The Java™ Web Services Tutorial

June 21, 2004

Contents

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, not Custom) of the Java WSDP with the Sun Java System Application Server Platform Edition 8 (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 8.

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* is an adjunct to the *J2EE Tutorial*. To use it, you must first:

1. Download and install the Sun Java System Application Server Platform Edition 8 Update 1 release (hereafter called the Application Server), which you will use as your Web container. You can download this software from `http://java.sun.com/webservices/containers/`.
2. Download and install the Java WSDP 1.4 software. The Java WSDP installer will integrate the Java WSDP component technologies into the Application Server that you are using as your Web container. You can download this software from `http://java.sun.com/webservices/1.4/download.html#tutorial`.
3. If you are reading this online, you can download and install a local copy of this tutorial, which you can get from `http://java.sun.com/webservices/downloads/webservicespack.html`. All of the examples for this tutorial are installed with the Java WSDP 1.4 bundle and can be found in the subdirectories of the `<`*`JWSDP_HOME`*`>/<`*`technology`*`>/samples` directories, where *`JWSDP_HOME`* is the directory where you installed Java WSDP 1.4.
4. Download and install the *J2EE Tutorial*, which you can get from `http://java.sun.com/j2ee/1.4/download.html#tutorial`.

The *Java Web Services Tutorial* addresses the following technology areas, which are *not* covered in the J2EE Tutorial:

- The Java Architecture for XML Binding (JAXB)
- XML and Web Services Security (XWS Security)
- XML Digital Signature
- The Java WSDP Registry Server
- The Registry Browser

Java WSDP technology areas that are not covered in the *Java Web Services Tutorial* are addressed in the *J2EE Tutorial*, which opens with three introductory chapters that you should read before proceeding to any specific technology area. Java WSDP users should first look at Chapters 2 and 3, which cover XML basics and getting started with Web applications.

When you have digested the basics, you can delve into one or more of the following main XML technology areas:

- The Java XML chapters cover the technologies for developing applications that process XML documents and implement Web services components:
    - The Java API for XML Processing (JAXP)
    - The Java API for XML-based RPC (JAX-RPC)
    - SOAP with Attachments API for Java (SAAJ)
    - The Java API for XML Registries (JAXR)
- The Web-tier technology chapters cover the components used in developing the presentation layer of a J2EE or stand-alone Web application:
    - Java Servlet
    - JavaServer Pages (JSP)
    - JavaServer Pages Standard Tag Library (JSTL)
    - JavaServer Faces
    - Web application internationalization and localization
- The platform services chapters cover system services used by all J2EE component technologies. Java WSDP users should look at the Web-tier section of the Security chapter.

After you have become familiar with some of the technology areas, you are ready to tackle a case study, which ties together several of the technologies discussed in the tutorial. The Coffee Break Application (Chapter 35) describes an application that uses the Web application and Web services APIs.

Finally, the following appendixes contain auxiliary information helpful to the Web Services application developer:

- Java encoding schemes (Appendix A)
- XML Standards (Appendix B)
- HTTP overview (Appendix C)

Building the Examples

Most of the examples in the Java WSDP 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 in the J2EE Tutorial 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.

To run the `asant` scripts in the J2EE Tutorial, you must set two common build properties as follows:

- Set the `j2ee.home` property in the file `<`*`INSTALL`*`>/j2eetutorial14/examples/common/build.properties` to the location of your Application Server installation. The build process uses the `j2ee.home` property to include the libraries in `<`*`J2EE_HOME`*`>/lib/` in the classpath. All examples that run on the Application Server include the J2EE library archive—`<`*`J2EE_HOME`*`>/lib/j2ee.jar`—in the build classpath. Some examples use additional libraries in `<`*`J2EE_HOME`*`>/lib/` and `<`*`J2EE_HOME`*`>/lib/endorsed/`; the required libraries are enumerated in the individual technology chapters. <*`J2EE_HOME`*> refers to the directory where you have installed the Application Server or the J2EE 1.4 SDK.

Note: On Windows, you must escape any backslashes in the `j2ee.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 `j2ee.home` as follows:

```
j2ee.home = C:\\Sun\\AppServer
```

or

```
j2ee.home=C:/Sun/AppServer
```

- If you did not use port 8080 when you installed the Application Server, set the value of the `domain.resources.port` property in `<`*`INSTALL`*`>/j2eetutorial14/examples/common/build.properties` to the correct port.
- Set the `admin.user` and `admin.password` properties in the file `<`*`INSTALL`*`>/j2eetutorial14/examples/common/build.properties` to the values you specified when you installed the J2EE 1.4 Application Server. The build scripts use these values when you invoke an administration task such as creating a database pool. The default value for `admin.user` is set to the installer's default value, which is `admin`.

In order to run the Ant scripts, you must configure your environment and properties files as follows:

- Add the `bin` directory of your J2SE SDK installation to the front of your path.
- Add `<`*`JWSDP_HOME`*`>/jwsdp-shared/bin` to the front of your path so the Java WSDP 1.4 scripts that are shared by multiple components override other installations.
- Add `<`*`JWSDP_HOME`*`>/apache-ant/bin` to the front of your path so that the Java WSDP 1.4 Ant script overrides other installations.

Further Information

This tutorial includes the basic information that you need to deploy applications on and administer the Application Server.

For reference information on the tools distributed with the Application Server, see the man pages at `http://docs.sun.com/db/doc/817-6092`.

See the *Sun Java™ System Application Server Platform Edition 8 Developer's Guide* at `http://docs.sun.com/db/doc/817-6087` for information about developer features of the Application Server.

See the *Sun Java™ System Application Server Platform Edition 8 Administration Guide* at `http://docs.sun.com/db/doc/817-6088` for information about administering the Application Server.

For information about the PointBase database included with the Application Server, see the PointBase Web site at `www.pointbase.com`.

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
<*`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

Binding XML Schema to Java Classes with JAXB

THE Java™ Architecture for XML Binding (JAXB) provides a fast and convenient way to bind XML schemas to Java representations, making it easy for Java developers to incorporate XML data and processing functions in Java applications. As part of this process, JAXB provides methods for unmarshalling XML instance documents into Java content trees, and then marshalling Java content trees back into XML instance documents.

What this all means is that you can leverage the flexibility of platform-neutral XML data in Java applications without having to deal with or even know XML programming techniques. Moreover, you can take advantage of XML strengths without having to rely on heavyweight, complex XML processing models like SAX or DOM. JAXB hides the details and gets rid of the extraneous relationships in SAX and DOM—generated JAXB classes describe only the relationships actually defined in the source schemas. The result is highly portable XML data joined with highly portable Java code that can be used to create flexible, lightweight applications and Web services.

This chapter describes the JAXB architecture, functions, and core concepts. You should read this chapter before proceeding to Chapter 2, which provides sample code and step-by-step procedures for using JAXB.

JAXB Architecture

This section describes the components and interactions in the JAXB processing model. After providing a general overview, this section goes into more detail about core JAXB features. The topics in this section include:

- Architectural Overview
- The JAXB Binding Process
- JAXB Binding Framework
- More About javax.xml.bind
- More About Unmarshalling
- More About Marshalling
- More About Validation

Architectural Overview

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

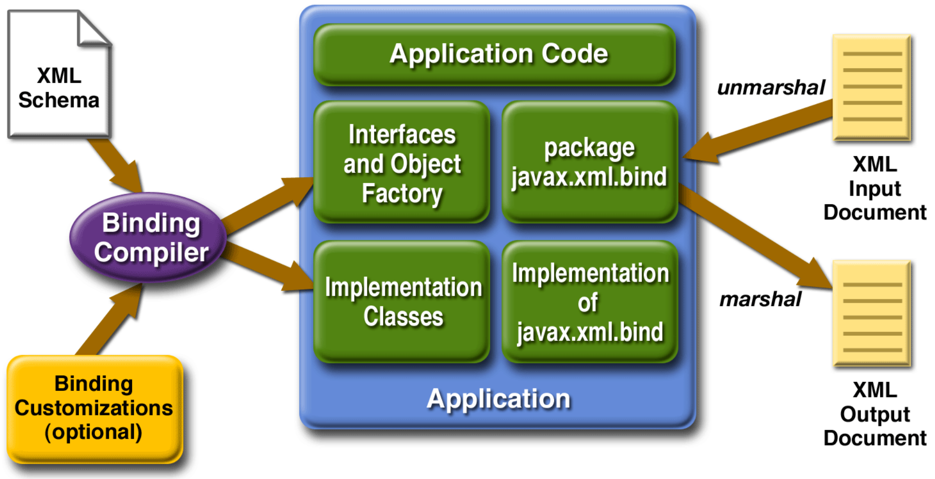

Figure 1–1 JAXB Architectural Overview

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

Table 1–1 Core Components in a JAXB Implementation

Component	**Description**
XML Schema	An XML schema uses XML syntax to describe the relationships among elements, attributes and entities in an XML document. The purpose of an XML schema is to define a class of XML documents that must adhere to a particular set of structural rules and data constraints. For example, you may want to define separate schemas for chapter-oriented books, for an online purchase order system, or for a personnel database. In the context of JAXB, an XML document containing data that is constrained by an XML schema is referred to as a *document instance*, and the structure and data within a document instance is referred to as a *content tree*.
Binding Customizations	By default, the JAXB binding compiler binds Java classes and packages to a source XML schema based on rules defined in Section 5, “Binding XML Schema to Java Representations,” in the *JAXB Specification*. In most cases, the default binding rules are sufficient to generate a robust set of schema-derived classes from a wide range of schemas. There may be times, however, when the default binding rules are not sufficient for your needs. JAXB supports customizations and overrides to the default binding rules by means of *binding customizations* made either inline as annotations in a source schema, or as statements in an external binding customization file that is passed to the JAXB binding compiler. Note that custom JAXB binding customizations also allow you to customize your generated JAXB classes beyond the XML-specific constraints in an XML schema to include Java-specific refinements such as class and package name mappings.
Binding Compiler	The JAXB binding compiler is the core of the JAXB processing model. Its function is to transform, or bind, a source XML schema to a set of JAXB *content classes* in the Java programming language. Basically, you run the JAXB binding compiler using an XML schema (optionally with custom binding declarations) as input, and the binding compiler generates Java classes that map to constraints in the source XML schema.
Implementation of `javax.xml.bind`	The JAXB binding framework implementation is a runtime API that provides interfaces for unmarshalling, marshalling, and validating XML content in a Java application. The binding framework comprises interfaces in the `javax.xml.bind` package.
Schema-Derived Classes	These are the schema-derived classes generated by the binding JAXB compiler. The specific classes will vary depending on the input schema.

Table 1–1 Core Components in a JAXB Implementation (Continued)

Component	**Description**
Java Application	In the context of JAXB, a Java application is a client application that uses the JAXB binding framework to unmarshal XML data, validate and modify Java content objects, and marshal Java content back to XML data. Typically, the JAXB binding framework is wrapped in a larger Java application that may provide UI features, XML transformation functions, data processing, or whatever else is desired.
XML Input Documents	XML content that is unmarshalled as input to the JAXB binding framework -- that is, an XML instance document, from which a Java representation in the form of a content tree is generated. In practice, the term "document" may not have the conventional meaning, as an XML instance document does not have to be a completely formed, selfstanding document file; it can instead take the form of streams of data passed between applications, or of sets of database fields, or of *XML infosets*, in which blocks of information contain just enough information to describe where they fit in the schema structure. In JAXB, the unmarshalling process supports *validation* of the XML input document against the constraints defined in the source schema. This validation process is optional, however, and there may be cases in which you know by other means that an input document is valid and so you may choose for performance reasons to skip validation during unmarshalling. In any case, validation before (by means of a third-party application) or during unmarshalling is important, because it assures that an XML document generated during marshalling will also be valid with respect to the source schema. Validation is discussed more later in this chapter.
XML Output Documents	XML content that is marshalled out to an XML document. In JAXB, marshalling involves parsing an XML content object tree and writing out an XML document that is an accurate representation of the original XML document, and is valid with respect the source schema. JAXB can marshal XML data to XML documents, SAX content handlers, and DOM nodes.

