messages are required to have a fault actor, so the code tests to see whether 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.toString());
  System.out.println("  Local name = " + code.getLocalPart());
  System.out.println("  Namespace prefix = " +
    code.getPrefix() + ", bound to " +
    code.getNamespaceURI());
  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 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 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 168) for an example that uses code like that shown in this section.

Code Examples

The first part of this tutorial uses 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, DOMSrcExample.java, Attachments.java, and SOAPFaultTest.java.

Note: Before you run any of the examples, follow the preliminary setup instructions in Building the Examples (page xxxiii).

Request.java

The class `Request.java` puts together the code fragments used in the section Tutorial (page 123) 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 javax.xml.namespace.QName;
import java.util.Iterator;
import java.net.URL;

public class Request {
  public static void main(String[] args){
    try {
      SOAPConnectionFactory soapConnectionFactory =
        SOAPConnectionFactory.newInstance();
      SOAPConnection connection =
        soapConnectionFactory.createConnection();

      MessageFactory factory =
        MessageFactory.newInstance();
      SOAPMessage message = factory.createMessage();

      SOAPHeader header = message.getSOAPHeader();
      SOAPBody body = message.getSOAPBody();
      header.detachNode();

      QName bodyName = new QName("http://wombat.ztrade.com",
        "GetLastTradePrice", "m");
      SOAPBodyElement bodyElement =
        body.addBodyElement(bodyName);

      QName name = new QName("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);
            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();
         }
      }
   }
```

For `Request.java` to be runnable, the second argument supplied to the `call` method would have to be a valid existing URI, and this 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 from which you can get information about businesses that have registered themselves with the registry service. For this example, the MyUddiPing application is accessing a test (demo) version of a UDDI service registry. 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>/javaeetutorial5/examples/saaj/myuddiping/
```

Note: *<INSTALL>* is the directory where you installed the tutorial bundle.

In the `myuddiping` directory, you will find three files and the `src` directory. The `src` directory contains one source file, `MyUddiPing.java`.

The file `build.xml` is the `asant` build file for this example.

The file `build.properties` defines one property.

The file `uddi.properties` contains the URL of the destination (a UDDI test registry). To install this registry, follow the instructions in Preliminaries: Getting Access to a Registry (page 175). If the Application Server where you install the registry is running on a remote system, open `uddi.properties` in a text editor and replace `localhost` with the name of the remote system.

The `prepare` target creates a directory named `build`. To invoke the `prepare` target, you type the following at the command line:

```
asant 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:

```
asant 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 lines of code import the interfaces used in the application.

```
import javax.xml.soap.SOAPConnectionFactory;
import javax.xml.soap.SOAPConnection;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.SOAPHeader;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPBodyElement;
import javax.xml.soap.SOAPElement;
import javax.xml.namespace.QName;
import java.net.URL;
import java.util.Properties;
import java.util.Enumeration;
import java.util.Iterator;
import java.io.FileInputStream;
```

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 to check to see whether two arguments were supplied. If they were 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("Usage: asant run " +
          "-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, which is in the myuddiping directory.

```
      Properties myprops = new Properties();
      myprops.load(new FileInputStream(args[0]));

      Properties props = System.getProperties();

      Enumeration propNames = myprops.propertyNames();
      while (propNames.hasMoreElements()) {
        String s = (String) propNames.nextElement();
        props.setProperty(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 a SOAP 1.1 MessageFactory, using the MessageFactory instance 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();
header.detachNode();
SOAPBody body = message.getSOAPBody();
```

The following lines of code create the UDDI find_business message. The first line creates a 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 that has the QName object name and adds text to the element by using the method addTextNode. The added text is the business name you will supply at the command line when you run the application.

```
SOAPBodyElement findBusiness =
   body.addBodyElement(new QName(
      "urn:uddi-org:api_v2", "find_business"));
findBusiness.addAttribute(new QName("generic"), "2.0");
findBusiness.addAttribute(new QName("maxRows"), "100");

SOAPElement businessName =
   findBusiness.addChildElement(new QName("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 properties.

```
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 SOAP-Message 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 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 only 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 step is to 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(new QName(
      "urn:uddi-org:api_v2",
      "businessList"));
```

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 must 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 in the same way it retrieved the businessList. Children of SOAPBodyElement objects and all child elements from this point forward are SOAPElement objects.

```
Iterator businessInfosIterator =
   businessList.getChildElements(new QName(
      "urn:uddi-org:api_v2", "businessInfos"));

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 that have 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();

               Iterator nameIterator =
                  businessInfo.getChildElements(new QName(
                     "urn:uddi-org:api_v2", "name"));

               while (nameIterator.hasNext()) {
                  businessName =
                     (SOAPElement)nameIterator.next();
                  System.out.println("Company name: " +
                     businessName.getValue());
               }
               Iterator descriptionIterator =
                  businessInfo.getChildElements(new QName(
                     "urn:uddi-org:api_v2", "description"));

               while (descriptionIterator.hasNext()) {
                  SOAPElement businessDescription =
                     (SOAPElement) descriptionIterator.next();
                  System.out.println("Description: " +
                     businessDescription.getValue());
               }
               System.out.println("");
            }
         }
      }
```

Finally, the program closes the connection:

```
      connection.close();
```

Running MyUddiPing

You are now 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 that is set in `build.properties`. The other argument is the first letters of 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 a command like the following to run the example:

```
   asant run -Dbusiness-name=the
```

The program output depends on the contents of the registry. For example:

```
Content extracted from the reply message:

Company name: The Coffee Break
Description: Purveyor of the finest coffees. Established 1950
```

The program will not return any results until you have run the examples in Chapter 6

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:

```
asant clean
```

HeaderExample.java

The example `HeaderExample.java`, based on the code fragments in the section Adding Attributes (page 139), creates a message that has several headers. It then retrieves the contents of the headers and prints them. The example generates either a SOAP 1.1 message or a SOAP 1.2 message, depending on arguments you specify. You will find the code for `HeaderExample` in the following directory:

```
<INSTALL>/javaeetutorial5/examples/saaj/headers/src/
```

Running HeaderExample

To run HeaderExample, you use the file `build.xml` that is in the directory `<INSTALL>/javaeetutorial5/examples/saaj/headers/`.

To run HeaderExample, use one of the following commands:

```
asant run -Dsoap=1.1
asant run -Dsoap=1.2
```

This command executes the `prepare`, `build`, and `run` targets in the `build.xml` and `targets.xml` files.

When you run HeaderExample to generate a SOAP 1.1 message, 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 xmlns:ns="http://gizmos.com/NSURI"
SOAP-ENV:actor="http://gizmos.com/orders"/>
<ns:shippingDesk xmlns:ns="http://gizmos.com/NSURI"
SOAP-ENV:actor="http://gizmos.com/shipping"/>
<ns:confirmationDesk xmlns:ns="http://gizmos.com/NSURI"
SOAP-ENV:actor="http://gizmos.com/confirmations"
SOAP-ENV:mustUnderstand="1"/>
<ns:billingDesk xmlns:ns="http://gizmos.com/NSURI"
SOAP-ENV:actor="http://gizmos.com/billing"/>
</SOAP-ENV:Header><SOAP-ENV:Body/></SOAP-ENV:Envelope>

Header name is {http://gizmos.com/NSURI}orderDesk
Actor is http://gizmos.com/orders
mustUnderstand is false

Header name is {http://gizmos.com/NSURI}shippingDesk
Actor is http://gizmos.com/shipping
mustUnderstand is false

Header name is {http://gizmos.com/NSURI}confirmationDesk
Actor is http://gizmos.com/confirmations
mustUnderstand is true

Header name is {http://gizmos.com/NSURI}billingDesk
Actor is http://gizmos.com/billing
mustUnderstand is false
```

When you run HeaderExample to generate a SOAP 1.2 message, you will see output similar to the following:

```
----- Request Message ----

<env:Envelope
xmlns:env="http://www.w3.org/2003/05/soap-envelope">
<env:Header>
<ns:orderDesk xmlns:ns="http://gizmos.com/NSURI"
env:role="http://gizmos.com/orders"/>
<ns:shippingDesk xmlns:ns="http://gizmos.com/NSURI"
env:role="http://gizmos.com/shipping"/>
```

```
<ns:confirmationDesk xmlns:ns="http://gizmos.com/NSURI"
env:mustUnderstand="true"
env:role="http://gizmos.com/confirmations"/>
<ns:billingDesk xmlns:ns="http://gizmos.com/NSURI"
env:relay="true" env:role="http://gizmos.com/billing"/>
</env:Header><env:Body/></env:Envelope>

Header name is {http://gizmos.com/NSURI}orderDesk
Role is http://gizmos.com/orders
mustUnderstand is false
relay is false

Header name is {http://gizmos.com/NSURI}shippingDesk
Role is http://gizmos.com/shipping
mustUnderstand is false
relay is false

Header name is {http://gizmos.com/NSURI}confirmationDesk
Role is http://gizmos.com/confirmations
mustUnderstand is true
relay is false

Header name is {http://gizmos.com/NSURI}billingDesk
Role is http://gizmos.com/billing
mustUnderstand is false
relay 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 traverse its contents. They show two 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>/javaeetutorial5/examples/saaj/dom/src/
```

Examining DOMExample

DOMExample first creates a DOM document by parsing an XML document. 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()) {
     Node node = (Node) iterator.next();
     SOAPElement element = null;
     Text text = null;
     if (node instanceof SOAPElement) {
        element = (SOAPElement)node;
        QName name = element.getElementQName();
        System.out.println(indent + "Name is " +
           name.toString());
        Iterator attrs = element.getAllAttributesAsQNames();
        while (attrs.hasNext()){
           QName attrName = (QName)attrs.next();
           System.out.println(indent + " Attribute name is " +
              attrName.toString());
           System.out.println(indent + " Attribute value is " +
              element.getAttributeValue(attrName));
        }
        Iterator iter2 = element.getChildElements();
        getContents(iter2, indent + " ");
```

```
            } else {
               text = (Text) node;
               String content = text.getValue();
               System.out.println(indent +
                  "Content is: " + content);
               }
            }
         }
```

Examining DOMSrcExample

DOMSrcExample differs from DOMExample in only a few ways. First, after it parses the document, DOMSrcExample 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 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, DOMSrcExample 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, and 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>/javaeetutorial5/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`, another file that consists only of a body but that contains spaces

You can use any of these four files when you run DOMExample. To run DOMExample, use a command like the following:

```
asant run-dom -Dxml-file=uddimsg.xml
```

When you run DOMExample using the file `uddimsg.xml`, you will see output that begins like the following:

```
Running DOMExample.
Name is {urn:uddi-org:api_v2}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
...
```

You can use either `domsrc1.xml` or `domsrc2.xml` to run DOMSrcExample. To run DOMSrcExample, use a command like the following:

```
asant run-domsrc -Dxml-file=domsrc2.xml
```

When you run DOMSrcExample using the file `domsrc2.xml`, you will see output that begins like the following:

```
run-domsrc:
  Running DOMSrcExample.
  Body contents:
  Content is:

  Name is {urn:uddi-org:api_v2}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 137) and Accessing an AttachmentPart Object (page 139), creates a message that has a text attachment and an image attachment. It then retrieves the contents of the attachments and prints the contents of the text attachment. You will find the code for Attachments in the following directory:

```
<INSTALL>/javaeetutorial5/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.

Running Attachments

To run Attachments, you use the file `build.xml` that is in the directory `<`*`INSTALL`*`>/javaeetutorial5/examples/saaj/attachments/`.

To run Attachments, use the following command:

```
asant 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:

```
asant 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 147) and Retrieving Fault Information (page 149), creates a message that has a `SOAPFault` object. It then retrieves the contents of the `SOAPFault` object and prints them. You will find the code for SOAPFaultTest in the following directory:

```
<INSTALL>/javaeetutorial5/examples/saaj/fault/src/
```

Running SOAPFaultTest

To run SOAPFaultTest, you use the file `build.xml` that is in the directory `<INSTALL>/javaeetutorial5/examples/saaj/fault/`.

Like HeaderExample, this example contains code that allows you to generate either a SOAP 1.1 or a SOAP 1.2 message.

To run SOAPFaultTest, use one of the following commands:

```
asant run -Dsoap=1.1
asant run -Dsoap=1.2
```

When you run SOAPFaultTest to generate a SOAP 1.1 message, 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 = {http://schemas.xmlsoap.org/soap/envelope/}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
```