The JAXB Binding Process

Figure 1–2 shows what occurs during the JAXB binding process.

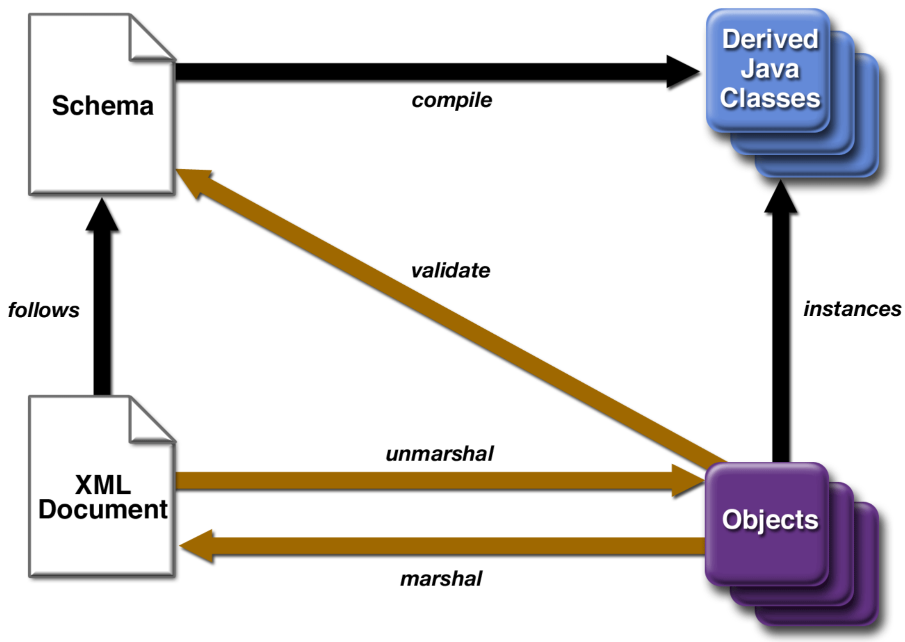

Figure 1–2 Steps in the JAXB Binding Process

The general steps in the JAXB data binding process are:

1. Generate classes. An XML schema is used as input to the JAXB binding compiler to generate JAXB classes based on that schema.
2. Compile classes. All of the generated classes, source files, and application code must be compiled.
3. Unmarshal. XML documents written according to the constraints in the source schema are unmarshalled by the JAXB binding framework. Note that JAXB also supports unmarshalling XML data from sources other than files/documents, such as DOM nodes, string buffers, SAX Sources, and so forth.
4. Generate content tree. The unmarshalling process generates a content tree of data objects instantiated from the generated JAXB classes; this content tree represents the structure and content of the source XML documents.

5. Validate (optional). The unmarshalling process optionally involves validation of the source XML documents before generating the content tree. Note that if you modify the content tree in Step 6, below, you can also use the JAXB Validate operation to validate the changes before marshalling the content back to an XML document.
6. Process content. The client application can modify the XML data represented by the Java content tree by means of interfaces generated by the binding compiler.
7. Marshal. The processed content tree is marshalled out to one or more XML output documents. The content may be validated before marshalling.

To summarize, using JAXB involves two discrete sets of activities:

- Generate and compile JAXB classes from a source schema, and build an application that implements these classes
- Run the application to unmarshal, process, validate, and marshal XML content through the JAXB binding framework

These two steps are usually performed at separate times in two distinct phases. Typically, for example, there is an application development phase in which JAXB classes are generated and compiled, and a binding implementation is built, followed by a deployment phase in which the generated JAXB classes are used to process XML content in an ongoing “live” production setting.

Note: Unmarshalling is not the only means by which a content tree may be created. Schema-derived content classes also support the programmatic construction of content trees by direct invocation of the appropriate factory methods. Once created, a content tree may be revalidated, either in whole or in part, at any time. See Create Marshal Example (page 49) for an example of using the `ObjectFactory` class to directly add content to a content tree.

JAXB Binding Framework

The JAXB binding framework is implemented in three Java packages:

- The `javax.xml.bind` package defines abstract classes and interfaces that are used directly with content classes.

  The `javax.xml.bind` package defines the `Unmarshaller`, `Validator`, and `Marshaller` classes, which are auxiliary objects for providing their respective operations.

The `JAXBContext` class is the entry point for a Java application into the JAXB framework. A `JAXBContext` instance manages the binding relationship between XML element names to Java content interfaces for a JAXB implementation to be used by the unmarshal, marshal and validation operations.

The `javax.xml.bind` package also defines a rich hierarchy of validation event and exception classes for use when marshalling or unmarshalling errors occur, when constraints are violated, and when other types of errors are detected.

- The `javax.xml.bind.util` package contains utility classes that may be used by client applications to manage marshalling, unmarshalling, and validation events.
- The `javax.xml.bind.helper` package provides partial default implementations for some of the javax.xml.bind interfaces. Implementations of JAXB can extend these classes and implement the abstract methods. These APIs are not intended to be directly used by applications using JAXB architecture.

The main package in the JAXB binding framework, `javax.bind.xml`, is described in more detail below.

More About javax.xml.bind

The three core functions provided by the primary binding framework package, `javax.xml.bind`, are marshalling, unmarshalling, and validation. The main client entry point into the binding framework is the `JAXBContext` class.

`JAXBContext` provides an abstraction for managing the XML/Java binding information necessary to implement the unmarshal, marshal and validate operations. A client application obtains new instances of this class by means of the `newInstance(contextPath)` method; for example:

```
JAXBContext jc = JAXBContext.newInstance(
"com.acme.foo:com.acme.bar" );
```

The `contextPath` parameter contains a list of Java package names that contain schema-derived interfaces—specifically the interfaces generated by the JAXB binding compiler. The value of this parameter initializes the `JAXBContext` object to enable management of the schema-derived interfaces. To this end, the JAXB provider implementation must supply an implementation class containing a method with the following signature:

```
public static JAXBContext createContext( String contextPath,
ClassLoader classLoader )

      throws JAXBException;
```

Note: The JAXB provider implementation must generate a `jaxb.properties` file in each package containing schema-derived classes. This property file must contain a property named `javax.xml.bind.context.factory` whose value is the name of the class that implements the `createContext` API.

The class supplied by the provider does not have to be assignable to `javax.xml.bind.JAXBContext`, it simply has to provide a class that implements the `createContext` API. By allowing for multiple Java packages to be specified, the `JAXBContext` instance allows for the management of multiple schemas at one time.

More About Unmarshalling

The `Unmarshaller` class in the `javax.xml.bind` package provides the client application the ability to convert XML data into a tree of Java content objects. The `unmarshal` method for a schema (within a namespace) allows for any global XML element declared in the schema to be unmarshalled as the root of an instance document. The `JAXBContext` object allows the merging of global elements across a set of schemas (listed in the `contextPath`). Since each schema in the schema set can belong to distinct namespaces, the unification of schemas to an unmarshalling context should be namespace-independent. This means that a client application is able to unmarshal XML documents that are instances of any of the schemas listed in the `contextPath`; for example:

```
JAXBContext jc = JAXBContext.newInstance(
  "com.acme.foo:com.acme.bar" );

Unmarshaller u = jc.createUnmarshaller();

FooObject fooObj =
  (FooObject)u.unmarshal( new File( "foo.xml" ) ); // ok

BarObject barObj =
  (BarObject)u.unmarshal( new File( "bar.xml" ) ); // ok
```

```
BazObject bazObj =
  (BazObject)u.unmarshal( new File( "baz.xml" ) );
  // error, "com.acme.baz" not in contextPath
```

A client application may also generate Java content trees explicitly rather than unmarshalling existing XML data. To do so, the application needs to have access and knowledge about each of the schema-derived `ObjectFactory` classes that exist in each of Java packages contained in the `contextPath`. For each schema-derived Java class, there will be a static factory method that produces objects of that type. For example, assume that after compiling a schema, you have a package `com.acme.foo` that contains a schema-derived interface named `PurchaseOrder`. To create objects of that type, the client application would use the following factory method:

```
ObjectFactory objFactory = new ObjectFactory();

com.acme.foo.PurchaseOrder po =
  objFactory.createPurchaseOrder();
```

Note: Because multiple `ObjectFactory` classes are generated when there are multiple packages on the `contextPath`, if you have multiple packages on the `contextPath`, you should use the complete package name when referencing an `ObjectFactory` class in one of those packages.

Once the client application has an instance of the schema-derived object, it can use the mutator methods to set content on it.

Note: The JAXB provider implementation must generate a class in each package that contains all of the necessary object factory methods for that package named `ObjectFactory` as well as the `newInstance(javaContentInterface)` method.

More About Marshalling

The `Marshaller` class in the `javax.xml.bind` package provides the client application the ability to convert a Java content tree back into XML data. There is no difference between marshalling a content tree that is created manually using the factory methods and marshalling a content tree that is the result an unmarshal operation. Clients can marshal a Java content tree back to XML data to a java.io.OutputStream or a java.io.Writer. The marshalling process can alternatively produce SAX2 event streams to a registered ContentHandler or produce a DOM Node object.

A simple example that unmarshals an XML document and then marshals it back out is a follows:

```
JAXBContext jc = JAXBContext.newInstance( "com.acme.foo" );

// unmarshal from foo.xml
Unmarshaller u = jc.createUnmarshaller();
FooObject fooObj =
  (FooObject)u.unmarshal( new File( "foo.xml" ) );

// marshal to System.out
Marshaller m = jc.createMarshaller();
m.marshal( fooObj, System.out );
```

By default, the Marshaller uses UTF-8 encoding when generating XML data to a java.io.OutputStream or a java.io.Writer. Use the setProperty API to change the output encoding used during these marshal operations. Client applications are expected to supply a valid character encoding name as defined in the W3C XML 1.0 Recommendation (http://www.w3.org/TR/2000/REC-xml-20001006#charencoding) and supported by your Java Platform.

Client applications are not required to validate the Java content tree prior to calling one of the marshal APIs. There is also no requirement that the Java content tree be valid with respect to its original schema in order to marshal it back into XML data. Different JAXB Providers can support marshalling invalid Java content trees at varying levels, however all JAXB providers must be able to marshal a valid content tree back to XML data. A JAXB provider must throw a MarshalException when it is unable to complete the marshal operation due to invalid content. Some JAXB providers will fully allow marshalling invalid content, others will fail on the first validation error.

Table 1–2 shows the properties that the `Marshaller` class supports.

Table 1–2 Marshaller Properties

Property	**Description**
`jaxb.encoding`	Value must be a `java.lang.String`; the output encoding to use when marshalling the XML data. The `Marshaller` will use “UTF-8” by default if this property is not specified.
`jaxb.formatted.output`	Value must be a `java.lang.Boolean`; controls whether or not the `Marshaller` will format the resulting XML data with line breaks and indentation. A `true` value for this property indicates human readable indented XML data, while a `false` value indicates unformatted XML data. The `Marshaller` defaults to `false` (unformatted) if this property is not specified.
`jaxb.schemaLocation`	Value must be a `java.lang.String`; allows the client application to specify an `xsi:schemaLocation` attribute in the generated XML data. The format of the `schemaLocation` attribute value is discussed in an easy to understand, non-normative form in Section 5.6 of the *W3C XML Schema Part 0: Primer* and specified in Section 2.6 of the *W3C XML Schema Part 1: Structures*.
`jaxb.noNamespaceSchemaLocation`	Value must be a `java.lang.String`; allows the client application to specify an `xsi:noNamespaceSchemaLocation` attribute in the generated XML data.

More About Validation

The `Validator` class in the `javax.xml.bind` package is responsible for controlling the validation of content trees during runtime. When the unmarshalling process incorporates validation and it successfully completes without any validation errors, both the input document and the resulting content tree are guaranteed to be valid. By contrast, the marshalling process does not actually perform validation. If only validated content trees are marshalled, this guarantees that generated XML documents are always valid with respect to the source schema.

Some XML parsers, like SAX and DOM, allow schema validation to be disabled, and there are cases in which you may want to disable schema validation to improve processing speed and/or to process documents containing invalid or incomplete content. JAXB supports these processing scenarios by means of the exception handling you choose implement in your JAXB-enabled application. In general, if a JAXB implementation cannot unambiguously complete unmarshalling or marshalling, it will terminate processing with an exception.

Note: The `Validator` class is responsible for managing On-Demand Validation (see below). The `Unmarshaller` class is responsible for managing Unmarshal-Time Validation during the unmarshal operations. Although there is no formal method of enabling validation during the marshal operations, the `Marshaller` may detect errors, which will be reported to the `ValidationEventHandler` registered on it.

A JAXB client can perform two types of validation:

- **Unmarshal-Time validation** enables a client application to receive information about validation errors and warnings detected while unmarshalling XML data into a Java content tree, and is completely orthogonal to the other types of validation. To enable or disable it, use the `Unmarshaller.setValidating` method. All JAXB Providers are required to support this operation.
- **On-Demand validation** enables a client application to receive information about validation errors and warnings detected in the Java content tree. At any point, client applications can call the `Validator.validate` method on the Java content tree (or any sub-tree of it). All JAXB Providers are required to support this operation.

If the client application does not set an event handler on its `Validator`, `Unmarshaller`, or `Marshaller` prior to calling the validate, unmarshal, or marshal methods, then a default event handler will receive notification of any errors or warnings encountered. The default event handler will cause the current operation to halt after encountering the first error or fatal error (but will attempt to continue after receiving warnings).

There are three ways to handle events encountered during the unmarshal, validate, and marshal operations:

- Use the default event handler.

The default event handler will be used if you do not specify one via the `setEventHandler` APIs on `Validator`, `Unmarshaller`, or `Marshaller`.

- Implement and register a custom event handler.

  Client applications that require sophisticated event processing can implement the `ValidationEventHandler` interface and register it with the `Unmarshaller` and/or `Validator`.

- Use the `ValidationEventCollector` utility.

  For convenience, a specialized event handler is provided that simply collects any `ValidationEvent` objects created during the unmarshal, validate, and marshal operations and returns them to the client application as a `java.util.Collection`.

Validation events are handled differently, depending on how the client application is configured to process them. However, there are certain cases where a JAXB Provider indicates that it is no longer able to reliably detect and report errors. In these cases, the JAXB Provider will set the severity of the `ValidationEvent` to `FATAL_ERROR` to indicate that the unmarshal, validate, or marshal operations should be terminated. The default event handler and `ValidationEventCollector` utility class must terminate processing after being notified of a fatal error. Client applications that supply their own `ValidationEventHandler` should also terminate processing after being notified of a fatal error. If not, unexpected behavior may occur.

XML Schemas

Because XML schemas are such an important component of the JAXB processing model—and because other data binding facilities like JAXP work with DTDs instead of schemas—it is useful to review here some basics about what XML schemas are and how they work.

XML Schemas are a powerful way to describe allowable elements, attributes, entities, and relationships in an XML document. A more robust alternative to DTDs, the purpose of an XML schema is to define classes of XML documents that must adhere to a particular set of structural and data constraints—that is, you may want to define separate schemas for chapter-oriented books, for an online purchase order system, or for a personnel database. In the context of JAXB, an XML document containing data that is constrained by an XML schema is referred to as a *document instance*, and the structure and data within a document instance is referred to as a *content tree*.

Note: In practice, the term “document” is not always accurate, as an XML instance document does not have to be a completely formed, selfstanding document file; it can instead take the form of streams of data passed between applications, or of sets of database fields, or of *XML infosets* in which blocks of information contain just enough information to describe where they fit in the schema structure.

The following sample code is taken from the W3C's *Schema Part 0: Primer* (`http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/`), and illustrates an XML document, `po.xml`, for a simple purchase order.

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
<comment>Hurry, my lawn is going wild!</comment>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <comment>Confirm this is electric</comment>
    </item>
    <item partNum="926-AA">
      <productName>Baby Monitor</productName>
      <quantity>1</quantity>
      <USPrice>39.98</USPrice>
      <shipDate>1999-05-21</shipDate>
    </item>
  </items>
</purchaseOrder>
```

The root element, `purchaseOrder`, contains the child elements `shipTo`, `billTo`, `comment`, and `items`. All of these child elements except `comment` contain other child elements. The leaves of the tree are the child elements like `name`, `street`, `city`, and `state`, which do not contain any further child elements. Elements that contain other child elements or can accept attributes are referred to as *complex types*. Elements that contain only `PCDATA` and no child elements are referred to as *simple types*.

The complex types and some of the simple types in `po.xml` are defined in the purchase order schema below. Again, this example schema, `po.xsd`, is derived from the W3C's *Schema Part 0: Primer* (`http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/`).

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
<xsd:element name="comment" type="xsd:string"/>
<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
     <xsd:element name="shipTo" type="USAddress"/>
     <xsd:element name="billTo" type="USAddress"/>
     <xsd:element ref="comment" minOccurs="0"/>
     <xsd:element name="items" type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>

<xsd:complexType name="USAddress">
  <xsd:sequence>
     <xsd:element name="name" type="xsd:string"/>
     <xsd:element name="street" type="xsd:string"/>
     <xsd:element name="city" type="xsd:string"/>
     <xsd:element name="state" type="xsd:string"/>
     <xsd:element name="zip" type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN"
       fixed="US"/>
</xsd:complexType>

<xsd:complexType name="Items">
  <xsd:sequence>
     <xsd:element name="item" minOccurs="1"
                  maxOccurs="unbounded">
        <xsd:complexType>
           <xsd:sequence>
              <xsd:element name="productName"
                           type="xsd:string"/>
              <xsd:element name="quantity">
                 <xsd:simpleType>
                    <xsd:restriction base="xsd:positiveInteger">
                    <xsd:maxExclusive value="100"/>
                  </xsd:restriction>
                </xsd:simpleType>
              </xsd:element>
              <xsd:element name="USPrice" type="xsd:decimal"/>
              <xsd:element ref="comment" minOccurs="0"/>
              <xsd:element name="shipDate" type="xsd:date"
                           minOccurs="0"/>
            </xsd:sequence>
            <xsd:attribute name="partNum" type="SKU"
                           use="required"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>

    <!-- Stock Keeping Unit, a code for identifying products -->
    <xsd:simpleType name="SKU">
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="\d{3}-[A-Z]{2}"/>
      </xsd:restriction>
    </xsd:simpleType>

    </xsd:schema>
```

In this example, the schema comprises, similar to a DTD, a main or root `schema` element and several child elements, `element`, `complexType`, and `simpleType`. Unlike a DTD, this schema also specifies as attributes data types like `decimal`, `date`, `fixed`, and `string`. The schema also specifies constraints like `pattern value`, `minOccurs`, and `positiveInteger`, among others. In DTDs, you can only specify data types for textual data (`PCDATA` and `CDATA`); XML schema supports more complex textual and numeric data types and constraints, all of which have direct analogs in the Java language.

Note that every element in this schema has the prefix `xsd:`, which is associated with the W3C XML Schema namespace. To this end, the namespace declaration, `xmlns:xsd="http://www.w3.org/2001/XMLSchema"`, is declared as an attribute to the `schema` element.

Namespace support is another important feature of XML schemas because it provides a means to differentiate between elements written against different schemas or used for varying purposes, but which may happen to have the same name as other elements in a document. For example, suppose you declared two namespaces in your schema, one for `foo` and another for `bar`. Two XML documents are combined, one from a billing database and another from an shipping database, each of which was written against a different schema. By specifying namespaces in your schema, you can differentiate between, say, `foo:address` and `bar:address`.

Representing XML Content

This section describes how JAXB represents XML content as Java objects. Specifically, the topics in this section are as follows:

- Binding XML Names to Java Identifiers
- Java Representation of XML Schema

Binding XML Names to Java Identifiers

XML schema languages use *XML names*—strings that match the *Name* production defined in *XML 1.0 (Second Edition)* (`http://www.w3.org/XML/`) to label schema components. This set of strings is much larger than the set of valid Java class, method, and constant identifiers. To resolve this discrepancy, JAXB uses several name-mapping algorithms.

The JAXB name-mapping algorithm maps XML names to Java identifiers in a way that adheres to standard Java API design guidelines, generates identifiers that retain obvious connections to the corresponding schema, and is unlikely to result in many collisions.

Refer to Chapter 2 for information about changing default XML name mappings. See Appendix C in the *JAXB Specification* for complete details about the JAXB naming algorithm.

Java Representation of XML Schema

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

- A name, which is either derived directly from the XML namespace URI, or specified by a binding customization of the XML namespace URI
- A set of Java content interfaces representing the content models declared within the schema
- A Set of Java element interfaces representing element declarations occurring within the schema

- An `ObjectFactory` class containing:
  - An instance factory method for each Java content interface and Java element interface within the package; for example, given a Java content interface named `Foo`, the derived factory method would be:

    ```
    public Foo createFoo() throws JAXBException;
    ```

  - Dynamic instance factory allocator; creates an instance of the specified Java content interface; for example:

    ```
    public Object newInstance(Class javaContentInterface)
       throws JAXBException;
    ```

  - `getProperty` and `setProperty` APIs that allow the manipulation of provider-specified properties
- Set of typesafe enum classes
- Package javadoc

Binding XML Schemas

This section describes the default XML-to-Java bindings used by JAXB. All of these bindings can be overridden on global or case-by-case levels by means of a custom binding declaration. The topics in this section are as follows:

- Simple Type Definitions
- Default Data Type Bindings
- Default Binding Rules Summary

See the *JAXB Specification* for complete information about the default JAXB bindings.

Simple Type Definitions

A schema component using a simple type definition typically binds to a Java property. Since there are different kinds of such schema components, the following Java property attributes (common to the schema components) include:

- Base type
- Collection type, if any

- Predicate

The rest of the Java property attributes are specified in the schema component using the `simple` type definition.

Default Data Type Bindings

The Java language provides a richer set of data type than XML schema. Table 1–3 lists the mapping of XML data types to Java data types in JAXB.

Table 1–3 JAXB Mapping of XML Schema Built-in Data Types

XML Schema Type	Java Data Type
`xsd:string`	`java.lang.String`
`xsd:integer`	`java.math.BigInteger`
`xsd:int`	`int`
`xsd.long`	`long`
`xsd:short`	`short`
`xsd:decimal`	`java.math.BigDecimal`
`xsd:float`	`float`
`xsd:double`	`double`
`xsd:boolean`	`boolean`
`xsd:byte`	`byte`
`xsd:QName`	`javax.xml.namespace.QName`
`xsd:dateTime`	`java.util.Calendar`
`xsd:base64Binary`	`byte[]`
`xsd:hexBinary`	`byte[]`
`xsd:unsignedInt`	`long`
`xsd:unsignedShort`	`int`
`xsd:unsignedByte`	`short`

Table 1–3 JAXB Mapping of XML Schema Built-in Data Types (Continued)

XML Schema Type	Java Data Type
`xsd:time`	`java.util.Calendar`
`xsd:date`	`java.util.Calendar`
`xsd:anySimpleType`	`java.lang.String`

Default Binding Rules Summary

The JAXB binding model follows the default binding rules summarized below:

- Bind the following to Java package:
    - XML Namespace URI
- Bind the following XML Schema components to Java content interface:
    - Named complex type
    - Anonymous inlined type definition of an element declaration
- Bind to typesafe enum class:
    - A named simple type definition with a basetype that derives from "`xsd:NCName`" and has enumeration facets.
- Bind the following XML Schema components to a Java Element interface:
    - A global element declaration to a Element interface.
    - Local element declaration that can be inserted into a general content list.
- Bind to Java property:
    - Attribute use
    - Particle with a term that is an element reference or local element declaration.
- Bind model group with a repeating occurrence and complex type definitions with mixed `{content type}` to:
    - A general content property; a List content-property that holds Java instances representing element information items and character data items.