When you run SOAPFaultTest to generate a SOAP 1.2 message, the output looks like this:

```
Here is what the XML message looks like:

<env:Envelope
xmlns:env="http://www.w3.org/2003/05/soap-envelope">
<env:Header/><env:Body>
<env:Fault>
<env:Code><env:Value>env:Sender</env:Value></env:Code>
<env:Reason><env:Text xml:lang="en-US">
Message does not have necessary info
</env:Text></env:Reason>
<env:Role>http://gizmos.com/order</env:Role>
<env:Detail>
<PO:order xmlns:PO="http://gizmos.com/orders/">
Quantity element does not have a value</PO:order>
<PO:confirmation xmlns:PO="http://gizmos.com/confirm">
Incomplete address: no zip code</PO:confirmation>
</env:Detail></env:Fault>
</env:Body></env:Envelope>

SOAP fault contains:
 Fault code = {http://www.w3.org/2003/05/soap-envelope}Sender
 Local name = Sender
 Namespace prefix = env, bound to
```

```
http://www.w3.org/2003/05/soap-envelope
 Fault reason text = Message does not have necessary info
 Fault role = 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 1.3 specification, available from
  `http://java.sun.com/xml/downloads/saaj.html`
- SAAJ web site:
  `http://java.sun.com/xml/saaj/`
- Simple Object Access Protocol (SOAP) 1.1:
  `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/`
- SOAP Version 1.2 Part 0: Primer:
  `http://www.w3.org/TR/soap12-part0/`
- SOAP Version 1.2 Part 1: Messaging Framework:
  `http://www.w3.org/TR/soap12-part1/`
- SOAP Version 1.2 Part 2: Adjuncts:
  `http://www.w3.org/TR/soap12-part2/`
- WS-I Basic Profile:
  `http://www.ws-i.org/Profiles/BasicProfile-1.1.html`
- WS-I Attachments Profile:
  `http://www.ws-i.org/Profiles/AttachmentsProfile.html`
- SOAP Message Transmission Optimization Mechanism (MTOM):
  `http://www.w3.org/TR/soap12-mtom/`
- XML-binary Optimized Packaging (XOP):
  `http://www.w3.org/TR/xop10/`
- JAXM web site:
  `http://java.sun.com/xml/jaxm/`

6

Java API for XML Registries

THE Java API for XML Registries (JAXR) provides a uniform and standard Java API for accessing various kinds of XML registries.

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 various 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 Application Server, the JAXR provider 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, the JAXR provider supports access only to UDDI version 2 registries.

Currently no public UDDI registries exist. However, you can use the Java WSDP Registry Server, a private UDDI version 2 registry that comes with release 1.5 of the Java Web Services Developer Pack (Java WSDP).

Service Registry, an ebXML registry and repositry with a JAXR provider, is available as part of the Sun Java Enterprise System.

JAXR Architecture

The high-level architecture of JAXR consists of the following parts:

- *A JAXR client*: This is a client program that uses the JAXR API to access a business registry via a JAXR provider.
- *A JAXR provider*: This is 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 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 6–1 illustrates the architecture of JAXR. In the Application Server, 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 Application Server supplies a JAXR provider for UDDI registries.

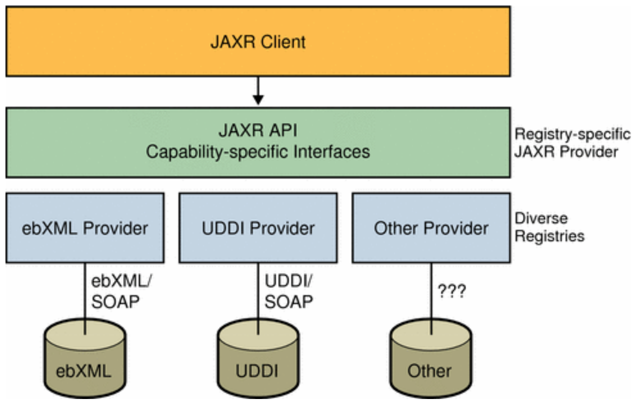

Figure 6–1 JAXR Architecture

Implementing a JAXR Client

This section describes the basic steps to follow in order to implement a JAXR client that can perform queries and updates to a UDDI registry. A JAXR client is a client program that can access registries using the JAXR API. This section 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 Application Server itself is an example of a JAXR provider.

The Application Server provides JAXR in the form of a resource adapter using the Java EE Connector architecture. The resource adapter is in the directory `<JAVAEE_HOME>/lib/install/applications/jaxr-ra`. (`<JAVAEE_HOME>` is the directory where the Application Server is installed.)

This tutorial includes several client examples, which are described in Running the Client Examples (page 199), and a Java EE application example, described in Using JAXR Clients in Java EE Applications (page 206). The examples are in the directory `<INSTALL>/javaeetutorial5/examples/jaxr/`. (`<INSTALL>` is the directory where you installed the tutorial bundle.) Each example directory has a `build.xml` file (which refers to a `targets.xml` file) and a `build.properties` file in the directory `<INSTALL>/javaeetutorial5/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

To use the Java WSDP Registry Server, a private UDDI version 2 registry, you need to download and install Java WSDP 1.5 and then to install the Registry Server in the Application Server.

To download Java WSDP 1.5, perform these steps:

1. Go to the following URL:
   `http://java.sun.com/webservices/downloads/1.5/index.html`
2. Under Java Web Services Developer Pack v1.5, click Download.

3. On the Login page, click the Download link. (You do not have to log in.)
4. Select the Accept radio button to accept the license agreement.
5. Click the download arrow for your platform (Solaris or Windows).
6. Choose the directory where you will download Java WSDP.

Install Java WSDP as follows:

1. Go to the directory where you downloaded Java WSDP 1.5.
2. Run the Java WSDP installer. You can follow the instructions that are linked to from `http://java.sun.com/webservices/downloads/1.5/index.html`, although these instructions refer to a newer version of Java WSDP.
3. On the Select a Web Container page of the installer, select No Web Container.
4. Choose a directory where you will install Java WSDP.
5. Select either a Typical or a Custom installation. If you select Custom, remove the check marks from every checkbox you can except Java WSDP Registry Server. (You cannot remove the check marks from JAXB, JAXP, JAXR, or SAAJ; these technologies are required.)

After the installation completes, install the Registry Server in the Application Server as follows:

1. Stop the Application Server if it is running.
2. Copy the two WAR files in the directory `<`*`JWSDP_HOME`*`>/registry-server/webapps`, `RegistryServer.war` and `Xindice.war`, to the following directory:

   `<`*`JAVAEE_HOME`*`>/domains/domain1/autodeploy`
3. Start the Application Server.

Any user of a JAXR client can perform queries on a registry. To add data to the registry or to update registry data, however, a user must obtain permission from the registry to access it.

To add or update data in the Java WSDP Registry Server, you can use the default user name and password, `testuser` and `testuser`.

Obtaining a Connection Factory

A client creates a connection from a connection factory. A JAXR provider can supply one or more preconfigured connection factories. Clients can obtain these factories by using resource injection.

At this release of the Application Server, JAXR supplies a connection factory through the JAXR RA, but you need to use a connector resource whose JNDI name is `eis/JAXR` to access this connection factory from a Java EE application. To inject this resource in a Java EE component, use code like the following:

```
import javax.annotation.Resource;.*;
import javax.xml.registry.ConnectionFactory;
...
  @Resource(mappedName="eis/JAXR")
  public ConnectionFactory factory;
```

Later in this chapter you will learn how to create this connector resource.

To use JAXR in a stand-alone client program, you must create an instance of the abstract class `ConnectionFactory`:

```
import javax.xml.registry.ConnectionFactory;
...
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 a hypothetical registry. (There should be no line break in the strings.)

```
Properties props = new Properties();
props.setProperty("javax.xml.registry.queryManagerURL",
  "http://localhost:8080/RegistryServer/");
props.setProperty("javax.xml.registry.lifeCycleManagerURL",
  "http://localhost:8080/RegistryServer/");
```

With the Application Server 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 Application Server 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 Application Server. Tables 6–1 and 6–2 list and describe these properties.

Table 6–1 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

Table 6–1 Standard JAXR Connection Properties (Continued)

Property Name and Description	Data Type	Default Value

`javax.xml.registry.semanticEquivalences`
 	
Specifies semantic equivalences of concepts as one or more tuples of the ID values of two equivalent concepts separated by a comma. The tuples are separated by vertical bars:
`id1,id2|id3,id4`
	String	None

`javax.xml.registry.security.authenticationMethod` Provides a hint to the JAXR provider on the authentication method to be used for authenticating with the registry provider.	String	None; `UDDI_GET_AUTHTOKEN` is the only supported value
`javax.xml.registry.uddi.maxRows` The maximum number of rows to be returned by find operations. Specific to UDDI providers.	String	100
`javax.xml.registry.postalAddressScheme` The ID of a `ClassificationScheme` to be used as the default postal address scheme. See Specifying Postal Addresses (page 197) for an example.	String	None

Table 6–2 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

Table 6–2 Implementation-Specific JAXR Connection Properties (Continued)

Property Name and Description	Data Type	Default Value
`com.sun.xml.registry.https.proxyHost` Specifies the HTTPS proxy host to be used for accessing external registries.	String	Same as HTTP proxy host value
`com.sun.xml.registry.https.proxyPort` Specifies the HTTPS proxy port to be used for accessing external registries; usually 8080.	String	Same as HTTP proxy port value
`com.sun.xml.registry.http.proxyUserName` Specifies the user name for the proxy host for HTTP proxy authentication, if one is required.	String	None
`com.sun.xml.registry.http.proxyPassword` Specifies the password for the proxy host for HTTP proxy authentication, if one is required.	String	None
`com.sun.xml.registry.useCache` Tells the JAXR implementation to look for registry objects in the cache first and then to look in the registry if not found.	Boolean, passed in as String	True
`com.sun.xml.registry.userTaxonomyFile-names` For details on setting this property, see Defining a Taxonomy (page 194).	String	None

Obtaining and Using a RegistryService Object

After creating the connection, the client uses the connection to obtain a `Regis-tryService` 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 as follows:

- `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 service bindings

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 takes 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 sort them in alphabetical order.

```
// Define find qualifiers and name patterns
Collection<String> findQualifiers = new ArrayList<String>();
findQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);
Collection<String> namePatterns = new ArrayList<String>();
namePatterns.add(qString);

// Find orgs whose names begin with qString
BulkResponse response =
  bqm.findOrganizations(findQualifiers, namePatterns, null,
     null, null, null);
Collection orgs = response.getCollection();
```

The last four arguments to `findOrganizations` allow you to search using other criteria than the name: classifications, specification concepts, external identifiers, or external links. Finding Organizations by Classification (page 183) describes searching by classification and by specification concept. The other searches are less common and are not described in this tutorial.

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<String> findQualifiers = new ArrayList<String>();
findQualifiers.add(FindQualifier.CASE_SENSITIVE_MATCH);
Collection<String> namePatterns = new ArrayList<String>();
namePatterns.add("%" + qString + "%");

// Find orgs with names that contain 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 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`.)The NAICS taxonomy has a well-known universally unique identifier (UUID) that is defined by the UDDI specification. The `getRegistryObject` method finds an object based upon its key. (See Creating an Organization, page 186 for more information about keys)

```
String uuid_naics =
  "uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2";
ClassificationScheme cScheme =
  (ClassificationScheme) bqm.getRegistryObject(uuid_naics,
    LifeCycleManager.CLASSIFICATION_SCHEME);
InternationalString sn = blcm.createInternationalString(
  "All Other Specialty Food Stores"));
String sv = "445299";
Classification classification =
  blcm.createClassification(cScheme, sn, sv);
Collection<Classification> classifications =
  new ArrayList<Classification>();
classifications.add(classification);
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 preceding example, because the client must first find the specification concepts and then find 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<Classification> classifications =
  new ArrayList<Classification>();
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<Concept> specConcepts1 =
          new ArrayList<Concept>();
        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 JAX-WS.

Finding Services and Service Bindings

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 = "testuser";
String password = "testuser";

// Get authorization from the registry
PasswordAuthentication passwdAuth =
  new PasswordAuthentication(username,
     password.toCharArray());

HashSet<PasswordAuthentication> creds =
  new HashSet<PasswordAuthentication>();
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 `Postal-Address` 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 to be published to a UDDI registry, it does not include a key; the registry returns the new key when it accepts the newly created organization. The `blcm` object in the following code fragment is the `BusinessLifeCycleManager` object returned in Obtaining and Using a RegistryService Object (page 180). An `International-String` object is used for string values that may need to be localized.

```
// Create organization name and description
InternationalString s =
  blcm.createInternationalString("The Coffee Break");
Organization org = blcm.createOrganization(s);
s = blcm.createInternationalString("Purveyor of the " +
  "finest coffees. Established 1950");
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<TelephoneNumber> phoneNums =
  new ArrayList<TelephoneNumber>();
phoneNums.add(tNum);
primaryContact.setTelephoneNumbers(phoneNums);
```

```
// Set primary contact email address
EmailAddress emailAddress =
  blcm.createEmailAddress("jane.doe@TheCoffeeBreak.com");
Collection<EmailAddress> emailAddresses =
  new ArrayList<EmailAddress>();
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:1997");
```

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 the value of the concept.

```
// Create and add classification
InternationalString sn =
  blcm.createInternationalString(
    “All Other Specialty Food Stores“));
String sv = “445299“;
Classification classification =
  blcm.createClassification(cScheme, sn, sv);
Collection<Classification> classifications =
  new ArrayList<Classification>();
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, a description, and 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 by using the service binding.

The following code fragment shows how to create a collection of services, add service bindings to a service, and 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<Service> services = new ArrayList<Service>();
InternationalString s =
  blcm.createInternationalString("My Service Name"));
Service service = blcm.createService(s);
s = blcm.createInternationalString("My Service Description");
service.setDescription(is);

// Create service bindings
Collection<ServiceBinding> serviceBindings =
  new ArrayList<ServiceBinding>();
ServiceBinding binding = blcm.createServiceBinding();
s = blcm.createInternationalString("My Service Binding " +
  "Description");
binding.setDescription(is);
// allow us to publish a fictitious URI 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<Organization> orgs = new ArrayList<Organization>();
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()) {
    Key orgKey = (Key) keyIter.next();
    String id = orgKey.getId();
    System.out.println("Organization key is " + id);
  }
}
```

Publishing a Specification Concept

A service binding can 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 Simple Web Service and Client with JAX-WS (page xvi). 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, and 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-jaxws/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`, using its key ID. (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 uuid_types =
  "uuid:c1acf26d-9672-4404-9d70-39b756e62ab4";
ClassificationScheme uddiOrgTypes =
  (ClassificationScheme) bqm.getRegistryObject(uuid_types,
    LifeCycleManager.CLASSIFICATION_SCHEME);

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<Concept> concepts = new ArrayList<Concept>();
concepts.add(specConcept);
BulkResponse concResponse = blcm.saveConcepts(concepts);
```

After you have published the concept, you normally add the concept for the WSDL document to a service binding. To do this, you can retrieve the key for the concept from the response returned by the saveConcepts method; you use a code sequence very similar to that of finding the key for a saved organization.

```
String conceptKeyId = null;
Collection concExceptions = concResponse.getExceptions();
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 = (Key) keyIter.next();
    conceptKeyId = concKey.getId();
    System.out.println("Concept key is " + conceptKeyId);
  }
}
```

Then you can 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. Now the organization can be found via queries such as those described in Finding Organizations by Classification (page 183).