Customizing JAXB Bindings

The default JAXB bindings can be overridden at a global scope or on a case-by-case basis as needed by using custom binding declarations. As described previously, JAXB uses default binding rules that can be customized by means of binding declarations made in either of two ways:

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

Custom JAXB binding declarations also allow you to customize your generated JAXB classes beyond the XML-specific constraints in an XML schema to include Java-specific refinements such as class and package name mappings.

You do not need to provide a binding instruction for every declaration in your schema to generate Java classes. For example, the binding compiler uses a general name-mapping algorithm to bind XML names to names that are acceptable in the Java programming language. However, if you want to use a different naming scheme for your classes, you can specify custom binding declarations to make the binding compiler generate different names. There are many other customizations you can make with the binding declaration, including:

- Name the package, derived classes, and methods
- Assign types to the methods within the derived classes
- Choose which elements to bind to classes
- Decide how to bind each attribute and element declaration to a property in the appropriate content class
- Choose the type of each attribute-value or content specification

Note: Relying on the default JAXB binding behavior rather than requiring a binding declaration for each XML Schema component bound to a Java representation makes it easier to keep pace with changes in the source schema. In most cases, the default rules are robust enough that a usable binding can be produced with no custom binding declaration at all.

Code examples showing how to customize JAXB bindings are provided in Chapter 2.

Scope

When a customization value is defined in a binding declaration, it is associated with a *scope*. A scope of a customization value is the set of schema elements to which it applies. If a customization value applies to a schema element, then the schema element is said to be covered by the scope of the customization value.

Table 1–4 lists the four scopes for custom bindings.

Table 1–4 Custom Binding Scopes

Scope	Description
Global	A customization value defined in `<globalBindings>` has global scope. A global scope covers all the schema elements in the source schema and (recursively) any schemas that are included or imported by the source schema.
Schema	A customization value defined in `<schemaBindings>` has schema scope. A schema scope covers all the schema elements in the target name space of a schema.
Definition	A customization value in binding declarations of a type definition and global declaration has definition scope. A definition scope covers all schema elements that reference the type definition or the global declaration.
Component	A customization value in a binding declaration has component scope if the customization value applies only to the schema element that was annotated with the binding declaration.

Scope Inheritance

The different scopes form a taxonomy. The taxonomy defines both the inheritance and overriding semantics of customization values. A customization value defined in one scope is inherited for use in a binding declaration covered by another scope as shown by the following inheritance hierarchy:

- A schema element in schema scope inherits a customization value defined in global scope.
- A schema element in definition scope inherits a customization value defined in schema or global scope.
- A schema element in component scope inherits a customization value defined in definition, schema or global scope.

Similarly, a customization value defined in one scope can override a customization value inherited from another scope as shown below:

- Value in schema scope overrides a value inherited from global scope.
- Value in definition scope overrides a value inherited from schema scope or global scope.
- Value in component scope overrides a value inherited from definition, schema or global scope.

What is Not Supported

See Section E.2, "Not Required XML Schema Concepts," in the *JAXB Specification* for the latest information about unsupported or non-required schema concepts.

JAXB APIs and Tools

The JAXB APIs and tools are shipped in the `jaxb` subdirectory of the Java WSDP. This directory contains sample applications, a JAXB binding compiler (`xjc`), and implementations of the runtime binding framework APIs contained in the `javax.xml.bind` package. For instructions on using the JAXB, see Chapter 2.

2 Using JAXB

THIS chapter provides instructions for using several of the sample Java applications that were included in the Java WSDP. These examples demonstrate and build upon key JAXB features and concepts. It is recommended that you follow these procedures in the order presented.

After reading this chapter, you should feel comfortable enough with JAXB that you can:

- Generate JAXB Java classes from an XML schema
- Use schema-derived JAXB classes to unmarshal and marshal XML content in a Java application
- Create a Java content tree from scratch using schema-derived JAXB classes
- Validate XML content during unmarshalling and at runtime
- Customize JAXB schema-to-Java bindings

The primary goals of the basic examples are to highlight the core set of JAXB functions using default settings and bindings. After familiarizing yourself with these core features and functions, you may wish to continue with Customizing JAXB Bindings (page 56) for instructions on using five additional examples that demonstrate how to modify the default JAXB bindings.

Note: The Purchase Order schema, `po.xsd`, and the Purchase Order XML file, `po.xml`, used in these samples are derived from the W3C XML Schema Part 0: Primer (`http://www.w3.org/TR/xmlschema-0/`), edited by David C. Fallside.

General Usage Instructions

This section provides general usage instructions for the examples used in this chapter, including how to build and run the applications both manually and using the Ant build tool, and provides details about the default schema-to-JAXB bindings used in these examples.

Description

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

Table 2–1 Sample JAXB Application Descriptions

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

Table 2–1 Sample JAXB Application Descriptions

Example Name	**Description**
Datatype Converter Example	Similar to the Customize Inline example, this example illustrates alternate, more terse bindings of XML `simpleType` definitions to Java datatypes.
External Customize Example	Illustrates how to use an external binding declarations file to pass binding customizations for a read-only schema to the JAXB binding compiler.
Fix Collides Example	Illustrates how to use customizations to resolve name conflicts reported by the JAXB binding compiler. It is recommended that you first run `ant fail` in the application directory to see the errors reported by the JAXB binding compiler, and then look at `binding.xjb` to see how the errors were resolved. Running `ant` alone uses the binding customizations to resolve the name conflicts while compiling the schema.
Bind Choice Example	Illustrates how to bind a `choice` model group to a Java interface.

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

Each example directory contains several base files:

- `po.xsd` is the XML schema you will use as input to the JAXB binding compiler, and from which schema-derived JAXB Java classes will be generated. For the Customize Inline and Datatype Converter examples, this file contains inline binding customizations. Note that the Bind Choice and Fix Collides examples use `example.xsd` rather than `po.xsd`.
- `po.xml` is the *Purchase Order* XML file containing sample XML content, and is the file you will unmarshal into a Java content tree in each example. This file is almost exactly the same in each example, with minor content differences to highlight different JAXB concepts. Note that the Bind Choice and Fix Collides examples use `example.xml` rather than `po.xml`.
- `Main.java` is the main Java class for each example.
- `build.xml` is an Ant project file provided for your convenience. As shown later in this chapter, you can generate and compile schema-derived JAXB classes manually using standard Java and JAXB commands, or you can use

Ant to generate, compile, and run the classes automatically. The `build.xml` file varies across the examples.

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

Using the Examples

As with all applications that implement schema-derived JAXB classes, as described above, there are two distinct phases in using JAXB:

1. Generating and compiling JAXB Java classes from an XML source schema
2. Unmarshalling, validating, processing, and marshalling XML content

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

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

Configuring and Running the Examples Manually

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

Solaris/Linux

1. Set the following environment variables:

   ```
   export JAVA_HOME=<your J2SE installation directory>
   export JWSDP_HOME=<your JWSDP 1.4 installation directory>
   ```

2. Change to the desired example directory.

   For example, to run the Unmarshal Read example:

   ```
   cd <JWSDP_HOME>/jaxb/samples/unmarshal-read
   ```

   (*<JWSDP_HOME>* is the directory where you installed the Java WSDP 1.4 bundle.)

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

   ```
   $JWSDP_HOME/jaxb/bin/xjc.sh po.xsd -p primer.po
   ```

   `po.xsd` is the name of the source XML schema. The `-p primer.po` switch tells the JAXB compiler to put the generated classes in a Java package named `primer.po`. For the purposes of this example, the package name must be `primer.po`. See JAXB Compiler Options (page 32) for a complete list of JAXB binding compiler options.

4. Generate API documentation for the application using the Javadoc tool (optional).

```
$JAVA_HOME/bin/javadoc -package primer.po -sourcepath .
-d docs/api -windowtitle "Generated Interfaces for po.xsd"
```

5. Compile the generated JAXB Java classes.

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

6. Run the `Main` class.

   ```
   $JAVA_HOME/bin/java Main
   ```

   The `po.xml` file is unmarshalled into a Java content tree, and the XML data in the content tree is written to `System.out`.

Windows NT/2000/XP

1. Set the following environment variable:

   ```
   set JAVA_HOME=<your J2SE installation directory>
   set JWSDP_HOME=<your JWSDP 1.4 installation directory>
   ```

2. Change to the desired example directory.

   For example, to run the Unmarshal Read example:

   ```
   cd <JWSDP_HOME>\jaxb\samples\unmarshal-read
   ```

   (<*JWSDP_HOME*> is the directory where you installed the Java WSDP 1.4 bundle.)

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

   ```
   %JWSDP_HOME%\jaxb\bin\xjc.bat po.xsd -p primer.po
   ```

   `po.xsd` is the name of the source XML schema. The `-p primer.po` switch tells the JAXB compiler to put the generated classes in a Java package named `primer.po`. For the purposes of this example, the package name must be `primer.po`. See JAXB Compiler Options (page 32) for a complete list of JAXB binding compiler options.

4. Generate API documentation for the application using the Javadoc tool (optional).

   ```
   %JAVA_HOME%\bin\javadoc -package primer.po -sourcepath .
   -d docs\api -windowtitle "Generated Interfaces for po.xsd"
   ```

5. Compile the schema-derived JAXB Java classes.

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

6. Run the `Main` class.

   ```
   %JAVA_HOME%\bin\java Main
   ```

   The `po.xml` file is unmarshalled into a Java content tree, and the XML data in the content tree is written to `System.out`.

The schema-derived JAXB classes and how they are bound to the source schema is described in About the Schema-to-Java Bindings (page 34). The methods used for building and processing the Java content tree in each of the basic examples are analyzed in Basic Examples (page 45).

Configuring and Running the Samples With Ant

The `build.xml` file included in each example directory is an Ant project file that, when run, automatically performs all the steps listed in Configuring and Running the Examples Manually (page 28). Specifically, using Ant with the included `build.xml` project files does the following:

1. Updates your `CLASSPATH` to include the necessary schema-derived JAXB classes.
2. Runs the JAXB binding compiler to generate JAXB Java classes from the XML source schema, `po.xsd`, and puts the classes in a package named `primer.po`.
3. Generates API documentation from the schema-derived JAXB classes using the Javadoc tool.
4. Compiles the schema-derived JAXB classes.
5. Runs the `Main` class for the example.

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

Solaris/Linux

1. Set the following environment variables:

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

2. Change to the desired example directory.

   For example, to run the Unmarshal Read example:

   ```
   cd <JWSDP_HOME>/jaxb/samples/unmarshal-read
   ```

   (*<JWSDP_HOME>* is the directory where you installed the Java WSDP 1.4 bundle.)

3. Run ant:

   ```
   $JWSDP_HOME/apache-ant/bin/ant -emacs
   ```

4. Repeat these steps for each example.

Windows NT/2000/XP

1. Set the following environment variables:

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

2. Change to the desired example directory.

   For example, to run the Unmarshal Read example:

   ```
   cd <JWSDP_HOME>\jaxb\samples\unmarshal-read
   ```

   (`<JWSDP_HOME>` is the directory where you installed the Java WSDP 1.4 bundle.)

3. Run ant:

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

4. Repeat these steps for each example.

The schema-derived JAXB classes and how they are bound to the source schema is described in About the Schema-to-Java Bindings (page 34). The methods used for building and processing the Java content tree are described in Basic Examples (page 45).

JAXB Compiler Options

The JAXB schema binding compiler is located in the `<JWSDP_HOME>/jaxb/bin` directory. There are two scripts in this directory: `xjc.sh` (Solaris/Linux) and `xjc.bat` (Windows).

Both `xjc.sh` and `xjc.bat` take the same command-line options. You can display quick usage instructions by invoking the scripts without any options, or with the `-help` switch. The syntax is as follows:

```
xjc [-options ...] <schema>
```

The `xjc` command-line options are listed in Table 2–2.

Table 2–2 `xjc` Command-Line Options

Option or Argument	**Description**
`<schema>`	One or more schema files to compile.

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

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, `xjc` strictly enforces the rules outlined in the Compatibility chapter of the *JAXB Specification*. Specifically, Appendix E.2 defines a set of W3C XML Schema features that are not completely supported by JAXB v1.0. In some cases, you may be able to use these extensions with the `-extension` switch. In the default (strict) mode, you are also limited to using only the binding customizations defined in the specification. By using the `-extension` switch, you can enable the JAXB Vendor Extensions.

`-b` *`<file>`*	Specify one or more external binding files to process (each binding file must have it's own `-b` switch). The syntax of the external binding files is extremely flexible. You may have a single binding file that contains customizations for multiple schemas, or you can break the customizations into multiple bindings files; for example:
	
	`xjc schema1.xsd schema2.xsd schema3.xsd -b bindings123.xjb`
	`xjc schema1.xsd schema2.xsd schema3.xsd -b bindings1.xjb -b bindings2.xjb -b bindings3.xjb`
	
	Note that the ordering of schema files and binding files on the command line does not matter.

`-d` *`<dir>`*	By default, `xjc` will generate Java content classes in the current directory. Use this option to specify an alternate output directory. The directory must already exist; `xjc` will not create it for you.
`-p` *`<pkg>`*	Specifies the target package for schema-derived classes. This option overrides any binding customization for package name as well as the default package name algorithm defined in the *JAXB Specification*.
`-host` *`<proxyHost>`*	Set `http.proxyHost` to *`<proxyHost>`*.
`-port` *`<proxyPort>`*	Set `http.proxyPort` to *`<proxyPort>`*.

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

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

The command invoked by the `xjc.sh` and `xjc.bat` scripts is equivalent to the Java command:

```
$JAVA_HOME/bin/java -jar $JAXB_HOME/lib/jaxb-xjc.jar
```

About the Schema-to-Java Bindings

When you run the JAXB binding compiler against the `po.xsd` XML schema used in the basic examples (Unmarshal Read, Modify Marshal, Create Marshal, Unmarshal Validate, Validate-On-Demand), the JAXB binding compiler generates a Java package named `primer.po` containing eleven classes, making a total of twelve classes in each of the basic examples:

Table 2–3 Schema-Derived JAXB Classes in the Basic Examples

Class	Description
`primer/po/Comment.java`	Public interface extending `javax.xml.bind.Element`; binds to the global schema `element` named `comment`. Note that JAXB generates element interfaces for all global element declarations.
`primer/po/Items.java`	Public interface that binds to the schema `complexType` named `Items`.
`primer/po/ObjectFactory.java`	Public class extending `com.sun.xml.bind.DefaultJAXB-ContextImpl`; used to create instances of specified interfaces. For example, the `ObjectFactory` `createComment()` method instantiates a `Comment` object.
`primer/po/PurchaseOrder.java`	Public interface extending `javax.xml.bind.Element`, and `PurchaseOrderType`; binds to the global schema `element` named `PurchaseOrder`.
`primer/po/PurchaseOrderType.java`	Public interface that binds to the schema `complexType` named `PurchaseOrderType`.
`primer/po/USAddress.java`	Public interface that binds to the schema `complexType` named `USAddress`.
`primer/po/impl/CommentImpl.java`	Implementation of `Comment.java`.
`primer/po/impl/ItemsImpl.java`	Implementation of `Items.java`
`primer/po/impl/PurchaseOrderImpl.java`	Implementation of `PurchaseOrder.java`
`primer/po/impl/PurchaseOrderType-Impl.java`	Implementation of `PurchaseOrderType.java`
`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 2–4 Schema-to-Java Bindings for the Basic Examples

XML Schema	JAXB Binding
`<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">`	
`<xsd:element name="purchaseOrder" type="PurchaseOrderType"/>`	`PurchaseOrder.java`
`<xsd:element name="comment" type="xsd:string"/>`	`Comment.java`

`<xsd:complexType name="PurchaseOrderType">`
`  <xsd:sequence>`
`    <xsd:element name="shipTo" type="USAddress"/>`
`    <xsd:element name="billTo" type="USAddress"/>`
`    <xsd:element ref="comment" minOccurs="0"/>`
`    <xsd:element name="items" type="Items"/>`
`  </xsd:sequence>`
`  <xsd:attribute name="orderDate" type="xsd:date"/>`
`</xsd:complexType>`
	`PurchaseOrder-`
	`Type.java`

`<xsd:complexType name="USAddress">`
`  <xsd:sequence>`
`    <xsd:element name="name" type="xsd:string"/>`
`    <xsd:element name="street" type="xsd:string"/>`
`    <xsd:element name="city" type="xsd:string"/>`
`    <xsd:element name="state" type="xsd:string"/>`
`    <xsd:element name="zip" type="xsd:decimal"/>`
`  </xsd:sequence>`
`<xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>`
`</xsd:complexType>`
	`USAddress.java`

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

Table 2–4 Schema-to-Java Bindings for the Basic Examples (Continued)

XML Schema	JAXB Binding

`      <xsd:complexType>`
`        <xsd:sequence>`
`         <xsd:element name="productName" type="xsd:string"/>`
`         <xsd:element name="quantity">`
`           <xsd:simpleType>`
`             <xsd:restriction base="xsd:positiveInteger">`
`               <xsd:maxExclusive value="100"/>`
`             </xsd:restriction>`
`           </xsd:simpleType>`
`         </xsd:element>`
`         <xsd:element name="USPrice" type="xsd:decimal"/>`
`         <xsd:element ref="comment" minOccurs="0"/>`
`<xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>`
`        </xsd:sequence>`
`  <xsd:attribute name="partNum" type="SKU" use="required"/>`
`      </xsd:complexType>`
	`Items.ItemType`

`    </xsd:element>`
`  </xsd:sequence>`
`</xsd:complexType>`	

`<!-- Stock Keeping Unit, a code for identifying products -->`	

`<xsd:simpleType name="SKU">`
`  <xsd:restriction base="xsd:string">`
`    <xsd:pattern value="\d{3}-[A-Z]{2}"/>`
`  </xsd:restriction>`
`</xsd:simpleType>`	

`</xsd:schema>`	

Schema-Derived JAXB Classes

The code for the individual classes generated by the JAXB binding compiler for the basic examples is listed below, followed by brief explanations of its functions. The classes listed here are:

- `Comment.java`
- `Items.java`
- `ObjectFactory.java`
- `PurchaseOrder.java`
- `PurchaseOrderType.java`
- `USAddress.java`

Comment.java

In `Comment.java`:

- The `Comment.java` class is part of the `primer.po` package.
- `Comment` is a public interface that extends `javax.xml.bind.Element`.
- Content in instantiations of this class bind to the XML schema element named `comment`.
- The `getValue()` and `setValue()` methods are used to get and set strings representing XML `comment` elements in the Java content tree.

The `Comment.java` code looks like this:

```
package primer.po;

public interface Comment
    extends javax.xml.bind.Element
{

    String getValue();
    void setValue(String value);
}
```

Items.java

In `Items.java`, below:

- The `Items.java` class is part of the `primer.po` package.
- The class provides public interfaces for `Items` and `ItemType`.
- Content in instantiations of this class bind to the XML ComplexTypes `Items` and its child element `ItemType`.
- `Item` provides the `getItem()` method.
- `ItemType` provides methods for:
  - `getPartNum();`
  - `setPartNum(String value);`
  - `getComment();`
  - `setComment(java.lang.String value);`
  - `getUSPrice();`
  - `setUSPrice(java.math.BigDecimal value);`
  - `getProductName();`
  - `setProductName(String value);`
  - `getShipDate();`

- `setShipDate(java.util.Calendar value);`
- `getQuantity();`
- `setQuantity(java.math.BigInteger value);`

The `Items.java` code looks like this:

```
package primer.po;

public interface Items {
   java.util.List getItem();

   public interface ItemType {
        String getPartNum();
        void setPartNum(String value);
        java.lang.String getComment();
        void setComment(java.lang.String value);
        java.math.BigDecimal getUSPrice();
        void setUSPrice(java.math.BigDecimal value);
        String getProductName();
        void setProductName(String value);
        java.util.Calendar getShipDate();
        void setShipDate(java.util.Calendar value);
        java.math.BigInteger getQuantity();
        void setQuantity(java.math.BigInteger value);
    }
}
```

ObjectFactory.java

In `ObjectFactory.java`, below:

- The `ObjectFactory` class is part of the `primer.po` package.
- `ObjectFactory` provides factory methods for instantiating Java interfaces representing XML content in the Java content tree.
- Method names are generated by concatenating:
  - The string constant `create`
  - If the Java content interface is nested within another interface, then the concatenation of all outer Java class names
  - The name of the Java content interface
  - JAXB implementation-specific code was removed in this example to make it easier to read.

For example, in this case, for the Java interface `primer.po.Items.ItemType`, `ObjectFactory` creates the method `createItemsItemType()`.

The ObjectFactory.java code looks like this:

```
package primer.po;

public class ObjectFactory
    extends com.sun.xml.bind.DefaultJAXBContextImpl {

    /**
     * Create a new ObjectFactory that can be used to create
     * new instances of schema derived classes for package:
     * primer.po
     */
    public ObjectFactory() {
        super(new primer.po.ObjectFactory.GrammarInfoImpl());
    }

    /**
     * Create an instance of the specified Java content
     * interface.
     */
    public Object newInstance(Class javaContentInterface)
        throws javax.xml.bind.JAXBException
    {
        return super.newInstance(javaContentInterface);
    }

    /**
     * Get the specified property. This method can only be
     * used to get provider specific properties.
     * Attempting to get an undefined property will result
     * in a PropertyException being thrown.
     */
    public Object getProperty(String name)
        throws javax.xml.bind.PropertyException
    {
        return super.getProperty(name);
    }

    /**
     * Set the specified property. This method can only be
     * used to set provider specific properties.
     * Attempting to set an undefined property will result
     * in a PropertyException being thrown.
     */
    public void setProperty(String name, Object value)
        throws javax.xml.bind.PropertyException
    {
        super.setProperty(name, value);
```

```
}

/**
 * Create an instance of PurchaseOrder
 */
public primer.po.PurchaseOrder createPurchaseOrder()
    throws javax.xml.bind.JAXBException
{
    return ((primer.po.PurchaseOrder)
        newInstance((primer.po.PurchaseOrder.class)));
}

/**
 * Create an instance of ItemsItemType
 */
public primer.po.Items.ItemType createItemsItemType()
    throws javax.xml.bind.JAXBException
{
    return ((primer.po.Items.ItemType)
        newInstance((primer.po.Items.ItemType.class)));
}

/**
 * Create an instance of USAddress
 */
public primer.po.USAddress createUSAddress()
    throws javax.xml.bind.JAXBException
{
    return ((primer.po.USAddress)
        newInstance((primer.po.USAddress.class)));
}

/**
 * Create an instance of Comment
 */
public primer.po.Comment createComment()
    throws javax.xml.bind.JAXBException
{
    return ((primer.po.Comment)
        newInstance((primer.po.Comment.class)));
}

/**
 * Create an instance of Comment
 */
public primer.po.Comment createComment(String value)
    throws javax.xml.bind.JAXBException
{
```

```
            return new primer.po.impl.CommentImpl(value);
        }

        /**
         * Create an instance of Items
         */
        public primer.po.Items createItems()
            throws javax.xml.bind.JAXBException
        {
            return ((primer.po.Items)
                newInstance((primer.po.Items.class)));
        }

        /**
         * Create an instance of PurchaseOrderType
         */
        public primer.po.PurchaseOrderType
    createPurchaseOrderType()
            throws javax.xml.bind.JAXBException
        {
            return ((primer.po.PurchaseOrderType)
                newInstance((primer.po.PurchaseOrderType.class)));
        }
    }
```

PurchaseOrder.java

In PurchaseOrder.java, below:

- The PurchaseOrder class is part of the primer.po package.
- PurchaseOrder is a public interface that extends javax.xml.bind.Element and primer.po.PurchaseOrderType.
- Content in instantiations of this class bind to the XML schema element named purchaseOrder.