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 looks very similar to the code used to search for a WSDL document in Finding Organizations by

Classification (page 183), except that you also create a collection of name patterns and include that in your search. Here is an 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 it 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<Key> keys = new ArrayList<Key>();
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();
  Key orgKey = null;
  if (keyIter.hasNext()) {
    orgKey = (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 Application Server to perform these tasks:

- To define your own taxonomies
- To specify postal addresses for an organization

Defining a Taxonomy

The JAXR specification requires that a JAXR provider 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 Application Server 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 Application Server is defined by the JAXR Pre-defined Concepts DTD, which is declared both in the file `jaxrconcepts.dtd` and, in XML schema form, in the file `jaxrconcepts.xsd`. The file `jaxrcon-cepts.xml` contains the taxonomies for the implementation of JAXR in the Application Server. All these files are contained in the `<`*`JAVAEE_HOME`*`>/lib/appserv-ws.jar` file. This JAR file also includes files that define the well-known taxonomies used by the implementation of JAXR in the Application Server: `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 `Pre-definedConcepts` is the root of the structure and must be present. The `JAXR-ClassificationScheme` 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. 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.

```
InternationalString sn =
   blcm.createInternationalString("MyScheme");
InternationalString sd = blcm.createInternationalString(
   "A Classification Scheme");
ClassificationScheme postalScheme =
   blcm.createClassificationScheme(sn, sd);
String uuid_types =
   "uuid:c1acf26d-9672-4404-9d70-39b756e62ab4";
ClassificationScheme uddiOrgTypes =
   (ClassificationScheme) bqm.getRegistryObject(uuid_types,
      LifeCycleManager.CLASSIFICATION_SCHEME);
if (uddiOrgTypes != null) {
   Classification classification =
      blcm.createClassification(uddiOrgTypes,
         "postalAddress", "postalAddress" );
   postalScheme.addClassification(classification);
   InternationalString ld =
      blcm.createInternationalString("My Scheme");
   ExternalLink externalLink =
      blcm.createExternalLink(
         "http://www.mycom.com/myscheme.xml", ld);
   postalScheme.addExternalLink(externalLink);
   Collection<ClassificationScheme> schemes =
      new ArrayList<ClassificationScheme>();
   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()) {
      Key key = (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 foregoing code fragment, 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 universally unique identifier (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 connection property `com.sun.xml.registry.userTaxonomyFilenames` in your client program. You set the property as follows:

   ```
   props.setProperty
   ("com.sun.xml.registry.userTaxonomyFilenames",
      "c:\mydir\xxx.xml|c:\mydir\xxx2.xml");
   ```

   Use the vertical bar (`|`) as a separator if you specify more than one file name.

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 can 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 Application Server.

In the JAXR API, a `PostalAddress` object has the fields `streetNumber`, `street`, `city`, `state`, `postalCode`, and `country`. In the implementation of JAXR in the Application Server, 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 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 scheme you published:

```
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<PostalAddress> postalAddresses =
  new ArrayList<PostalAddress>();
postalAddresses.add(postAddr);
primaryContact.setPostalAddresses(postalAddresses);
```

If the postal address scheme and semantic equivalences for the query are the same as those specified for the publication, a JAXR query can then retrieve the postal address using `PostalAddress` methods. 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 Java WSDP Registry Server for queries and updates. (To install the Registry Server, follow the instructions in Preliminaries: Getting Access to a Registry (page 175).

The examples, in the `<INSTALL>/javaeetutorial5/examples/jaxr/simple/src/` 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>/javaeetutorial5/examples/jaxr/simple/` directory also contains the following:

- A `build.xml` file for the examples
- A `JAXRExamples.properties` file, in the `src` subdirectory, that supplies string values used by the sample programs
- A file called `postalconcepts.xml` that serves as the taxonomy file for the postal address examples

Before You Compile the Examples

Before you compile the examples, edit the file `<INSTALL>/javaeetutorial5/examples/jaxr/simple/src/JAXRExamples.properties` as follows.

1. If the Application Server where you installed the Registry Server is running on a system other than your own or if itis using a nondefault HTTP port, change the following lines:

   ```
   query.url=http://localhost:8080/RegistryServer/
   publish.url=http://localhost:8080/RegistryServer/
   ...
   link.uri=http://localhost:8080/hello-jaxws/hello?WSDL
   ...
   wsdlorg.svcbnd.uri=http://localhost:8080/hello-jaxws/hello
   ```

   Specify the fully qualified host name instead of `localhost`, or change `8080` to the correct value for your system.

2. (Optional) 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.

   ```
   ## HTTP and HTTPS proxy host and port
   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.

Your entries usually follow this pattern:

```
http.proxyHost=proxyhost.mydomain
http.proxyPort=8080
https.proxyHost=proxyhost.mydomain
https.proxyPort=8080
```

You need to specify a proxy only if you want to specify an external link or service binding that is outside your firewall.

3. 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. Try to make the organization names unusual so that queries will return relatively few results.

You can edit the `src/JAXRExamples.properties` file at any time. The `asant` targets that run the client examples will use the latest version of the file.

Note: Before you compile any of the examples, follow the preliminary setup instructions in Building the Examples (page xxxiii).

Compiling the Examples

To compile the programs, go to the `<INSTALL>/javaeetutorial5/examples/jaxr/simple/` directory. A `build.xml` file allows you to use the following command to compile all the examples:

```
asant
```

This command uses the default target, `build`, which performs the compilation. The `asant` tool creates a subdirectory called `build`.

Running the Examples

You must start the Application Server in order to run the examples against the Registry Server.

Running the JAXRPublish Example

To run the JAXRPublish program, use the run-publish target with no command-line arguments:

```
asant run-publish
```

The program output displays the string value of the key of the new organization.

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 asant 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 "coffee" (searching is not case-sensitive):

```
asant -Dquery-string=coffee 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 All Other Specialty Food Stores classification, the same one used for the organization created by JAXRPublish. To do so, use the asant target run-query-naics:

```
asant run-query-naics
```

Running the JAXRDelete Example

To run the JAXRDelete program, specify the key string displayed by the JAXRPublish program as input to the run-delete target:

```
asant -Dkey-string=keyString run-delete
```

Publishing a Classification Scheme

To publish organizations with postal addresses, you must first publish a classification scheme for the postal address.

To run the `JAXRSaveClassificationScheme` program, use the target `run-save-scheme`:

```
asant run-save-scheme
```

The program returns a UUID string, which you will use in the next section.

Running the Postal Address Examples

Before you run the postal address examples, perform these steps:

1. Open the file `src/postalconcepts.xml` in an editor.
2. 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 a given registry, you only need to publish the classification scheme and edit `postalconcepts.xml` once. After you perform those steps, you can run the `JAXRPublishPostal` and `JAXRQueryPostal` programs multiple times.

1. Run the `JAXRPublishPostal` program. Specify the string you entered in the `postalconcepts.xml` file, including the `uuid:` prefix, as input to the `run-publish-postal` target:

   ```
   asant -Duuid-string=uuidstring run-publish-postal
   ```

   The *`uuidstring`* would look something like this:

   ```
   uuid:938d9ccd-a74a-4c7e-864a-e6e2c6822519
   ```

   The program output displays the string value of the key of the new organization.

2. Run the `JAXRQueryPostal` program. The `run-query-postal` target specifies the `postalconcepts.xml` file in a <sysproperty> tag.

   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:

   ```
   asant -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.

If you want to delete the organization you published, follow the instructions in Running the JAXRDelete Example (page 202).

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:

```
asant -Duuid-string=uuidstring run-delete-scheme
```

Publishing a Concept for a WSDL Document

To publish the location of the WSDL document for the JAX-WS `Hello` service, first deploy the service to the Application Server as described in Creating a Simple Web Service and Client with JAX-WS (page xvi).

Then run the `JAXRPublishConcept` program using the `run-publish-concept` target:

```
asant run-publish-concept
```

The program output displays the UUID string of the new specification concept, which is named HelloConcept. You will use this string in the next section.

After you run the `JAXRPublishConcept` program, you can run `JAXRPublish-HelloOrg` 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 `asant` target `run-publish-hello-org`. Specify the string returned from `JAXRPublishConcept` (including the `uuid:` prefix) as input to this target:

```
asant -Duuid-string=uuidstring run-publish-hello-org
```

The *uuidstring* would look something like this:

```
uuid:10945f5c-f2e1-0945-2f07-5897ebcfaa35
```

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 `asant` 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):

```
asant -Dquery-string=helloconcept run-query-wsdl
```

This example finds the concept and organization you published.

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:

```
asant -Duuid-string=uuidString run-delete-concept
```

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:

```
asant run-get-objects
```

Other Targets

To remove the `build` directory and class files, use the command

```
asant clean
```

To obtain a syntax reminder for the targets, use the command

```
asant -projecthelp
```

Using JAXR Clients in Java EE Applications

You can create Java EE applications that use JAXR clients to access registries. This section explains how to write, compile, package, deploy, and run a Java EE application that uses JAXR to publish an organization to a registry and then query the registry for that organization. The application in this section uses two components: an application client and a stateless session bean.

The section covers the following topics:

- Coding the application client: `MyAppClient.java`
- Coding the `PubQuery` session bean
- Compiling the source files
- Starting the Application Server
- Creating JAXR resources
- Creating and packaging the application
- Deploying the application
- Running the application client

You will find the source files for this section in the directory `<`*`INSTALL`*`>/javaeetutorial5/examples/jaxr/clientsession`. Path names in this section are relative to this directory.

The following directory contains a built version of this application:

```
<INSTALL>/javaeetutorial5/examples/jaxr/provided-ears
```

Coding the Application Client: MyAppClient.java

The application client class, `src/MyAppClient.java`, accesses the `PubQuery` enterprise bean's remote interface. The program calls the bean's two business methods, `executePublish` and `executeQuery`.

Coding the PubQuery Session Bean

The `PubQuery` bean is a stateless session bean that has two business methods. The bean uses remote interfaces rather than local interfaces because it is accessed from the application client.

The remote interface, `src/PubQueryRemote.java`, declares two business methods: `executePublish` and `executeQuery`. The bean class, `src/PubQueryBean.java`, implements the `executePublish` and `executeQuery` methods and their helper methods `getName`, `getDescription`, and `getKey`. These methods are very similar to the methods of the same name in the simple examples `JAXRQuery.java` and `JAXRPublish.java`. The `executePublish` method uses information in the file `PubQueryBeanExample.properties` to create an organization named The Coffee Enterprise Bean Break. The `executeQuery` method uses the organization name, specified in the application client code, to locate this organization.

The bean class injects a `ConnectionFactory` resource. It implements a `@PostConstruct` method named `makeConnection`, which uses the `ConnectionFactory` to create the `Connection`. Finally, a `@PreDestroy` method named `endConnection` closes the `Connection`.

Editing the Properties File

Before you compile the application, edit the `PubQueryBeanExamples.properties` file in the same way you edited the `JAXRExamples.properties` file to run the simple examples. Feel free to change any of the organization data in the file.

Compiling the Source Files

To compile the application source files, go to the directory `<INSTALL>/javaeetutorial5/examples/jaxr/clientsession`. Use the following command:

```
asant build
```

The `build` target places the properties file and the class files in the `build` directory.

Starting the Application Server

To run this example, you need to start the Application Server. Follow the instructions in Starting and Stopping the Application Server (page 28).

Creating JAXR Resources

To use JAXR in a Java EE application that uses the Application Server, you need to access the JAXR resource adapter (see Implementing a JAXR Client, page 174) through a connector connection pool and a connector resource. You can create these resources in the Admin Console.

If you have not done so, start the Admin Console as described in Starting the Admin Console (page 29).

To create the connector connection pool, perform the following steps:

1. In the tree component, expand the Resources node, then expand the Connectors node.
2. Click Connector Connection Pools.
3. Click New.
4. On the General Settings page:
    a. Type `jaxr-pool` in the Name field.
    b. Choose `jaxr-ra` from the Resource Adapter drop-down list.
    c. Choose `com.sun.connector.jaxr.JaxrConnectionFactory` (the only choice) from the Connection Definition drop-down list
    d. Click Next.
5. On the next page, click Finish.

To create the connector resource, perform the following steps:

1. Under the Connectors node, click Connector Resources.
2. Click New. The Create Connector Resource page appears.
3. In the JNDI Name field, type `eis/JAXR`.
4. Choose `jaxr-pool` from the Pool Name drop-down list.
5. Click OK.

If you are in a hurry, you can create these objects by executing the following command (from the directory `<INSTALL>/javaeetutorial5/examples/jaxr/clientsession`):

```
asant create-resource
```

Packaging the Application

The `build.xml` file in the `clientsession` directory defines Ant targets that package the `clientsession` application. To package the application, use the following command:

```
asant pack-ear
```

The `pack-ear` target depends on the `pack-client` and `pack-ejb` targets, which in turn depend on the `build` target.

The `pack-client` target creates a JAR file that contains the client class file, a manifest file, and the `PubQueryBeanExample.properties` file.

The `pack-ejb` target packages the session bean. It creates a JAR file that contains the bean class files, a manifest file, and the `PubQueryBeanExample.properties` file.

The `pack-ear` target packages the two JAR files along with an `application.xml` file. It creates a file named `clientsession.ear` in the `clientsession` directory.

Deploying the Application

The `build.xml` file in the `clientsession` directory defines an Ant target that deploys the `clientsession.ear` file and returns a client JAR file. Use the following command:

```
asant deploy-ear
```

This command deploys the application and returns a JAR file named `clientsessionClient.jar` in the `clientsession` directory.