The PurchaseOrder.java code looks like this:

```
package primer.po;

public interface PurchaseOrder
extends javax.xml.bind.Element, primer.po.PurchaseOrderType{
}
```

PurchaseOrderType.java

In `PurchaseOrderType.java`, below:

- The `PurchaseOrderType` class is part of the `primer.po` package.
- Content in instantiations of this class bind to the XML schema child element named `PurchaseOrderType`.
- `PurchaseOrderType` is a public interface that provides the following methods:
    - `getItems();`
    - `setItems(primer.po.Items value);`
    - `getOrderDate();`
    - `setOrderDate(java.util.Calendar value);`
    - `getComment();`
    - `setComment(java.lang.String value);`
    - `getBillTo();`
    - `setBillTo(primer.po.USAddress value);`
    - `getShipTo();`
    - `setShipTo(primer.po.USAddress value);`

The `PurchaseOrderType.java` code looks like this:

```
package primer.po;

public interface PurchaseOrderType {
    primer.po.Items getItems();
    void setItems(primer.po.Items value);
    java.util.Calendar getOrderDate();
    void setOrderDate(java.util.Calendar value);
    java.lang.String getComment();
    void setComment(java.lang.String value);
    primer.po.USAddress getBillTo();
    void setBillTo(primer.po.USAddress value);
    primer.po.USAddress getShipTo();
    void setShipTo(primer.po.USAddress value);
}
```

USAddress.java

In `USAddress.java`, below:

- The `USAddress` class is part of the `primer.po` package.
- Content in instantiations of this class bind to the XML schema element named `USAddress`.
- `USAddress` is a public interface that provides the following methods:
    - `getState();`
    - `setState(String value);`
    - `getZip();`
    - `setZip(java.math.BigDecimal value);`
    - `getCountry();`
    - `setCountry(String value);`
    - `getCity();`
    - `setCity(String value);`
    - `getStreet();`
    - `setStreet(String value);`
    - `getName();`
    - `setName(String value);`

The `USAddress.java` code looks like this:

```
package primer.po;

public interface USAddress {
    String getState();
    void setState(String value);
    java.math.BigDecimal getZip();
    void setZip(java.math.BigDecimal value);
    String getCountry();
    void setCountry(String value);
    String getCity();
    void setCity(String value);
    String getStreet();
    void setStreet(String value);
    String getName();
    void setName(String value);
}
```

Basic Examples

This section describes five basic examples (Unmarshal Read, Modify Marshal, Create Marshal, Unmarshal Validate, Validate-On-Demand) that demonstrate how to:

- Unmarshal an XML document into a Java content tree and access the data contained within it
- Modify a Java content tree
- Use the `ObjectFactory` class to create a Java content tree from scratch and then marshal it to XML data
- Perform validation during unmarshalling
- Validate a Java content tree at runtime

Unmarshal Read Example

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

1. The `<`*`JWSDP_HOME`*`>/jaxb/samples/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 `<*JWSDP_HOME*>/jaxb/samples/modify-marshal/Main.java` class declares imports for three standard Java classes plus four JAXB binding framework classes and `primer.po` package:

   ```
   import java.io.FileInputStream;
   import java.io.IOException;
   import java.math.BigDecimal;
   import javax.xml.bind.JAXBContext;
   import javax.xml.bind.JAXBException;
   import javax.xml.bind.Marshaller;
   import javax.xml.bind.Unmarshaller;
   import primer.po.*;
   ```

2. A `JAXBContext` instance is created for handling classes generated in `primer.po`.

   ```
   JAXBContext jc = JAXBContext.newInstance( "primer.po" );
   ```

3. An `Unmarshaller` instance is created, and `po.xml` is unmarshalled.

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

4. `set` methods are used to modify information in the `address` branch of the content tree.

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

5. A `Marshaller` instance is created, and the updated XML content is marshalled to `system.out`. The `setProperty` API is used to specify output encoding; in this case formatted (human readable) XML format.

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

Sample Output

Running `java Main` for this example produces the following output:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<purchaseOrder orderDate="1999-10-20-05:00">
<shipTo country="US">
<name>Alice Smith</name>
<street>123 Maple Street</street>
<city>Cambridge</city>
<state>MA</state>
<zip>12345</zip>
</shipTo>
<billTo country="US">
<name>John Bob</name>
<street>242 Main Street</street>
<city>Beverly Hills</city>
<state>CA</state>
<zip>90210</zip>
</billTo>
<items>
<item partNum="242-NO">
```

```
<productName>Nosferatu - Special Edition (1929)</productName>
<quantity>5</quantity>
<USPrice>19.99</USPrice>
</item>
<item partNum="242-MU">
<productName>The Mummy (1959)</productName>
<quantity>3</quantity>
<USPrice>19.98</USPrice>
</item>
<item partNum="242-GZ">
<productName>
Godzilla and Mothra: Battle for Earth/Godzilla vs. King Ghidora
</productName>
<quantity>3</quantity>
<USPrice>27.95</USPrice>
</item>
</items>
</purchaseOrder>
```

Create Marshal Example

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

1. The `<`*`JWSDP_HOME`*`>/jaxb/samples/create-marshal/Main.java` class declares imports for four standard Java classes plus three JAXB binding framework classes and the `primer.po` package:

   ```
   import java.math.BigDecimal;
   import java.math.BigInteger;
   import java.util.Calendar;
   import java.util.List;
   import javax.xml.bind.JAXBContext;
   import javax.xml.bind.JAXBException;
   import javax.xml.bind.Marshaller;
   import primer.po.*;
   ```

2. A `JAXBContext` instance is created for handling classes generated in `primer.po`.

   ```
   JAXBContext jc = JAXBContext.newInstance( "primer.po" );
   ```

3. The `ObjectFactory` class is used to instantiate a new empty `PurchaseOrder` object.

   ```
   // creating the ObjectFactory
   ObjectFactory objFactory = new ObjectFactory();
   ```

```
// create an empty PurchaseOrder
PurchaseOrder po = objFactory.createPurchaseOrder();
```

4. Per the constraints in the `po.xsd` schema, the `PurchaseOrder` object requires a value for the `orderDate` attribute. To satisfy this constraint, the `orderDate` is set using the standard `Calendar.getInstance()` method from `java.util.Calendar`.

```
po.setOrderDate( Calendar.getInstance() );
```

5. The `ObjectFactory` is used to instantiate new empty `USAddress` objects, and the required attributes are set.

```
USAddress shipTo = createUSAddress( "Alice Smith",
                   "123 Maple Street",
                   "Cambridge",
                   "MA",
                   "12345" );
   po.setShipTo( shipTo );

USAddress billTo = createUSAddress( "Robert Smith",
                   "8 Oak Avenue",
                   "Cambridge",
                   "MA",
                   "12345" );
po.setBillTo( billTo );
```

6. The `ObjectFactory` class is used to instantiate a new empty `Items` object.

```
Items items = objFactory.createItems();
```

7. A `get` method is used to get a reference to the `ItemType` list.

```
List itemList = items.getItem();
```

8. `ItemType` objects are created and added to the `Items` list.

```
itemList.add( createItemType(
                "Nosferatu - Special Edition (1929)",
                new BigInteger( "5" ),
                new BigDecimal( "19.99" ),
                null,
                null,
                "242-NO" ) );
itemList.add( createItemType( "The Mummy (1959)",
                new BigInteger( "3" ),
                new BigDecimal( "19.98" ),
                null,
                null,
                "242-MU" ) );
itemList.add( createItemType(
                "Godzilla and Mothra: Battle for Earth/Godzilla
                   vs. King Ghidora",
                new BigInteger( "3" ),
                new BigDecimal( "27.95" ),
                null,
                null,
                "242-GZ" ) );
```

9. The `items` object now contains a list of `ItemType` objects and can be added to the po object.

```
po.setItems( items );
```

10. A `Marshaller` instance is created, and the updated XML content is marshalled to `system.out`. The `setProperty` API is used to specify output encoding; in this case formatted (human readable) XML format.

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

11. An empty `USAddress` object is created and its properties set to comply with the schema constraints.

```
public static USAddress createUSAddress(
                ObjectFactory objFactory,
                String name, String street,
                String city,
                String state,
                String zip )
      throws JAXBException {

   // create an empty USAddress objects
   USAddress address = objFactory.createUSAddress();

   // set properties on it
   address.setName( name );
   address.setStreet( street );
   address.setCity( city );
   address.setState( state );
   address.setZip( new BigDecimal( zip ) );

   // return it
```

```
      return address;
   }
```

12.Similar to the previous step, an empty `ItemType` object is created and its properties set to comply with the schema constraints.

```
   public static Items.ItemType createItemType(ObjectFactory
        objFactory,
                String productName,
                BigInteger quantity,
                BigDecimal price,
                String comment,
                Calendar shipDate,
                String partNum )
        throws JAXBException {

      // create an empty ItemType object
      Items.ItemType itemType =
      objFactory.createItemsItemType();

      // set properties on it
      itemType.setProductName( productName );
      itemType.setQuantity( quantity );
      itemType.setUSPrice( price );
      itemType.setComment( comment );
      itemType.setShipDate( shipDate );
      itemType.setPartNum( partNum );

      // return it
      return itemType;
   }
```

Sample Output

Running `java Main` for this example produces the following output:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<purchaseOrder orderDate="2002-09-24-05:00">
<shipTo>
<name>Alice Smith</name>
<street>123 Maple Street</street>
<city>Cambridge</city>
<state>MA</state>
<zip>12345</zip>
</shipTo>
<billTo>
<name>Robert Smith</name>
<street>8 Oak Avenue</street>
```

```
<city>Cambridge</city>
<state>MA</state>
<zip>12345</zip>
</billTo>
<items>
<item partNum="242-NO">
<productName>Nosferatu - Special Edition (1929)</productName>
<quantity>5</quantity
<USPrice>19.99</USPrice>
</item>
<item partNum="242-MU">
<productName>The Mummy (1959)</productName>
<quantity>3</quantity>
<USPrice>19.98</USPrice>
</item>
<item partNum="242-GZ">
<productName>Godzilla and Mothra: Battle for Earth/Godzilla vs.
King Ghidora</productName>
<quantity>3</quantity>
<USPrice>27.95</USPrice>
</item>
</items>
</purchaseOrder>
```

Unmarshal Validate Example

The Unmarshal Validate example demonstrates how to enable validation during unmarshalling (*Unmarshal-Time Validation*). Note that JAXB provides functions for validation during unmarshalling but not during marshalling. Validation is explained in more detail in More About Validation (page 11).

1. The `<JWSDP_HOME>/jaxb/samples/unmarshal-validate/Main.java` class declares imports for three standard Java classes plus seven JAXB binding framework classes and the `primer.po` package:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.math.BigDecimal;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.UnmarshalException;
import javax.xml.bind.Unmarshaller;
import javax.xml.bind.ValidationEvent;
import javax.xml.bind.util.ValidationEventCollector;
import primer.po.*;
```

2. A `JAXBContext` instance is created for handling classes generated in `primer.po`.

   ```
   JAXBContext jc = JAXBContext.newInstance( "primer.po" );
   ```
3. An `Unmarshaller` instance is created.

   ```
   Unmarshaller u = jc.createUnmarshaller();
   ```
4. The default JAXB Unmarshaller `ValidationEventHandler` is enabled to send to validation warnings and errors to `system.out`. The default configuration causes the unmarshal operation to fail upon encountering the first validation error.

   ```
   u.setValidating( true );
   ```
5. An attempt is made to unmarshal `po.xml` into a Java content tree. For the purposes of this example, the `po.xml` contains a deliberate error.

   ```
   PurchaseOrder po =
      (PurchaseOrder)u.unmarshal(
         new FileInputStream("po.xml"));
   ```
6. The default validation event handler processes a validation error, generates output to `system.out`, and then an exception is thrown.

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

Sample Output

Running `java Main` for this example produces the following output:

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

Validate-On-Demand Example

The Validate-On-Demand example demonstrates how to validate a Java content tree at runtime (*On-Demand Validation*). At any point, client applications can call the `Validator.validate` method on the Java content tree (or any subtree of it). All JAXB Providers are required to support this operation. Validation is explained in more detail in More About Validation (page 11).

1. The <*JWSDP_HOME*>`/jaxb/samples/ondemand-validate/Main.java` class declares imports for five standard Java classes plus nine JAXB Java classes and the `primer.po` package:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.math.BigDecimal;
import java.math.BigInteger;
import java.util.List;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.UnmarshalException;
import javax.xml.bind.Unmarshaller;
import javax.xml.bind.ValidationEvent;
import javax.xml.bind.ValidationException;
import javax.xml.bind.Validator;
import javax.xml.bind.util.ValidationEventCollector;
import primer.po.*;
```

2. A `JAXBContext` instance is created for handling classes generated in `primer.po`.

```
JAXBContext jc = JAXBContext.newInstance( "primer.po" );
```

3. An `Unmarshaller` instance is created, and a valid `po.xml` document is unmarshalled into a Java content tree. Note that `po.xml` is valid at this point; invalid data will be added later in this example.