Running the Application Client

To run the client, use the following command:

```
appclient -client clientsessionClient.jar
```

The program output in the terminal window looks like this:

```
To view the bean output,
 check <install_dir>/domains/domain1/logs/server.log.
```

In the server log, you will find the output from the `executePublish` and `executeQuery` methods, wrapped in logging information.

After you run the example, use the following command to undeploy the application:

```
asant undeploy-ear
```

You can use the `run-delete` target in the `simple` directory to delete the organization that was published.

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/`

- Universal Description, Discovery and Integration (UDDI) project:
  `http://www.uddi.org/`
- ebXML:
  `http://www.ebxml.org/`
- Service Registry (ebXML Registry/Repository):
  `http://www.sun.com/products/soa/registry/`
- Open Source JAXR Provider for ebXML Registries:
  `http://ebxmlrr.sourceforge.net/jaxr/`
- Java Platform, Enterprise Edition:
  `http://java.sun.com/javaee/`
- Java Technology and XML:
  `http://java.sun.com/xml/`
- Java Technology and Web Services:
  `http://java.sun.com/webservices/`

7

Java XML Digital Signature API

THE Java XML Digital Signature API is a standard Java API for generating and validating XML Signatures. This API was defined under the Java Community Process as JSR 105 (see `http://jcp.org/en/jsr/detail?id=105`). This JSR is final and this release of Java WSDP contains an FCS access implementation of the Final version of the APIs.

XML Signatures can be applied to data of any type, XML or binary (see `http://www.w3.org/TR/xmldsig-core/`). The resulting signature is represented in XML. An XML Signature can be used to secure your data and provide data integrity, message authentication, and signer authentication.

After providing a brief overview of XML Signatures and the XML Digital Signature API, this chapter presents two examples that demonstrate how to use the API to validate and generate an XML Signature. This chapter assumes that you have a basic knowledge of cryptography and digital signatures.

The API is designed to support all of the required or recommended features of the W3C Recommendation for XML-Signature Syntax and Processing. The API is extensible and pluggable and is based on the Java Cryptography Service Provider Architecture. The API is designed for two types of developers:

- Java programmers who want to use the XML Digital Signature API to generate and validate XML signatures

- Java programmers who want to create a concrete implementation of the XML Digital Signature API and register it as a cryptographic service of a JCA provider (see `http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html#Provider`)

How XWS-Security and XML Digital Signature API Are Related

Before getting into specifics, it is important to see how XWS-Security and XML Digital Signature API are related. In this release of the Java WSDP, XWS-Security is based on non-standard XML Digital Signature APIs.

XML Digital Signature API is an API that should be used by Java applications and middleware that need to create and/or process XML Signatures. It can be used by Web Services Security (the goal for a future release) and by non-Web Services technologies (for example, signing documents stored or transferred in XML). Both JSR 105 and JSR 106 (XML Digital Encryption APIs) are core-XML security components. (See `http://www.jcp.org/en/jsr/detail?id=106` for more information about JSR 106.)

XWS-Security does not currently use the XML Digital Signature API or XML Digital Encryption APIs. XWS-Security uses the Apache libraries for XML-DSig and XML-Enc. The goal of XWS-Security is to move toward using these APIs in future releases.

XML Security Stack

Figure 7–1 shows how XML Digital Signature API (JSR 105) interacts with security components today and how it will interact with other security components, including XML Digital Encryption API (JSR 106), in future releases.

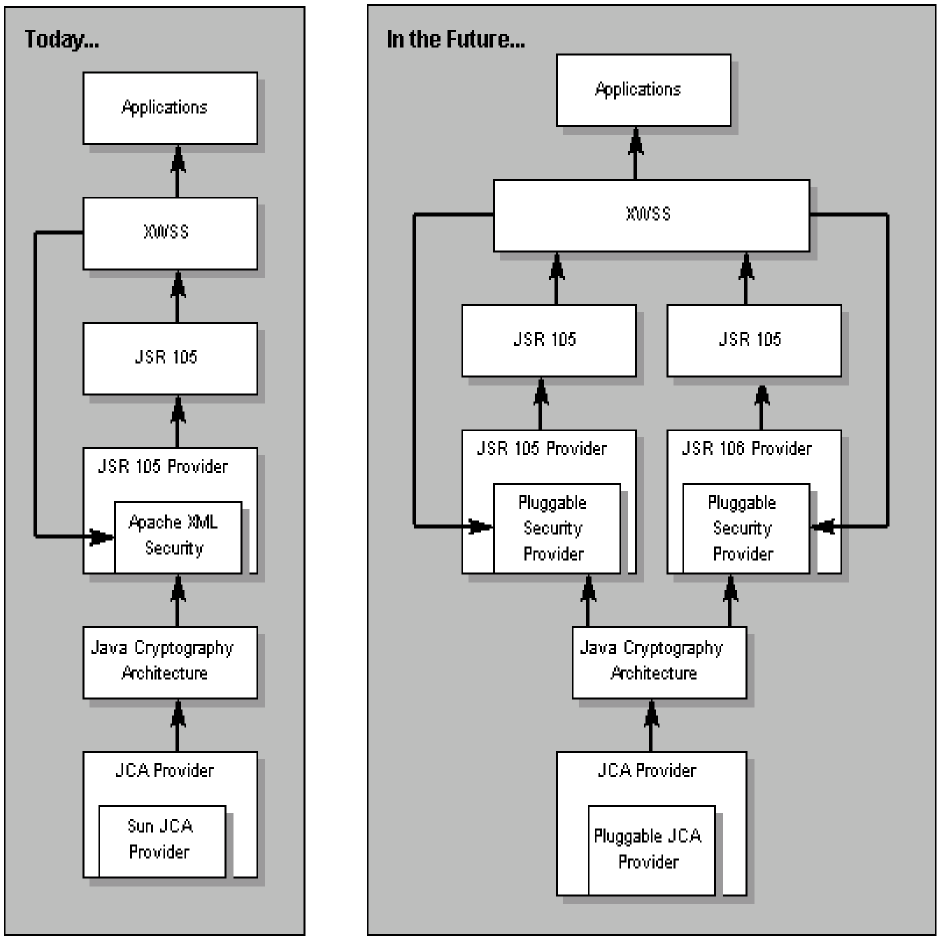

Figure 7–1 Java WSDP Security Components

XWSS calls Apache XML-Security directly today; in future releases, it should be able to call other pluggable security providers. The Apache XML-Security provider and the Sun JCA Provider are both pluggable components. Since JSR 105 is final today, the JSR 105 layer is standard now; the JSR 106 layer will be standard after that JSR becomes final.

Package Hierarchy

The six packages in the XML Digital Signature API are:

- `javax.xml.crypto`
- `javax.xml.crypto.dsig`
- `javax.xml.crypto.dsig.keyinfo`
- `javax.xml.crypto.dsig.spec`
- `javax.xml.crypto.dom`
- `javax.xml.crypto.dsig.dom`

The `javax.xml.crypto` package contains common classes that are used to perform XML cryptographic operations, such as generating an XML signature or encrypting XML data. Two notable classes in this package are the `KeySelector` class, which allows developers to supply implementations that locate and optionally validate keys using the information contained in a `KeyInfo` object, and the `URIDereferencer` class, which allows developers to create and specify their own URI dereferencing implementations.

The `javax.xml.crypto.dsig` package includes interfaces that represent the core elements defined in the W3C XML digital signature specification. Of primary significance is the `XMLSignature` class, which allows you to sign and validate an XML digital signature. Most of the XML signature structures or elements are represented by a corresponding interface (except for the `KeyInfo` structures, which are included in their own package and are discussed in the next paragraph). These interfaces include: `SignedInfo`, `CanonicalizationMethod`, `SignatureMethod`, `Reference`, `Transform`, `DigestMethod`, `XMLObject`, `Manifest`, `SignatureProperty`, and `SignatureProperties`. The `XMLSignatureFactory` class is an abstract factory that is used to create objects that implement these interfaces.

The `javax.xml.crypto.dsig.keyinfo` package contains interfaces that represent most of the `KeyInfo` structures defined in the W3C XML digital signature recommendation, including `KeyInfo`, `KeyName`, `KeyValue`, `X509Data`, `X509IssuerSerial`, `RetrievalMethod`, and `PGPData`. The `KeyInfoFactory` class is an abstract factory that is used to create objects that implement these interfaces.

The `javax.xml.crypto.dsig.spec` package contains interfaces and classes representing input parameters for the digest, signature, transform, or canonicalization algorithms used in the processing of XML signatures.

Finally, the `javax.xml.crypto.dom` and `javax.xml.crypto.dsig.dom` packages contains DOM-specific classes for the `javax.xml.crypto` and javax.xml.crypto.dsig packages, respectively. Only developers and users who are creating or using a DOM-based XMLSignatureFactory or KeyInfo-Factory implementation should need to make direct use of these packages.

Service Providers

A JSR 105 cryptographic service is a concrete implementation of the abstract XMLSignatureFactory and KeyInfoFactory classes and is responsible for creating objects and algorithms that parse, generate and validate XML Signatures and KeyInfo structures. A concrete implementation of XMLSignatureFactory *must* provide support for each of the *required* algorithms as specified by the W3C recommendation for XML Signatures. It *may* support other algorithms as defined by the W3C recommendation or other specifications.

JSR 105 leverages the JCA provider model for registering and loading XMLSignatureFactory and KeyInfoFactory implementations.

Each concrete XMLSignatureFactory or KeyInfoFactory implementation supports a specific XML mechanism type that identifies the XML processing mechanism that an implementation uses internally to parse and generate XML signature and KeyInfo structures. This JSR supports one standard type, DOM. The XML Digital Signature API early access provider implementation that is bundled with Java WSDP supports the DOM mechanism. Support for new standard types, such as JDOM, may be added in the future.

An XML Digital Signature API implementation *should* use underlying JCA engine classes, such as java.security.Signature and java.security.MessageDigest, to perform cryptographic operations.

In addition to the XMLSignatureFactory and KeyInfoFactory classes, JSR 105 supports a service provider interface for transform and canonicalization algorithms. The TransformService class allows you to develop and plug in an implementation of a specific transform or canonicalization algorithm for a particular XML mechanism type. The TransformService class uses the standard JCA provider model for registering and loading implementations. Each JSR 105 implementation *should* use the TransformService class to find a provider that supports transform and canonicalization algorithms in XML Signatures that it is generating or validating.

Introduction to XML Signatures

As mentioned, an XML Signature can be used to sign any arbitrary data, whether it is XML or binary. The data is identified via URIs in one or more Reference elements. XML Signatures are described in one or more of three forms: detached, enveloping, or enveloped. A detached signature is over data that is external, or outside of the signature element itself. Enveloping signatures are signatures over data that is inside the signature element, and an enveloped signature is a signature that is contained inside the data that it is signing.

Example of an XML Signature

The easiest way to describe the contents of an XML Signature is to show an actual sample and describe each component in more detail. The following is an example of an enveloped XML Signature generated over the contents of an XML document. The contents of the document before it is signed are:

```
<Envelope xmlns="urn:envelope">
</Envelope>
```

The resulting enveloped XML Signature, indented and formatted for readability, is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Envelope xmlns="urn:envelope">
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod
           Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-
20010315#WithComments"/>
      <SignatureMethod Algorithm="http://www.w3.org/2000/09/
xmldsig#dsa-sha1"/>
      <Reference URI="">
        <Transforms>
          <Transform Algorithm="http://www.w3.org/2000/09/
xmldsig#enveloped-signature"/>
        </Transforms>
        <DigestMethod Algorithm="http://www.w3.org/2000/09/
xmldsig#sha1"/>
       <DigestValue>uooqbWYa5VCqcJCbuymBKqm17vY=</DigestValue>
      </Reference>
    </SignedInfo>
<SignatureValue>
```

```
KedJuTob5gtvYx9qM3k3gm7kbLBwVbEQRl26S2tmXjqNND7MRGtoew==
    </SignatureValue>
    <KeyInfo>
      <KeyValue>
        <DSAKeyValue>
          <P>
/KaCzo4Syrom78z3EQ5SbbB4sF7ey80etKII864WF64B81uRpH5t9jQTxe
Eu0ImbzRMqzVDZkVG9xD7nN1kuFw==
          </P>
          <Q>li7dzDacuo67Jg7mtqEm2TRuOMU=</Q>
          <G>Z4Rxsnqc9E7pGknFFH2xqaryRPBaQ01khpMdLRQnG541Awtx/
XPaF5Bpsy4pNWMOHCBiNU0NogpsQW5QvnlMpA==
          </G>
          <Y>qV38IqrWJG0V/
mZQvRVi1OHw9Zj84nDC4jO8P0axi1gb6d+475yhMjSc/
BrIVC58W3ydbkK+Ri4OKbaRZlYeRA==
         </Y>
        </DSAKeyValue>
      </KeyValue>
    </KeyInfo>
  </Signature>
</Envelope>
```

The `Signature` element has been inserted inside the content that it is signing, thereby making it an enveloped signature. The required `SignedInfo` element contains the information that is actually signed:

```
<SignedInfo>
  <CanonicalizationMethod
    Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-
20010315#WithComments"/>
  <SignatureMethod Algorithm="http://www.w3.org/2000/09/
xmldsig#dsa-sha1"/>
  <Reference URI="">
      <Transforms>
        <Transform Algorithm="http://www.w3.org/2000/09/
xmldsig#enveloped-signature"/>
      </Transforms>
      <DigestMethod Algorithm="http://www.w3.org/2000/09/
xmldsig#sha1"/>
      <DigestValue>uooqbWYa5VCqcJCbuymBKqm17vY=</DigestValue>
  </Reference>
</SignedInfo>
```

The required `CanonicalizationMethod` element defines the algorithm used to canonicalize the `SignedInfo` element before it is signed or validated. Canonicalization is the process of converting XML content to a canonical form, to take into account changes that can invalidate a signature over that data. Canonicalization is necessary due to the nature of XML and the way it is parsed by different processors and intermediaries, which can change the data such that the signature is no longer valid but the signed data is still logically equivalent.

The required `SignatureMethod` element defines the digital signature algorithm used to generate the signature, in this case DSA with SHA-1.

One or more `Reference` elements identify the data that is digested. Each `Reference` element identifies the data via a URI. In this example, the value of the URI is the empty String (""), which indicates the root of the document. The optional `Transforms` element contains a list of one or more `Transform` elements, each of which describes a transformation algorithm used to transform the data before it is digested. In this example, there is one `Transform` element for the enveloped transform algorithm. The enveloped transform is required for enveloped signatures so that the signature element itself is removed before calculating the signature value. The required `DigestMethod` element defines the algorithm used to digest the data, in this case SHA1. Finally the required `DigestValue` element contains the actual base64-encoded digested value.

The required `SignatureValue` element contains the base64-encoded signature value of the signature over the `SignedInfo` element.

The optional `KeyInfo` element contains information about the key that is needed to validate the signature:

```
<KeyInfo>
  <KeyValue>
    <DSAKeyValue>
      <P>
/KaCzo4Syrom78z3EQ5SbbB4sF7ey80etKII864WF64B81uRpH5t9jQTxe
Eu0ImbzRMqzVDZkVG9xD7nN1kuFw==
      </P>
      <Q>li7dzDacuo67Jg7mtqEm2TRuOMU=</Q>
      <G>Z4Rxsnqc9E7pGknFFH2xqaryRPBaQ01khpMdLRQnG541Awtx/
XPaF5Bpsy4pNWMOHCBiNU0NogpsQW5QvnlMpA==
      </G>
      <Y>
qV38IqrWJG0V/mZQvRVi1OHw9Zj84nDC4jO8P0axi1gb6d+475yhMjSc/
BrIVC58W3ydbkK+Ri4OKbaRZlYeRA==
      </Y>
    </DSAKeyValue>
  </KeyValue>
</KeyInfo>
```

This `KeyInfo` element contains a `KeyValue` element, which in turn contains a `DSAKeyValue` element consisting of the public key needed to validate the signature. `KeyInfo` can contain various content such as X.509 certificates and PGP key identifiers. See the `KeyInfo section` of the XML Signature Recommendation for more information on the different `KeyInfo` types.

XML Digital Signature API Examples

The following sections describe two examples that show how to use the XML Digital Signature API:

- Validate example
- Signing example

To run the sample applications using the supplied Ant `build.xml` files, issue the following commands after you installed Java WSDP:

For Solaris/Linux:

1. `% export JWSDP_HOME=<`*`your Java WSDP installation directory`*`>`
2. `% export ANT_HOME=$JWSDP_HOME/apache-ant`
3. `% export PATH=$ANT_HOME/bin:$PATH`
4. `% cd $JWSDP_HOME/xmldsig/samples/<`*`sample-name`*`>`

For Windows 2000/XP:

1. `> set JWSDP_HOME=<`*`your Java WSDP installation directory`*`>`
2. `> set ANT_HOME=%JWSDP_HOME%\apache-ant`
3. `> set PATH=%ANT_HOME%\bin;%PATH%`
4. `> cd %JWSDP_HOME%\xmldsig\samples\<`*`sample-name`*`>`

validate Example

You can find the code shown in this section in the `Validate.java` file in the `<`*`JWSDP_HOME`*`>/xmldsig/samples/validate` directory. The file on which it operates, `envelopedSignature.xml`, is in the same directory.

To run the example, execute the following command from the `<`*`JWSDP_HOME`*`>/xmldsig/samples/validate` directory:

```
$ ant
```

The sample program will validate the signature in the file envelopedSignature.xml in the current working directory. To validate a different signature, run the following command:

```
$ ant -Dsample.args="signature.xml"
```

where "signature.xml" is the pathname of the file.

Validating an XML Signature

This example shows you how to validate an XML Signature using the JSR 105 API. The example uses DOM (the Document Object Model) to parse an XML document containing a Signature element and a JSR 105 DOM implementation to validate the signature.

Instantiating the Document that Contains the Signature

First we use a JAXP DocumentBuilderFactory to parse the XML document containing the Signature. An application obtains the default implementation for DocumentBuilderFactory by calling the following line of code:

```
DocumentBuilderFactory dbf =
  DocumentBuilderFactory.newInstance();
```

We must also make the factory namespace-aware:

```
dbf.setNamespaceAware(true);
```

Next, we use the factory to get an instance of a DocumentBuilder, which is used to parse the document:

```
DocumentBuilder builder = dbf.newDocumentBuilder();
Document doc = builder.parse(new FileInputStream(argv[0]));
```

Specifying the Signature Element to be Validated

We need to specify the Signature element that we want to validate, since there could be more than one in the document. We use the DOM method Document.getElementsByTagNameNS, passing it the XML Signature namespace URI and the tag name of the Signature element, as shown:

```
NodeList nl = doc.getElementsByTagNameNS
  (XMLSignature.XMLNS, "Signature");
if (nl.getLength() == 0) {
  throw new Exception("Cannot find Signature element");
}
```

This returns a list of all Signature elements in the document. In this example, there is only one Signature element.

Creating a Validation Context

We create an XMLValidateContext instance containing input parameters for validating the signature. Since we are using DOM, we instantiate a DOMValidateContext instance (a subclass of XMLValidateContext), and pass it two parameters, a KeyValueKeySelector object and a reference to the Signature element to be validated (which is the first entry of the NodeList we generated earlier):

```
DOMValidateContext valContext = new DOMValidateContext
  (new KeyValueKeySelector(), nl.item(0));
```

The KeyValueKeySelector is explained in greater detail in Using KeySelectors (page 225).

Unmarshaling the XML Signature

We extract the contents of the Signature element into an XMLSignature object. This process is called unmarshalling. The Signature element is unmarshalled using an XMLSignatureFactory object. An application can obtain a DOM implementation of XMLSignatureFactory by calling the following line of code:

```
XMLSignatureFactory factory =
  XMLSignatureFactory.getInstance("DOM");
```

We then invoke the `unmarshalXMLSignature` method of the factory to unmarshal an `XMLSignature` object, and pass it the validation context we created earlier:

```
XMLSignature signature =
  factory.unmarshalXMLSignature(valContext);
```

Validating the XML Signature

Now we are ready to validate the signature. We do this by invoking the `validate` method on the `XMLSignature` object, and pass it the validation context as follows:

```
boolean coreValidity = signature.validate(valContext);
```

The `validate` method returns "true" if the signature validates successfully according to the `core validation rules` in the `W3C XML Signature Recommendation`, and false otherwise.

What If the XML Signature Fails to Validate?

If the `XMLSignature.validate` method returns false, we can try to narrow down the cause of the failure. There are two phases in core XML Signature validation:

- `Signature validation` (the cryptographic verification of the signature)
- `Reference validation` (the verification of the digest of each reference in the signature)

Each phase must be successful for the signature to be valid. To check if the signature failed to cryptographically validate, we can check the status, as follows:

```
boolean sv =
  signature.getSignatureValue().validate(valContext);
System.out.println("signature validation status: " + sv);
```

We can also iterate over the references and check the validation status of each one, as follows:

```
Iterator i =
  signature.getSignedInfo().getReferences().iterator();
for (int j=0; i.hasNext(); j++) {
  boolean refValid = ((Reference)
    i.next()).validate(valContext);
  System.out.println("ref["+j+"] validity status: " +
     refValid);
}
```

Using KeySelectors

KeySelectors are used to find and select keys that are needed to validate an XMLSignature. Earlier, when we created a DOMValidateContext object, we passed a KeySelector object as the first argument:

```
DOMValidateContext valContext = new DOMValidateContext
  (new KeyValueKeySelector(), nl.item(0));
```

Alternatively, we could have passed a PublicKey as the first argument if we already knew what key is needed to validate the signature. However, we often don't know.

The KeyValueKeySelector is a concrete implementation of the abstract KeySelector class. The KeyValueKeySelector implementation tries to find an appropriate validation key using the data contained in KeyValue elements of the KeyInfo element of an XMLSignature. It does not determine if the key is trusted. This is a very simple KeySelector implementation, designed for illustration rather than real-world usage. A more practical example of a KeySelector is one that searches a KeyStore for trusted keys that match X509Data information (for example, X509SubjectName, X509IssuerSerial, X509SKI, or X509Certificate elements) contained in a KeyInfo.

The implementation of the KeyValueKeySelector is as follows:

```
private static class KeyValueKeySelector extends KeySelector {

  public KeySelectorResult select(KeyInfo keyInfo,
        KeySelector.Purpose purpose,
        AlgorithmMethod method,
        XMLCryptoContext context)
      throws KeySelectorException {

      if (keyInfo == null) {
        throw new KeySelectorException("Null KeyInfo object!");
      }
      SignatureMethod sm = (SignatureMethod) method;
      List list = keyInfo.getContent();

      for (int i = 0; i < list.size(); i++) {
            XMLStructure xmlStructure = (XMLStructure) list.get(i);
            if (xmlStructure instanceof KeyValue) {
               PublicKey pk = null;
               try {
                  pk = ((KeyValue)xmlStructure).getPublicKey();
               } catch (KeyException ke) {
                  throw new KeySelectorException(ke);
               }
               // make sure algorithm is compatible with method
               if (algEquals(sm.getAlgorithm(),
                     pk.getAlgorithm())) {
                  return new SimpleKeySelectorResult(pk);
               }
            }
         }
         throw new KeySelectorException("No KeyValue element
found!");
      }

      static boolean algEquals(String algURI, String algName) {
         if (algName.equalsIgnoreCase("DSA") &&
               algURI.equalsIgnoreCase(SignatureMethod.DSA_SHA1)) {
            return true;
         } else if (algName.equalsIgnoreCase("RSA") &&
               algURI.equalsIgnoreCase(SignatureMethod.RSA_SHA1)) {
            return true;
         } else {
            return false;
         }
      }
   }
```

genenveloped Example

The code discussed in this section is in the `GenEnveloped.java` file in the `<JWSDP_HOME>/xmldsig/samples/genenveloped` directory. The file on which it operates, `envelope.xml`, is in the same directory. It generates the file `envelopedSignature.xml`.

To compile and run this sample, execute the following command from the `<JWSDP_HOME>/xmldsig/samples/genenveloped` directory:

```
$ ant
```

The sample program will generate an enveloped signature of the document in the file `envelope.xml` and store it in the file `envelopedSignature.xml` in the current working directory.

Generating an XML Signature

This example shows you how to generate an XML Signature using the XML Digital Signature API. More specifically, the example generates an enveloped XML Signature of an XML document. An enveloped signature is a signature that is contained inside the content that it is signing. The example uses DOM (the Document Object Model) to parse the XML document to be signed and a JSR 105 DOM implementation to generate the resulting signature.

A basic knowledge of XML Signatures and their different components is helpful for understanding this section. See `http://www.w3.org/TR/xmldsig-core/` for more information.

Instantiating the Document to be Signed

First, we use a JAXP `DocumentBuilderFactory` to parse the XML document that we want to sign. An application obtains the default implementation for `DocumentBuilderFactory` by calling the following line of code:

```
DocumentBuilderFactory dbf =
  DocumentBuilderFactory.newInstance();
```

We must also make the factory namespace-aware:

```
dbf.setNamespaceAware(true);
```

Next, we use the factory to get an instance of a `DocumentBuilder`, which is used to parse the document:

```
DocumentBuilder builder = dbf.newDocumentBuilder();
Document doc = builder.parse(new FileInputStream(argv[0]));
```

Creating a Public Key Pair

We generate a public key pair. Later in the example, we will use the private key to generate the signature. We create the key pair with a `KeyPairGenerator`. In this example, we will create a DSA `KeyPair` with a length of 512 bytes :

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
kpg.initialize(512);
KeyPair kp = kpg.generateKeyPair();
```

In practice, the private key is usually previously generated and stored in a `KeyStore` file with an associated public key certificate.

Creating a Signing Context

We create an XML Digital Signature `XMLSignContext` containing input parameters for generating the signature. Since we are using DOM, we instantiate a `DOMSignContext` (a subclass of `XMLSignContext`), and pass it two parameters, the private key that will be used to sign the document and the root of the document to be signed:

```
DOMSignContext dsc = new DOMSignContext
  (kp.getPrivate(), doc.getDocumentElement());
```

Assembling the XML Signature

We assemble the different parts of the `Signature` element into an `XMLSignature` object. These objects are all created and assembled using an `XMLSignatureFactory` object. An application obtains a DOM implementation of `XMLSignatureFactory` by calling the following line of code:

```
XMLSignatureFactory fac =
  XMLSignatureFactory.getInstance("DOM");
```

We then invoke various factory methods to create the different parts of the `XMLSignature` object as shown below. We create a `Reference` object, passing to it the following:

- The URI of the object to be signed (We specify a URI of "", which implies the root of the document.)
- The `DigestMethod` (we use SHA1)

- A single `Transform`, the enveloped `Transform`, which is required for enveloped signatures so that the signature itself is removed before calculating the signature value

```
Reference ref = fac.newReference
  ("", fac.newDigestMethod(DigestMethod.SHA1, null),
    Collections.singletonList
      (fac.newTransform(Transform.ENVELOPED,
        (TransformParameterSpec) null)), null, null);
```

Next, we create the `SignedInfo` object, which is the object that is actually signed, as shown below. When creating the `SignedInfo`, we pass as parameters:

- The `CanonicalizationMethod` (we use inclusive and preserve comments)
- The `SignatureMethod` (we use DSA)
- A list of `References` (in this case, only one)

```
SignedInfo si = fac.newSignedInfo
  (fac.newCanonicalizationMethod
    (CanonicalizationMethod.INCLUSIVE_WITH_COMMENTS,
      (C14NMethodParameterSpec) null),
    fac.newSignatureMethod(SignatureMethod.DSA_SHA1, null),
    Collections.singletonList(ref));
```

Next, we create the optional `KeyInfo` object, which contains information that enables the recipient to find the key needed to validate the signature. In this example, we add a `KeyValue` object containing the public key. To create `KeyInfo` and its various subtypes, we use a `KeyInfoFactory` object, which can be obtained by invoking the `getKeyInfoFactory` method of the `XMLSignature-Factory`, as follows:

```
KeyInfoFactory kif = fac.getKeyInfoFactory();
```

We then use the `KeyInfoFactory` to create the `KeyValue` object and add it to a `KeyInfo` object:

```
KeyValue kv = kif.newKeyValue(kp.getPublic());
KeyInfo ki = kif.newKeyInfo(Collections.singletonList(kv));
```

Finally, we create the `XMLSignature` object, passing as parameters the `Signed-Info` and `KeyInfo` objects that we created earlier:

```
XMLSignature signature = fac.newXMLSignature(si, ki);
```

Notice that we haven't actually generated the signature yet; we'll do that in the next step.