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

4. A reference is obtained for the first item in the purchase order.

```
Items items = po.getItems();
List itemTypeList = items.getItem();
Items.ItemType item = (Items.ItemType)itemTypeList.get( 0 );
```

5. Next, the item quantity is set to an invalid number. When validation is enabled later in this example, this invalid quantity will throw an exception.

```
item.setQuantity( new BigInteger( "-5" ) );
```

Note: If `@enableFailFastCheck` was `"true"` and the optional `FailFast` validation method was supported by an implementation, a `TypeConstraintException` would be thrown here. Note that the JAXB implementation does not support the `FailFast` feature. Refer to the *JAXB Specification* for more information about `FailFast` validation.

6. A `Validator` instance is created, and the content tree is validated. Note that the `Validator` class is responsible for managing On-Demand validation, whereas the `Unmarshaller` class is responsible for managing Unmarshal-Time validation during unmarshal operations.

```
Validator v = jc.createValidator();
boolean valid = v.validateRoot( po );
System.out.println( valid );
```

7. The default validation event handler processes a validation error, generates output to `system.out`, and then an exception is thrown.

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

Sample Output

Running `java Main` for this example produces the following output:

```
DefaultValidationEventHandler: [ERROR]: "-5" does not satisfy
the "positiveInteger" type
Caught ValidationException
```

Customizing JAXB Bindings

The remainder of this chapter describes several examples that build on the concepts demonstrated in the basic examples.

The goal of this section is to illustrate how to customize JAXB bindings by means of custom binding declarations made in either of two ways:

- As annotations made inline in an XML schema
- As statements in an external file passed to the JAXB binding compiler

Unlike the examples in Basic Examples (page 45), 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 32).

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 2–4.

Figure 2–1 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.

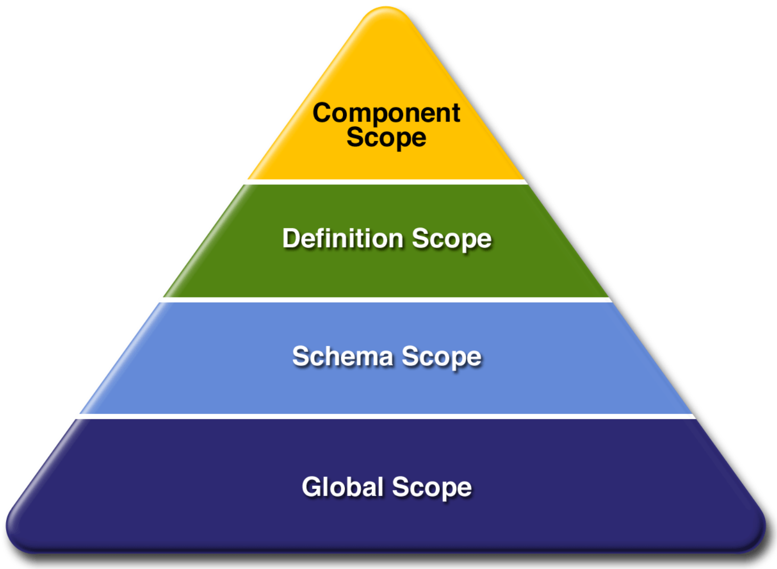

Figure 2–1 Customization Scope Inheritance and Precedence

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 68) 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 66) for more information.

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

Schema Binding Declarations

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

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

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

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

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

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

Class Binding Declarations

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

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

The syntax for `<class>` customizations is:

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

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

Property Binding Declarations

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

The syntax for `<property>` customizations is:

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

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

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

<javaType> Binding Declarations

The `<javaType>` declaration provides a way to customize the translation of XML datatypes to and from Java datatypes. XML provides more datatypes than

Java, and so the <javaType> declaration lets you specify custom datatype bindings when the default JAXB binding cannot sufficiently represent your schema.

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

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

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

The syntax for the <javaType> customization is:

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

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

The `<javaType>` declaration can be used in:

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

See MyDatatypeConverter Class (page 75) 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 63).

The syntax for the `<typesafeEnumClass>` customization is:

```
<typesafeEnumClass[ name = "enumClassName" ]
  [ <typesafeEnumMember> ... </typesafeEnumMember> ]*
  [ <javadoc> enumClassJavadoc </javadoc> ]
</typesafeEnumClass>
```

- `name` must be a legal Java Identifier, and must not have a package prefix.
- `<javadoc>` customizes the Javadoc tool annotations for the enumeration class.
- You can have zero or more `<typesafeEnumMember>` declarations embedded in a `<typesafeEnumClass>` declaration.

The syntax for the `<typesafeEnumMember>` customization is:

```
<typesafeEnumMember name = "enumMemberName">
            [ value = "enumMemberValue" ]
  [ <javadoc> enumMemberJavadoc </javadoc> ]
</typesafeEnumMember>
```

- `name` must always be specified and must be a legal Java identifier.
- `value` must be the enumeration value specified in the source schema.
- `<javadoc>` customizes the Javadoc tool annotations for the enumeration constant.

For inline annotations, the `<typesafeEnumClass>` declaration must be specified in the annotation element of the `<simpleType>` element. The `<typesafeEnum-Member>` must be specified in the annotation element of the enumeration member. This allows the enumeration member to be customized independently from the enumeration class.

For information about typesafe enum design patterns, see the sample chapter of Joshua Bloch's *Effective Java Programming* on the Java Developer Connection.

<javadoc> Binding Declarations

The `<javadoc>` declaration lets you add custom Javadoc tool annotations to schema-derived JAXB packages, classes, interfaces, methods, and fields. Note that `<javadoc>` declarations cannot be applied globally—that is, they are only valid as a sub-elements of other binding customizations.

The syntax for the `<javadoc>` customization is:

```
<javadoc>
  Contents in &lt;b>Javadoc&lt;\b> format.
</javadoc>
```

or

```
<javadoc>
  <<![CDATA[
  Contents in <b>Javadoc<\b> format
  ]]>
</javadoc>
```

Note that documentation strings in `<javadoc>` declarations applied at the package level must contain `<body>` open and close tags; for example:

```
<jxb:package name="primer.myPo">
       <jxb:javadoc><![CDATA[<body>Package level documentation
for generated package primer.myPo.</body>]]>
</jxb:javadoc>
      </jxb:package>
```

Customization Namespace Prefix

All standard JAXB binding declarations must be preceded by a namespace prefix that maps to the JAXB namespace URI (`http://java.sun.com/xml/ns/jaxb`). For example, in this sample, `jxb:` is used. To this end, any schema you want to customize with standard JAXB binding declarations *must* include the JAXB namespace declaration and JAXB version number at the top of the schema file. For example, in `po.xsd` for 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 62).

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 `<`*`JWSDP_HOME`*`>/jaxb/samples/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 <*JWSDP_HOME*>/jaxb/samples/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 78), 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 78) specifies the name of the schema to which this binding declarations file will apply, and the schema node at which the customizations will first take effect. Subsequent binding declarations in this file will reference specific nodes within the schema, but this first declaration should encompass the schema as a whole; for example, in `bindings.xjb`:

```
<jxb:bindings schemaLocation="po.xsd" node="/xs:schema">
```

Global and Schema Binding Declarations

The global schema binding declarations in `bindings.xjb` are the same as those in `po.xsd` for the Datatype Converter example. The only difference is that because the declarations in `po.xsd` are made inline, you need to embed them in `<xs:appinfo>` elements, which are in turn embedded in `<xs:annotation>` elements. Embedding declarations in this way is unnecessary in the external bindings file.

```
<jxb:globalBindings
    fixedAttributeAsConstantProperty="true"
    collectionType="java.util.Vector"
    typesafeEnumBase="xs:NCName"
    choiceContentProperty="false"
    typesafeEnumMemberName="generateError"
    bindingStyle="elementBinding"
    enableFailFastCheck="false"
    generateIsSetMethod="false"
    underscoreBinding="asCharInWord"/>
<jxb:schemaBindings>
    <jxb:package name="primer.myPo">
        <jxb:javadoc><![CDATA[<body>Package level
documentation for generated package primer.myPo.</body>]]>
        </jxb:javadoc>
    </jxb:package>
    <jxb:nameXmlTransform>
        <jxb:elementName suffix="Element"/>
    </jxb:nameXmlTransform>
</jxb:schemaBindings>
```

By comparison, the syntax used in `po.xsd` for the Datatype Converter example is:

```
<xsd:annotation>
  <xsd:appinfo>
    <jxb:globalBindings
        .
        <binding_declarations>
        .
    <jxb:schemaBindings>
        .
        <binding_declarations>
        .
    </jxb:schemaBindings>
  </xsd:appinfo>
</xsd:annotation>
```

Class Declarations

The class-level binding declarations in `bindings.xjb` differ from the analogous declarations in `po.xsd` for the Datatype Converter example in two ways:

- As with all other binding declarations in bindings.xjb, you do not need to embed your customizations in schema `<xsd:appinfo>` elements.
- You must specify the schema node to which the customization will be applied. The general syntax for this type of declaration is:

  `<jxb:bindings node="//<node_type>[@name='<node_name>']">`

For example, the following code shows binding declarations for the `complexType` named `USAddress`.

```
<jxb:bindings node="//xs:complexType[@name='USAddress']">
  <jxb:class>
    <jxb:javadoc>
<![CDATA[First line of documentation for a <b>USAddress</b>.]]>
    </jxb:javadoc>
  </jxb:class>

  <jxb:bindings node=".//xs:element[@name='name']">
    <jxb:property name="toName"/>
  </jxb:bindings>

  <jxb:bindings node=".//xs:element[@name='zip']">
```

```
      <jxb:property name="zipCode"/>
    </jxb:bindings>
  </jxb:bindings>
  <!-- node="//xs:complexType[@name='USAddress']" -->
```

Note in this example that `USAddress` is the parent of the child elements `name` and `zip`, and therefore a `</jxb:bindings>` tag encloses the `bindings` declarations for the child elements as well as the class-level `javadoc` declaration.

Fix Collides Example

The Fix Collides example illustrates how to resolve name conflicts—that is, places in which a declaration in a source schema uses the same name as another declaration in that schema (namespace collisions), or places in which a declaration uses a name that does translate by default to a legal Java name.

Note: Many name collisions can occur because XSD Part 1 introduces six unique symbol spaces based on type, while Java only has only one. There is a symbols space for type definitions, elements, attributes, and group definitions. As a result, a valid XML schema can use the exact same name for both a type definition and a global element declaration.

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

Like the External Customize example, the Fix Collides example uses an external binding declarations file, `binding.xjb`, to define the JAXB binding customizations.