Generating the XML Signature

Now we are ready to generate the signature, which we do by invoking the `sign` method on the `XMLSignature` object, and pass it the signing context as follows:

```
signature.sign(dsc);
```

The resulting document now contains a signature, which has been inserted as the last child element of the root element.

Printing or Displaying the Resulting Document

You can use the following code to print the resulting signed document to a file or standard output:

```
OutputStream os;
if (args.length > 1) {
  os = new FileOutputStream(args[1]);
} else {
  os = System.out;
}

TransformerFactory tf = TransformerFactory.newInstance();
Transformer trans = tf.newTransformer();
trans.transform(new DOMSource(doc), new StreamResult(os));
```

8 Securing Web Services

THE security model used for web services is based on specifications and recommendations of various standards organizations (see Web Services Security Initiatives and Organizations, page 236). The challenge behind the security model for Java EE-based web services is to understand and assess the risk involved in securing a web-based service today and, at the same time, track emerging standards and understand how they will be deployed to offset the risk in the future.

This chapter addresses using message security to address the characteristics of a web service that make its security needs different from those of other Java EE applications.

This chapter assumes that you are familiar with the web services technologies being discussed, or that you have read the following chapters in this tutorial that discuss these technologies:

- Chapter 1, “Building Web Services with JAX-WS”
- Chapter 3, “Using JAXB”
- Chapter 6, “Java API for XML Registries”
- Chapter 5, “SOAP with Attachments API for Java”

Securing Web Service Endpoints

Web services can be deployed as EJB endpoints or as web (servlet) endpoints. Securing web service endpoints is discussed in the following chapters:

- For information on securing web service endpoints of an enterprise bean, read Securing Enterprise Beans (page 1024).
- For information on securing web service endpoints of web components, read Chapter 9, “Securing Web Applications”.

Overview of Message Security

Java EE security is easy to implement and configure, and can offer fine-grained access control to application functions and data. However, as is inherent to security applied at the application layer, security properties are not transferable to applications running in other environments and only protect data while it is residing in the application environment. In the context of a traditional application, this is not necessarily a problem, but when applied to a web services application, Java EE security mechanisms provide only a partial solution.

The characteristics of a web service that make its security needs different than those of other Java EE applications include the following:

- Loose coupling between the service provider and service consumer
- Standards-based (read Web Services Security Initiatives and Organizations, page 236 for a discussion of web services security initiatives and organizations)
- Uses XML-formatted messages and metadata
- Highly-focused on providing interoperability
- Platform and programming language neutral
- Can use a variety of transport protocols, although HTTP is used most often
- Supports interactions with multiple hops between the service consumer and the service provider

Some of the characteristics of a web service that make it especially vulnerable to security attacks include the following:

- Interactions are performed over the Internet using transport protocols that are firewall friendly.
- Communication is often initiated by service consumers who have no prior relationship with the service provider.
- The message format is text-based.

Additionally, the distributed nature of web service interactions and dependencies might require a standard way to propagate identity and trust between application domains.

There are several well-defined aspects of application security that, when properly addressed, help to minimize the security threat faced by an enterprise. These include authentication, authorization, integrity, confidentiality, and non-repudiation, and more. These requirements are discussed in more detail in Characteristics of Application Security (page 946).

One of the methods that can be used to address the unique challenges of web services security is *message security.* Message security is discussed in this chapter which includes the following topics:

- Advantages of Message Security (page 233)
- Message Security Mechanisms (page 235)
- Web Services Security Initiatives and Organizations (page 236)
- Using Message Security with Java EE (page 241)

Advantages of Message Security

Before we get to message security, it is important to understand why security at the transport layer is not always sufficient to address the security needs of a web service. *Transport-layer security* is provided by the transport mechanisms used to transmit information over the wire between clients and providers, thus transport-layer security relies on secure HTTP transport (HTTPS) using Secure Sockets Layer (SSL). Transport security is a point-to-point security mechanism that can be used for authentication, message integrity, and confidentiality. 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. Security is “live” from the time it leaves the consumer until it arrives at the provider, or vice versa, even across intermediaries. The problem is that it is not protected once it gets to its destination. One solution is to encrypt the message before sending using *message security.*

In message-layer security, security information is contained within the SOAP message and/or SOAP message attachment, which allows security information to travel along with the message or attachment. 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-layer security is also sometimes referred to as *end-to-end security.*

The advantages of message-layer security include the following:

- Security stays with the message over all hops and after the message arrives at its destination.
- Is fine-grained. Can be selectively applied to different portions of a message (and to attachments if using XWSS).
- Can be used in conjunction with intermediaries over multiple hops.
- Is independent of the application environment or transport protocol.

The disadvantage to using message-layer security is that it is relatively complex and adds some overhead to processing.

The Application Server and the Java Web Services Developer Pack (Java WSDP) both support message security.

- The Sun Java System Application Server uses Web Services Security (WSS) to secure messages. Using WSS is discussed in Using the Application Server Message Security Implementation (page 242).
- The Java Web Services Developer Pack (Java WSDP) includes XML and Web Services Security (XWSS), a framework for securing JAX-RPC, JAX-WS, and SAAJ applications, as well as message attachments. An implementation of XWSS is included in the Application Server. Using XWSS is discussed in Using the Java WSDP XWSS Security Implementation (page 247).

Because neither of these options for message security are part of the Java EE platform, this document would not normally discuss using either of these options to secure messages. However, as there are currently no Java EE APIs that perform this function and message security is a very important component of web services security, this chapter presents a brief introduction to using both the WSS and XWSS functionality that is incorporated into the Sun Java System Application Server.

This chapter includes the following topics:

- Message Security Mechanisms (page 235)
- Web Services Security Initiatives and Organizations (page 236)
- Using Message Security with Java EE (page 241)

Message Security Mechanisms

Encryption is the transformation of data into a form that is as close to impossible as possible to read without the appropriate knowledge, which is contained in a *key*. Its purpose is to ensure privacy by keeping information hidden from anyone for whom it is not intended, even those who have access to the encrypted data. *Decryption* is the reverse of encryption; it is the transformation of encrypted data back into an intelligible form.

Encryption and decryption generally require the use of some secret information, referred to as a key. For some encryption mechanisms, the same key is used for both encryption and decryption; for other mechanisms, the keys used for encryption and decryption are different.

Authentication is as fundamentally a part of our lives as privacy. We use authentication throughout our everyday lives - when we sign our name to some document for instance - and, as we move to a world where our decisions and agreements are communicated electronically, we need to have electronic techniques for providing authentication.

The “crypt” in encryption and decryption is *cryptography*. Cryptography provides mechanisms for providing authentication, which include encryption and decryption, as well as digital signatures and digital timestamps. A *digital signature* binds a document to the possessor of a particular key, while a *digital timestamp* binds a document to its creation at a particular time. These cryptographic mechanisms can be used to control access to a shared disk drive, a high security installation, or a pay-per-view TV channel.

Authentication is any process through which one proves and verifies certain information. Sometimes one may want to verify the origin of a document, the identity of the sender, the time and date a document was sent and/or signed, the identity of a computer or user, and so on. A digital signature is a cryptographic means through which many of these may be verified. The digital signature of a document is a piece of information based on both the document and the signer's private key. It is typically created through the use of a hash function and a private signing function (encrypting with the signer's private key), but there are other methods.

For more information on cryptography, please read this document: *RSA Laboratories' Frequently Asked Questions About Today's Cryptography, Version 4.1*, available at `http://www.rsasecurity.com/rsalabs/node.asp?id=2152`. (Some of the text in this section was excerpted, by permission, from this document.)

Web Services Security Initiatives and Organizations

The following organizations work on web services security specifications, guidelines, and tools:

- The World Wide Web Consortium (W3C)
- Organization for Advancement of Structured Information Standards (OASIS)
- Web Services Interoperability Organization (WS-I)
- Java Community Process (JCP)

Basically, the JCP, W3C, and OASIS are developing specifications related to web services security. WS-I creates profiles that recommend what to implement from various specifications and provides direction on how to implement the specifications. The following sections briefly discuss the specifications and profiles being developed by each organization.

W3C Specifications

The mission of the World Wide Web Consortium (W3C), according to its Web site at `http://www.w3.org/`, is to lead the World Wide Web to its full potential by developing protocols and guidelines that ensure long-term growth for the web. W3C primarily pursues its mission through the creation of Web standards and guidelines. The W3C is working on the following specifications related to web services security:

- XML Encryption (XML-Enc)

  This specification provides requirements for XML syntax and processing for encrypting digital content, including portions of XML documents and protocol messages. The version of the specification current at the time of this writing may be viewed at http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/.

- XML Digital Signature (XML-Sig)

  This specification specifies an XML compliant syntax used for representing the signature of web resources and portions of protocol messages (anything referenceable by a URI) and procedures for computing and verifying such signatures. The version of the specification current at the time of this writing may be viewed at http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/.

- XML Key Management Specification (XKMS)

  The specification specifies protocols for distributing and registering public keys, suitable for use in conjunction with the W3C recommendations for XML Signature and XML Encryption. The version of the specification current at the time of this writing may be viewed at http://www.w3.org/TR/2005/REC-xkms2-20050628/.

OASIS Specifications

According to its web site at http://www.oasis-open.org/, the Organization for the Advancement of Structured Information Standards (OASIS) drives the development, convergence, and adoption of e-business standards. OASIS is working on the following specifications related to web services security. At the time this document was written, OASIS standards documents are available from http://www.oasis-open.org/specs/index.php.

- Web Services Security (WSS): SOAP Message Security

  This specification describes enhancements to SOAP messaging to provide message integrity, message confidentiality, and message authentication while accommodating a wide variety of security models and encryption technologies. This specification also defines an extensible, general-purpose mechanism for associating security tokens with message content, as well as how to encode binary security tokens, a framework for XML-based tokens, and how to include opaque encrypted keys.

- Security Assertion Markup Language (SAML)

  The SAML specification defines an XML-based mechanism for securing Business-to-Business (B2B) and Business-to-Consumer (B2C) e-commerce transactions. SAML defines an XML framework for exchanging authentication and authorization information. SAML uses XML-encoded security assertions and XML-encoded request/response protocol and specifies rules for using assertions with standard transport and messaging frameworks. SAML provides interoperability between disparate security systems. SAML can be applied to facilitate three use cases: single sign-on, distributed transactions, and authorization services.

- eXtensible Access Control Markup Language (XACML)

  The XACML specification defines a common language for expressing security policy. XACML defines an extensible structure for the core schema and namespace for expressing authorization policies in XML. A common policy language, when implemented across an enterprise, allows the enterprise to manage the enforcement of all the elements of its security policy in all the components of its information systems.

JCP Specifications

According to the Java Community Process (JCP) web site, the JCP holds the responsibility for the development of Java technology. The JCP primarily guides the development and approval of Java technical specifications. The JCP is working on the following specifications related to web services security. The specifications can be viewed from the JCP web site at http://www.jcp.org/en/jsr/all.

- JSR 104: XML Trust Service APIs

  JSR-104 defines a standard set of APIs and a protocol for a trust service. A key objective of the protocol design is to minimize the complexity of applications using XML Signature. By becoming a client of the trust service, the application is relieved of the complexity and syntax of the underlying PKI used to establish trust relationships, which may be based upon a different specification such as X.509/PKIX, SPKI or PGP.

- JSR 105: XML Digital Signature APIs

  JSR-105 defines a standard set of APIs for XML digital signature services. The XML Digital Signature specification is defined by the W3C.

This proposal is to define and incorporate the high-level implementation-independent Java APIs.

- JSR 106: XML Encryption APIs

  JSR-106 defines a standard set of APIs for XML digital encryption services. XML Encryption can be used to perform fine-grained, element-based encryption of fragments within an XML Document as well as encrypt arbitrary binary data and include this within an XML document.

- JSR 155: Web Services Security Assertions

  JSR-155 provides a set of APIs, exchange patterns, and implementation to securely (integrity and confidentiality) exchange assertions between web services based on OASIS SAML.

- JSR 183: Web Services Message Security APIs

  JSR-183 defines a standard set of APIs for Web services message security. The goal of this JSR is to enable applications to construct secure SOAP message exchanges.

- JSR 196: Java Authentication Service Provider Interface for Containers

  The proposed specification will define a standard service provider interface by which authentication mechanism providers may be integrated with containers. Providers integrated through this interface will be used to establish the authentication identities used in container access decisions, including those used by the container in invocations of components in other containers.

WS-I Specifications

According to the Web Services Interoperability Organization (WS-I) web site, WS-I is an open industry organization chartered to promote Web services interoperability across platforms, operating systems and programming languages. Specifically, WS-I creates, promotes and supports generic protocols for the interoperable exchange of messages between Web services. WS-I creates profiles, which recommend what to use and how to use it from the various web services specifications created by W3C, OASIS, and the JCP. WS-I is working on the following profiles related to web services security. The profiles can be viewed from the WS-I web site at `http://www.ws-i.org/deliverables/Default.aspx`.

- Basic Security Profile (BSP)

The Basic Security Profile provides guidance on the use of WS-Security and the User Name and X.509 security token formats.

- REL Token Profile

  The REL Token Profile is the interoperability profile for the Rights Expression Language (REL) security token that is used with WS-Security.

- SAML Token Profile

  This is the interoperability profile for the Security Assertion Markup Language (SAML) security token that is used with WS-Security.

- Security Challenges, Threats, and Countermeasures

  This document identifies potential security challenges and threats in a web service application, and identifies appropriate candidate technologies to address these challenges. The section Security Challenges, Threats, and Countermeasures (page 240) discusses the challenges, threats, and countermeasures in a bit more detail.

Security Challenges, Threats, and Countermeasures

The WS-I document titled *Security Challenges, Threats, and Countermeasures* can be read in its entirety at `http://www.ws-i.org/Profiles/BasicSecurity/SecurityChallenges-1.0.pdf`. Table 8–1 attempts to summarize many of the threats and countermeasures as an introduction to this document.

Table 8–1 Security Challenges, Threats, and Countermeasures

Challenge	Threats	Countermeasures
Peer Identification and Authentication	falsified messages, man in the middle, principal spoofing, forged claims, replay of message parts	-HTTPS with X.509 server authentication -HTTP client authentication (Basic or Digest) -HTTPS with X.509 mutual authentication of server and user agent -OASIS SOAP Message Security

Table 8–1 Security Challenges, Threats, and Countermeasures (Continued)

Challenge	**Threats**	**Countermeasures**
Data Origin Identification and Authentication	falsified messages, man in the middle, principal spoofing, forged claims, replay of message parts	-OASIS SOAP Message Security -MIME with XML Signature/XML Encryption -XML Signature
Data Integrity (including Transport Data Integrity and SOAP Message Integrity)	message alteration, replay	-SSL/TLS with encryption enabled -XML Signatures (as profiled in OASIS SOAP Message Security)
Data Confidentiality (including Transport Data Confidentiality and SOAP Message Confidentiality)	confidentiality	-SSL/TSL with encryption enabled -XML Signatures (as profiled in OASIS SOAP Message Security)
Message Uniqueness	replay of message parts, replay, denial of service	-SSL/TLS between the node that generated the request and the node that is guaranteeing -Signing of nonce, time stamp

As you can see from the countermeasures that are recommended in the table and in the document, the use of XML Encryption and XML Digital Signature to secure SOAP messages and attachments is strongly recommended by this organization. Using Message Security with Java EE (page 241) discusses some options for securing messages with Java EE.

Using Message Security with Java EE

Because message security is not yet a part of the Java EE platform, and because message security is a very important component of web services security, this section presents a brief introduction to using both the Application Server's Web Services Security (WSS) and the Java WSDP's XML and Web Services Security (XWSS) functionality.

- Using the Application Server Message Security Implementation (page 242)
- Using the Java WSDP XWSS Security Implementation (page 247)

Using the Application Server Message Security Implementation

The Sun Java System Application Server uses Web Services Security (WS-Security) to secure messages. WS-Security is a message security mechanism that uses XML Encryption and XML Digital Signature to secure web services messages sent over SOAP. The WS-Security specification defines the use of various security tokens including X.509 certificates, SAML assertions, and username/password tokens to authenticate and encrypt SOAP web services messages.

The Application Server offers integrated support for the WS-Security standard in its web services client and server-side containers. This functionality is integrated such that web services security is enforced by the containers of the Application Server on behalf of applications, and such that it can be applied to protect any web service application without requiring changes to the implementation of the application. The Application Server achieves this effect by providing facilities to bind SOAP layer *message security providers* and message protection policies to containers and to applications deployed in containers.

There are two ways to enable message security when using the Application Server:

- Configure the Application Server so that web services security will be applied to all web services applications deployed on the Application Server. For more information, read How Does WSS Work in the Application Server (page 242).
- Configure application-specific web services security by annotating the server-specific deployment descriptor. For more information, read Configuring Application-Specific Message Security (page 244).

How Does WSS Work in the Application Server

Web services deployed on the Application Server are secured by binding SOAP layer message security providers and message protection policies to the containers in which the applications are deployed or to web service endpoints served by the applications. SOAP layer message security functionality is configured in the client-side containers of the Application Server by binding SOAP layer message security providers and message protection policies to the client containers or to the portable service references declared by client applications.

When the Application Server is installed, SOAP layer message security providers are configured in the client and server-side containers of the Application

Server, where they are available for binding for use by the containers, or by individual applications or clients deployed in the containers. During installation, the providers are configured with a simple message protection policy that, if bound to a container, or to an application or client in a container, would cause the source of the content in all request and response messages to be authenticated by XML digital signature.

By default, message layer security is disabled on the Application Server. To configure message layer security at the Application Server level, read Configuring the Application Server for Message Security (page 243). To configure message security at the application level, read Configuring Application-Specific Message Security (page 244).

Configuring the Application Server for Message Security

The following steps briefly explain how to configure the Application Server for message security. For more detailed information on configuring the Application Server for message security, refer to the Application Server's *Administration Guide*. For a link to this document, see Further Information (page 251).

To configure the SOAP layer *message security providers* in the client and server-side containers of the Application Server, follow these steps:

1. Start the Application Server as described in Starting and Stopping the Application Server (page 28).
2. Start the Admin Console, as described in Starting the Admin Console (page 29).
3. In the Admin Console tree component, expand the Configuration node.
4. Expand the Security node.
5. Expand the Message Security node.
6. Select the SOAP node.
7. Select the Message Security tab.
8. On the Edit Message Security Configuration page, specify a provider to be used on the server side and/or a provider to be used on the client side for all applications for which a specific provider has not been bound. For more description of each of the fields on this page, select Help from the Admin Console.
9. Select Save.

10. To modify the message protection policies of the enabled providers, select the Providers tab.
11. Select a provider for which to modify message protection policies. For more description on each of the fields on the Edit Provider Configuration page, select Help from the Admin Console.
12. Click Save and restart the Application Server if so indicated.

Configuring Application-Specific Message Security

Application-specific web services message security functionality is configured (at application assembly) by adding `message-security-binding` elements to the web service endpoint. The `message-security-binding` elements are added to the runtime deployment descriptors of the application (`sun-ejb-jar.xml`, `sun-web.xml`, or `sun-application-client.xml`). These `message-security-binding` elements are used to associate a specific provider or message protection policy with a web services endpoint or service reference, and may be qualified so that they apply to a specific port or method of the corresponding endpoint or referenced service.

The following is an example of a `sun-ejb-jar.xml` deployment descriptor file to which a `message-security-binding` element has been added:

```
<sun-ejb-jar>
  <enterprise-beans>
    <unique-id>1</unique-id>
  <ejb>
    <ejb-name>HelloWorld</ejb-name>
    <jndi-name>HelloWorld</jndi-name>
    <webservice-endpoint>
      <port-component-name>HelloIF</port-component-name>
      <endpoint-address-uri>service/HelloWorld</endpoint-
address-uri>
      <message-security-binding auth-layer="SOAP">
        <message-security>
          <message>
            <java-method>
              <method-name>ejbTaxCalc</method-name>
            </java-method>
          </message>
          <message>
            <java-method>
              <method-name>sayHello</method-name>
            </java-method>
          </message>
          <request-protection auth-source="content" />
          <response-protection auth-source="content"/>
        </message-security>
      </message-security-binding>
    </webservice-endpoint>
    </ejb>
  </enterprise-beans>
</sun-ejb-jar>
```

In this example, the `message-security-binding` element has been added to a web service endpoint for an enterprise bean. The elements highlighted in **bold** above are described briefly below and in more detail in the Application Server's *Application Deployment Guide*. A link to this document is provided in Further Information (page 251).

- `message-security-binding`: This element specifies a custom authentication provider binding for a parent `webservice-endpoint` or `port-info` element by binding to a specific provider and/or by specifying the message security requirements enforced by the provider. It contains the attributes `auth-layer` and `provider-id` (optional).
  - `auth-layer`: This element specifies the message layer at which authentication is performed. The value must be `SOAP`.
  - `provider-id`: This element is optional and specifies the authentication provider used to satisfy application-specific message security requirements. If this attribute is not specified, a default provider is used, if there is one defined for the message layer. If no default provider is defined, authentication requirements defined in the `message-security-binding` element are not enforced.
- `message-security`: This element specifies message security requirements. If the grandparent element is `webservice-endpoint`, these requirements pertain to request and response messages of the endpoint. If the grandparent element is `port-info`, these requirements pertain to the port of the referenced service.
  - `message`: This element includes the methods (`java-method`) and operations (`method-name`) to which message security requirements apply. If this element is not included, message protection applies to all methods.
  - `request-protection`: This element defines the authentication policy requirements of the application's request processing. It has attributes of `auth-source` and `auth-recipient` to define what type of protection is applied and when it is applied.

- response-protection: This element defines the authentication policy requirements of the application's response processing. It has attributes of auth-source and auth-recipient to define what type of protection is applied and when it is applied.
- auth-source: This attribute specifies the type of required authentication, either sender (user name and password) or content (digital signature). This is an attribute of the request-protection and response-protection elements.
- auth-recipient: This attribute specifies whether recipient authentication occurs before or after content authentication. Allowed values are before-content and after-content. This is an attribute of the request-protection and response-protection elements.

For more detailed information on configuring application-specific web services security, refer to the Application Server's *Developer's Guide*. For more detailed information on the elements used for message security binding, read the Application Server's *Application Deployment Guide*. For a link to these documents, see Further Information (page 251).

Example: Using Application Server WS-Security

The Application Server ships with sample applications named xms and xms_apl_lvl. Both applications features a simple web service that is implemented by both a Java EE EJB endpoint and a Java Servlet endpoint. Both endpoints share the same service endpoint interface. The service endpoint interface defines a single operation, sayHello, which takes a String argument, and returns a String composed by pre-pending Hello to the invocation argument.

- The xms application shows how to enable message layer security at the Application Server level by enabling the Application Server's default message security providers. In this case, web services are protected using default configuration files and default WSS providers.
- The xms_apl_lvl application shows how to enable message layer security at the application level by modifying the runtime deployment descriptor (sun-ejb-jar.xml or sun-web.xml). In this case, you can selectively specify when/how message layer security can be applied to a specific method (or for all methods) in a web service.

The instructions which accompany the sample describe how to enable the WS-Security functionality of the Application Server such that it is used to secure the `xms` application. The sample also demonstrates the binding of WS-Security functionality directly to the application. (The `/samples/` directory will only exist if you selected `Install Samples Server` during installation.)

The sample applications are installed in the following directories:

- `<INSTALL>/samples/webservices/security/ejb/apps/xms/`
- `<INSTALL>/samples/webservices/security/ejb/apps/xms_apl_lvl`

For information on compiling, packaging, and running the sample applications, refer to the sample file at `<INSTALL>/samples/webservices/security/docs/common.html` or to the *Securing Applications* chapter of the Application Server *Developers' Guide* (see Further Information, page 251, for a link to this document).

Using the Java WSDP XWSS Security Implementation

The Java Web Services Developer Pack (Java WSDP) includes XML and Web Services Security (XWSS), a framework for securing JAX-RPC, JAX-WS, and SAAJ applications and message attachments.

XWS-Security includes the following features:

- Support for securing JAX-RPC and JAX-WS applications at the service, port, and operation levels.
- XWS-Security APIs for securing both JAX-RPC and JAX-WS applications and stand-alone applications that make use of SAAJ APIs only for their SOAP messaging.
- A sample security framework within which a JAX-RPC application developer will be able to secure applications by signing, verifying, encrypting, and/or decrypting parts of SOAP messages and attachments.

  The message sender can also make claims about the security properties by associating security tokens with the message. An example of a security claim is the identity of the sender, identified by a user name and password.

- Support for SAML Tokens and the WSS SAML Token Profile (partial).
- Support for securing attachments based on the WSS SwA Profile Draft.
- Partial support for sending and receiving WS-I Basic Security Profile (BSP) 1.0 compliant messages.
- Sample programs that demonstrate using the framework.
- Command-line tools that provide specialized utilities for keystore management, including `pkcs12import` and `keyexport`.

XWSS supports deployment onto any of the following containers:

- Sun Java System Application Server
- Sun Java System Web Server
- Apache Tomcat servlet container

Samples for using XWS-Security are included with Java WSDP in the directory `<`*`JWSDP_HOME`*`>/xws-security/samples/` or can be viewed online at `http://java.sun.com/webservices/docs/2.0/xws-security/samples.html`.

Configuring Message Security Using XWSS

The Application Server contains all of the JAR files necessary to use XWS-Security for securing JAX-WS applications, however, in order to view the sample applications, you must download and install the standalone Java WSDP bundle. You can download the Java WSDP from `http://java.sun.com/webservices/downloads/webservicespack.html`.

To add message security to an existing JAX-WS application using XWSS, follow these steps on the *client* side:

1. Create a client security configuration. The client security configuration file specifies the order and type of message security operations that will be used for the client application. For example, a simple security configuration to perform a digital signature operation looks like this:

```
<?xml version="1.0" encoding="UTF-8"?><xwss:JAXRPCSecurity
xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
<xwss:Service conformance= "bsp">
  <xwss:SecurityConfiguration dumpMessages="true" >

    <xwss:Sign id="s" includeTimestamp="true">
```

```
          <xwss:X509Token encodingType="http://docs.oasis-
open.org/wss/2004/01/
             oasis-200401-wss-soap-message-security-
1.0#Base64Binary"
             valueType="http://docs.oasis-open.org/wss/2004/
01/oasis-200401-wss-
             x509-token-profile-1.0#X509SubjectKeyIdentifier"
             certificateAlias="xws-security-client"
keyReferenceType="Identifier"/>
        </xwss:Sign>

      </xwss:SecurityConfiguration>
   </xwss:Service>

   <xwss:SecurityEnvironmentHandler>
      simple.client.SecurityEnvironmentHandler
   </xwss:SecurityEnvironmentHandler>
   </xwss:JAXRPCSecurity>
```

For more information on writing and understanding security configurations and setting up `SecurityEnvironmentHandlers`, please see the *Java Web Services Developer Pack Tutorial* at `http://java.sun.com/webservices/docs/1.6/tutorial/doc/index.html`.

2. In your client code, create an `XWSSecurityConfiguration` object initialized with the security configuration generated. Here is an example of the code that you would use in your client file. For an example of a complete file that uses this code, look at the example client in the `\jaxws2.0\simple-doclit\src\simple\client\` directory.

```
     FileInputStream f = new FileInputStream("./etc/
client_security_config.xml");
     XWSSecurityConfiguration config =

SecurityConfigurationFactory.newXWSSecurityConfiguration(f);
```

3. Set security configuration information on the `RequestContext` by using the `XWSSecurityConfiguration.MESSAGE_SECURITY_CONFIGURATION` property. For an example of a complete file that uses this code, look at the

example client in the \jaxws2.0\simple-doclit\src\simple\client\ directory.

```
    // put the security config info
    ((BindingProvider)stub).getRequestContext().

put(XWSSecurityConfiguration.MESSAGE_SECURITY_CONFIGURATION,
        config);
```

4. Invoke the method on the stub as you would if you were writing the client without regard to adding XWS-Security. The example for the application from the \jaxws2.0\simple-doclit\src\simple\client\ directory is as shown below:

```
    Holder<String> hold = new Holder("Hello !");
    stub.ping(ticket, hold);
```

To add message security to an existing JAX-RPC, JAX-WS, or SAAJ application using XWSS, follow these steps on the *server* side:

1. Create a server security configuration file and give it the name: `serviceName + "_" + "security_config.xml`

   An example of a server security configuration file can be found in the \jaxws2.0\simple-doclit\etc\server_security_config.xml directory.

2. No other changes need to be made to the server-side JAX-WS code.

Information about running the example application is included in <*JWSDP_HOME*>/xws-security/samples/jaxws2.0/simple-doclit/README.txt and in the *Java WSDP Tutorial*.

For more information on XWSS,

- Read the *Java Web Services Developer Pack Tutorial*. The tutorial can be accessed from http://java.sun.com/webservices/docs.html.
- Read the XWSS samples documentation, which is located in the <*JWSDP_HOME*>/xws-security/samples/ directory of your Java WSDP installation or at http://java.sun.com/webservices/docs/2.0/xws-security/samples.html online.
- Visit the XWSS home page at http://java.sun.com/webservices/xwss/.

- Take the Sun training class titled *Developing Secure Java Web Services*. To sign up, go to `https://www.sun.com/training/catalog/java/web_services.html`.

Further Information

- Java 2 Standard Edition, v.1.5.0 Security: http://java.sun.com/j2se/1.5.0/docs/guide/security/index.html
- *Java EE 5 Specification* at `http://java.sun.com/j2ee/download.html#platformspec`.
- *Java Web Services Developer Pack Tutorial* at `http://java.sun.com/webservices/docs/1.6/tutorial/doc/index.html`
- The *Developer's Guide* for the Application Server contains information on developing applications specifically for deployment onto the Application Server. As of this writing, this document is available for viewing at `http://docs.sun.com/app/docs/doc/819-3659`.
- The *Administration Guide* for the Application Server includes information on setting security settings for the Application Server. As of this writing, this document was available for viewing at `http://docs.sun.com/app/docs/doc/819-3658`.
- The *Application Deployment Guide* for the Application Server is available, as of this writing, at:

  `http://docs.sun.com/app/docs/doc/819-3660`
- *Web Services for Java EE* (JSR-109), at `http://jcp.org/aboutJava/communityprocess/maintenance/jsr109/index.html`.
- OASIS Standard 200401: Web Services Security: *SOAP Message Security 1.0*

`http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf`

- *XML Encryption Syntax and Processing*
  `http://www.w3.org/TR/xmlenc-core/`
- Digital Signatures Working Draft
  `http://www.w3.org/Signature/`
- JSR 105-*XML Digital Signature APIs*
  `http://www.jcp.org/en/jsr/detail?id=105`
- JSR 106-*XML Digital Encryption APIs*
  `http://www.jcp.org/en/jsr/detail?id=106`
- *Public-Key Cryptography Standards* (PKCS)
  `http://www.rsasecurity.com/rsalabs/pkcs/index.html`
- Java Authentication and Authorization Service (JAAS)
  `http://java.sun.com/products/jaas/`
- *WS-I Basic Security Profile* Version 1.0
  `http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0-2005-01-20.html`
- Web Services Security: *SOAP Messages with Attachments (SwA) Profile* 1.0
  `http://www.oasis-open.org/committees/download.php/10090/wss-swa-profile-1.0-draft-14.pdf`
- Web Services Security: *SOAP Messages with Attachments (SwA) Profile* 1.0, Interop 1 Scenarios
  `http://lists.oasis-open.org/archives/wss/200410/pdf00003.pdf`
- Web Services Security: *Security Assertion Markup Language (SAML) Token Profile 1.0*
  `http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.0.pdf`
- Web Services Security: *Security Assertion Markup Language (SAML) Interop Scenarios*
  `http://www.oasis-open.org/apps/org/workgroup/wss/download.php/7011/wss-saml-interop1-draft-11.doc`

Index

A

addChildElement method 128
addClassifications method 188
addExternalLink method 195
addServiceBindings method 189
addServices method 190
addTextNode method 128
AttachmentPart class 121, 137
    creating objects 137
    headers 137
attachments 120
    adding 137
    SAAJ example 166
attributes
    SOAP envelope 129
    SOAP header 141
    XML elements 139
authentication
    for XML registries 186

B

binding templates
    adding to an organization with JAXR 189
    finding with JAXR 185
businesses
    contacts 187
    creating with JAXR 186
    finding
        by name with JAXR 182, 202
        using WSDL documents with JAXR 205
    finding by classification with JAXR 183, 202
    keys 187, 193
    publishing with JAXR 190
    removing with JAXR 193, 202
    saving with JAXR 202–204
BusinessLifeCycleManager interface 173, 181, 186
BusinessQueryManager interface 173, 181

C

call method 122–123, 132
capability levels, JAXR 172
classification schemes
    finding with JAXR 188
    ISO 3166 181
    NAICS 181, 202
    postal address 195, 203
    publishing with JAXR 195, 203
    removing with JAXR 204
    UNSPSC 181

user-defined 194
classifications
creating with JAXR 188
clients, JAXR 173
examples 199
implementing 174
`close` method 132
`com.sun.xml.registry.http.proxyHost` connection property 179
`com.sun.xml.registry.http.proxyPort` connection property 179
`com.sun.xml.registry.https.proxyHost` connection property 180
`com.sun.xml.registry.https.proxyPassword` connection property 180
`com.sun.xml.registry.https.proxyPort` connection property 180
`com.sun.xml.registry.https.proxyUserName` connection property 180
`com.sun.xml.registry.useCache` connection property 180
`com.sun.xml.registry.userTaxonomyFilenames` connection property 180, 197
concepts
in user-defined classification schemes 194
publishing with JAXR 190, 204
removing with JAXR 205
using to create classifications with JAXR 188
connection factories, JAXR
creating 177
`Connection` interface (JAXR) 173, 177
connection properties, JAXR 178
examples 177
`ConnectionFactory` class (JAXR) 177
connections, JAXR
creating 177
setting properties 177
connections, SAAJ 122
closing 132
point-to-point 131
country codes
ISO 3166 181
`createClassification` method 188, 195
`createClassificationScheme` method 195
`createExternalLink` method 195
`createOrganization` method 187
`createPostalAddress` method 198
`createService` method 189
`createServiceBinding` method 189

D

`deleteOrganizations` method 193
`detachNode` method 126
`Detail` interface 148
`DetailEntry` interface 148
DOM
SAAJ and 122, 136, 162

E
ebXML
    registries 172–173
encrypting
    SOAP messages 247
end-to-end security 234
examples
    JAXR
        Java EE application 206
        simple 199
    required software xi
    SAAJ
        attachments 166
        DOM 162
        headers 160
        request-response 153
        SOAP faults 168
    setting build properties xi
    web services xvi

F
`findClassificationSchemeByName` method 188
`findConcepts` method 184
`findOrganization` method 182
framework
    XWS-Security 247
fully qualified names 127

G
`getAttachments` method 139
`getBody` method 126
`getEnvelope` method 126
`getHeader` method 126
`getRegistryObject` method 183
`getSOAPBody` method 126
`getSOAPHeader` method 126
`getSOAPPart` method 126
`getValue` method 132

H
HTTP xv–xvi
    setting proxies 179

I
information model, JAXR 172–173
ISO 3166 country codes 181

J
Java EE applications
    JAXR example 206
`javax.activation.DataHandler` class 137–138
`javax.xml.namespace.QName` class 127
`javax.xml.registry` package 173
`javax.xml.registry.infomodel` package 173
`javax.xml.registry.lifeCycleManagerURL` connection property 178
`javax.xml.registry.postalAddressScheme` connection property 179, 197
`javax.xml.registry.queryManagerURL` connection property 178
`javax.xml.registry.security.authenticationMethod` connection property 179
`javax.xml.registry.seman-`

ticEquivalences connection property 179, 197
javax.xml.registry.uddi.maxRows connection property 179
javax.xml.soap package 117
javax.xml.transform.Source interface 135
JAXM specification 118
JAXR 171
  adding
    classifications 188
    service bindings 189
    services 189
  architecture 173
  capability levels 172
  clients 173–174
  creating connections 177
  defining taxonomies 194
  definition 172
  establishing security credentials 186
  finding classification schemes 188
  information model 172
  Java EE application example 206
  organizations
    creating 186
    publishing 190
    removing 193
  overview 171
  provider 173
  publishing
    specification concepts 190
    WSDL documents 190
  querying a registry 181
  specification 172
  specifying postal addresses 197
  submitting data to a registry 186
JAX-RPC
  securing applications 247
  service endpoint interfaces xvii
JAX-RPC applications
  securing 247
JAX-WS
  defined xv
  specification xxiv

K
keyexport command 248

L
local names 129–130

M
MessageFactory class 124
messages, SAAJ
  accessing elements 126
  adding body content 127
  attachments 120
  creating 124
  getting the content 132
  overview 118
MIME
  headers 122

N
NAICS 181

using to find organizations 183, 202
`Name` interface 127
names
fully qualified 127, 130
local 129–130
namespaces 127
prefix 129
nodes
SAAJ and 118

O
`Organization` interface 187
organizations
creating with JAXR 186
finding
by classification 183, 202
by name 182, 202
using WSDL documents 205
keys 187, 193
primary contacts 187
publishing with JAXR 190, 202–204
removing with JAXR 193, 202

P
`pkcs12import` command 248
point-to-point connection, SAAJ 131
postal addresses
retrieving with JAXR 199, 203
specifying with JAXR 197, 203
prerequisites ix
printing the tutorial xiii
providers
JAXR 173
proxies xv
HTTP, setting 179

R
registries
definition 171
ebXML 172–173
querying with JAXR 181
submitting data with JAXR 186
UDDI 172
registry objects 173
retrieving with JAXR 205
`RegistryObject` interface 173
`RegistryService` interface 173, 180
request-response messaging 122
resource adapter, JAXR 175
creating resources 208
resource adapters
JAXR 208
resources
JAXR 208

S
SAAJ 117
examples 151
messages 118
overview 118
specification 117
tutorial 123
sample applications
XWS-Security
simple 248

sample programs
  XWS-Security 248
saveConcepts method 190
saveOrganizations method 190
SAX 77
security
  credentials for XML registries 186
  end-to-end 234
security tokens 247
service bindings
  adding to an organization with JAXR 189
  finding with JAXR 185
services
  adding to an organization with JAXR 189
  finding with JAXR 185
setContent method 135, 137
setPostalAddresses method 198
signing
  SOAP messages 247
SOAP xv–xvi, xxiv, 117
  body 129
    adding content 127
    Content-Type header 137
  envelope 129
  headers
    adding content 133
    Content-Id 137
    Content-Location 137
    Content-Type 137
    example 160
SOAP faults 145
  detail 147
  fault actor 147
  fault code 146
  fault string 147
  retrieving information 149
  SAAJ example 168
SOAP messages
  encrypting 247
  signing 247
  verifying 247
SOAPBody interface 119, 129
SOAPBodyElement interface 127, 129, 158
SOAPConnection class 122
  getting objects 131
SOAPElement interface 128, 158
SOAPEnvelope interface 119, 127, 129
SOAPFactory class 127
SOAPFault interface 146
  creating and populating objects 147
  detail element 147
  fault actor element 147
  fault code element 146
  fault string element 147
SOAPHeader interface 119, 133
SOAPHeaderElement interface 127, 133
SOAPMessage class 119, 125–126
SOAPPart class 119, 122, 128
  adding content 134
specification concepts
  publishing with JAXR 190, 204
  removing with JAXR 205

T
taxonomies
  finding with JAXR 188
  ISO 3166 181

NAICS 181, 202
UNSPSC 181
user-defined 194
using to find organizations 183
tokens
security 247
typographical conventions xiii

U
UDDI
accessing registries with SAAJ 153
registries 172
Universal Standard Products and Services Classification (UNSPSC) 181
UNSPSC 181

V
verifying
SOAP messages 247

W
W3C xvi, xxiv
web services
examples xvi
WSDL xvi, xxiv
publishing concepts for with JAXR 204
publishing with JAXR 190
removing concepts for with JAXR 205
using to find organizations 183, 205
`wsgen` tool xvii

X
XML xv
documents, and SAAJ 118
elements in SOAP messages 118
registries
establishing security credentials 186
XWS-Security
framework 247
sample programs 248