- The example.xsd Schema
- Looking at the Conflicts
- Output From ant fail
- The binding.xjb Declarations File
- Resolving the Conflicts in example.xsd

The example.xsd Schema

The XML schema, `<JWSDP_HOME>/jaxb/samples/fix-collides /example.xsd`, used in the Fix Collides example illustrates common name conflicts encountered when attempting to bind XML names to unique Java identifiers in a Java package. The schema declarations that result in name conflicts are highlighted in bold below.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
           jxb:version="1.0">

  <xs:element name="Class" type="xs:int"/>
  <xs:element name="FooBar" type="FooBar"/>
  <xs:complexType name="FooBar">
    <xs:sequence>
      <xs:element name="foo" type="xs:int"/>
      <xs:element ref="Class"/>
      <xs:element name="zip" type="xs:integer"/>
    </xs:sequence>
    <xs:attribute name="zip" type="xs:string"/>
  </xs:complexType>
</xs:schema>
```

Looking at the Conflicts

The first conflict in `example.xsd` is the declaration of the `element` name `Class`:

```
<xs:element name="Class" type="xs:int"/>
```

`Class` is a reserved word in Java, and while it is legal in the XML schema language, it cannot be used as a name for a schema-derived class generated by JAXB.

When this schema is run against the JAXB binding compiler with the `ant fail` command, the following error message is returned:

```
[xjc] [ERROR] Attempt to create a property having the same
name as the reserved word "Class".
[xjc] line 6 of example.xsd
```

The second conflict is that there are an `element` and a `complexType` that both use the name `Foobar`:

```
<xs:element name=”FooBar” type=”FooBar”/>
<xs:complexType name=”FooBar”>
```

In this case, the error messages returned are:

```
[xjc] [ERROR] A property with the same name "Zip" is
generated from more than one schema component.
[xjc] line 22 of example.xsd
[xjc] [ERROR] (Relevant to above error) another one is
generated from this schema component.
[xjc] line 20 of example.xsd
```

The third conflict is that there are an `element` and an `attribute` both named `zip`:

```
<xs:element name=”zip” type=”xs:integer”/>
<xs:attribute name=”zip” type=”xs:string”/>
```

The error messages returned here are:

```
[xjc] [ERROR] A property with the same name "Zip" is
generated from more than one schema component.
[xjc] line 22 of example.xsd
[xjc] [ERROR] (Relevant to above error) another one is
generated from this schema component.
[xjc] line 20 of example.xsd
```

Output From ant fail

Here is the complete output returned by running `ant  fail` in the `<`*`JWSDP_HOME`*`>/jaxb/samples/fix-collides` directory:

```
[echo] Compiling the schema w/o external binding file
(name collision errors expected)...
[xjc] Compiling file:/C:/jwsdp-1.4/jaxb/samples/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 <*JWSDP_HOME*>/jaxb/samples/fix-collides/binding.xjb binding declarations file resolves the conflicts in examples.xsd by means of several customizations.

Resolving the Conflicts in example.xsd

The first conflict in example.xsd, using the Java reserved name Class for an element name, is resolved in binding.xjb with the <class> and <property> declarations on the schema element node Class:

```
<jxb:bindings node="//xs:element[@name='Class']">
  <jxb:class name="Clazz"/>
  <jxb:property name="Clazz"/>
</jxb:bindings>
```

The second conflict in example.xsd, the namespace collision between the element FooBar and the complexType FooBar, is resolved in binding.xjb by using a <nameXmlTransform> declaration at the <schemaBindings> level to append the suffix Element to all element definitions.

This customization handles the case where there are many name conflicts due to systemic collisions between two symbol spaces, usually named type definitions and global element declarations. By appending a suffix or prefix to every Java identifier representing a specific XML symbol space, this single customization resolves all name collisions:

```
<jxb:schemaBindings>
  <jxb:package name="example"/>
    <jxb:nameXmlTransform>
      <jxb:elementName suffix="Element"/>
    </jxb:nameXmlTransform>
</jxb:schemaBindings>
```

The third conflict in `example.xsd`, the namespace collision between the `element zip` and the `attribute zip`, is resolved in `binding.xjb` by mapping the `attribute zip` to property named `zipAttribute`:

```
<jxb:bindings node=".//xs:attribute[@name='zip']">
  <jxb:property name="zipAttribute"/>
</jxb:bindings>
```

Running `ant` in the `<`*`JWSDP_HOME`*`>/jaxb/samples/fix-collides` directory will pass the customizations in `binding.xjb` to the `xjc` binding compiler, which will then resolve the conflicts in `example.xsd` in the schema-derived Java classes.

Bind Choice Example

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

The schema declarations in `<`*`JWSDP_HOME`*`>/jaxb/samples/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 *<JWSDP_HOME>*/jaxb/samples/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`.

3 XML and Web Services Security

THIS addendum discusses using XML and Web Services Security (XWS-Security) for *message-level security*. In message-level security, security information is contained within the SOAP message, which allows security information to travel along with the message. For example, a portion of the message may be signed by a sender and encrypted for a particular receiver. When the message is sent from the initial sender, it may pass through intermediate nodes before reaching its intended receiver. In this scenario, the encrypted portions continue to be opaque to any intermediate nodes and can only be decrypted by the intended receiver. For this reason, message-level security is also sometimes referred to as *end-to-end security*.

This release includes the following XWS-Security features:

- Support for securing JAX-RPC applications.
- A sample security framework within which a JAX-RPC application developer will be able to secure applications by signing/verifying parts of SOAP messages and/or encrypting/decrypting parts of a SOAP message.

  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.

- Sample programs that demonstrate using the framework and Web Services Security (WSS) interop scenarios.
- Command-line tools that provide specialized utilities for keystore management, including `pkcs12import` and `keyexport`.

The XWS-Security release contents are arranged in the structure shown in Table 3–1 within the Java WSDP release:

Table 3–1 XWS-Security directory structure

Directory Name	**Contents**
`<`*`JWSDP_HOME`*`>/xws-security/etc/`	Keystore files used for the examples.
`<`*`JWSDP_HOME`*`>/xws-security/docs/`	Release documentation for the XWS-Security framework.
`<`*`JWSDP_HOME`*`>/xws-security/lib/`	JAR files used by the XWS-Security framework.
`<`*`JWSDP_HOME`*`>/xws-security/samples/`	Example code. This release includes two sample applications, `simple` and `interop`. For more information on the samples, read Understanding and Running the Simple Sample Application (page 113) and Understanding and Running the Interop Sample Application (page 121).
`<`*`JWSDP_HOME`*`>/xws-security/bin/`	Command-line tools that provide specialized utilities for keystore management. For more information on these, read Useful XWS-Security Command-Line Tools (page 128).

This implementation of XWS-Security is based on the Oasis Web Services Security (WSS) specification, which can be viewed at the following URL:

```
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-
message-security-1.0.pdf
```

Some of the material in this chapter assumes that you understand basic security concepts. To learn more about these concepts, we recommend that you explore the following resources before you begin this chapter.

- The Java 2 Standard Edition discussion of security, which can be viewed from

  `http://java.sun.com/j2se/1.4.2/docs/guide/security/`
  `index.html`
- The *J2EE 1.4 Tutorial* chapter titled *Security*, which can be viewed from

  `http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html`

Does XWS-Security Implement Any Specifications?

XWS-Security is an implementation of the Web Services Security (WSS) specification developed at OASIS. WSS defines a SOAP extension providing quality of protection through message integrity, message confidentiality, and message authentication. WSS mechanisms can be used to accommodate a wide variety of security models and encryption technologies.

The WSS specification provides an extensible mechanism for associating security tokens with SOAP messages. The WSS specification itself does not define the format of the various types of security token. Instead, a series of security token profile documents have either been published or are in the process of being published. Each of these documents provides a specification for one type of security token. In addition to XML formatted security tokens, WSS also defines a mechanism for encoding binary security tokens so that they can be used too. Some of the token types either in use or under development for WSS include User Name Token, X.509, Kerberos, and SAML. It is common practice to use public-key cryptography to encrypt a random secret key which is used to do the actual encryption of the payload and often the response message as well. WSS provides a mechanism for including these encrypted session keys.

The WSS specification defines an end to end security framework that provides support for intermediary security processing. Message integrity is provided by using XML Signature in conjunction with security tokens to ensure that messages are transmitted without modifications. The integrity mechanisms can support multiple signatures, possibly by multiple actors (however, the configuration file schema does not provide support for intermediary security processing). The techniques are extensible such that they can support additional signature formats. Message confidentiality is granted by using XML Encryption in conjunction with security tokens to keep portions of SOAP messages confidential. The encryption mechanisms can support operations by multiple actors.

In this release, the XWS-Security framework provides the following options for securing JAX-RPC applications:

- XML Digital Signature (DSig)

  This implementation of XML and Web Services Security uses Apache's XML-DSig implementation, which is based on the XML Signature specification, which can be viewed at `http://www.w3.org/TR/xmldsig-core/`.

  XML digital signatures are designed for use in XML transactions. It is a standard that was jointly developed by W3C and the IETF (RFC 2807, RFC 3275). The standard defines a schema for capturing the result of a digital signature operation applied to arbitrary data and its processing. XML signatures add authentication, data integrity, and support for non-repudiation to the signed data.

  XML Signature has the ability to sign only specific portions of the XML tree rather than the complete document. This is important when a single XML document may need to be signed multiple times by a single or multiple parties. This flexibility can ensure the integrity of certain portions of an XML document, while leaving open the possibility for other portions of the document to change. Signature validation mandates that the data object that was signed be accessible to the party that is interested in the transaction. The XML signature will generally indicate the location of the original signed object.

  Samples containing code for signing and/or verifying parts of the SOAP message are included with this release in the directory `<`*`JWSDP_HOME`*`>/xws-security/samples/`. Read Understanding and Running the Simple Sample Application (page 113) and Understanding and Running the Interop Sample Application (page 121) for more information on these sample applications.

- XML Encryption (XML-Enc)

  This implementation of XML and Web Services Security uses Apache's XML-Enc implementation, which is based on the XML Encryption W3C standard. This standard can be viewed at `http://www.w3.org/TR/xmlenc-core/`.

XML Encryption specifies a process for encrypting data and representing the result in XML. The data may be arbitrary data (including an XML document), an XML element, or XML element content. The result of encrypting data is an XML Encryption element which contains or references the cipher data.

Samples containing code for encrypting and/or decrypting parts of the SOAP message are included with this release in the directory `<JWSDP_HOME>/xws-security/samples/simple/`. Read Understanding and Running the Simple Sample Application (page 113) for more information on these sample applications.

- Username Token Verification

  Username token verification specifies a process for sending `UserName` tokens along with the message. The receiver can validate the identity of the sender by validating the digital signature sent by the sender. A digital signature internally refers to a security token (for example, `UserName` token or an X509 Certificate Token) to indicate the key used for signing. Sending these tokens with a message binds the identity of the tokens (and any other claims occurring in the security token) to the message.

  This implementation of XML and Web Services Security provides support for Username Token Profile, which is based on OASIS WSS Username Token Profile 1.0 (which can be read at `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf`) and X509 Certificate Token Profile, which is based on OASIS WSS X509 Certificate Token Profile 1.0 (which can be read at `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0.pdf`).

  Samples containing code for sending user name and X509 certificate tokens along with the SOAP message are included with this release in the directory `<JWSDP_HOME>/xws-security/samples/simple/`. Read Understanding and Running the Simple Sample Application (page 113) for more information on these sample applications.

- Implement Interoperable Web Services Security applications

  This implementation of XML and Web Services Security fully supports the implementation of WSS Interop scenarios. Developers can use the XWS-Security framework to implement applications that have security requirements similar to those defined in the WSS interop scenarios.

The WSS interop scenarios define concrete scenarios for the purpose of an interop demonstration. Some of the scenarios have been purposely oversimplified, and serve just to bootstrap interop. In other words, the scenarios have not been selected primarily for their utility, but more to prove some basic interop of the methodology. The scenarios should be interpreted more as test plans to demonstrate basic features of the protocol than as descriptions of scenarios for which there must be solutions.

The following are some of the interoperability scenarios documents that are supported by this implementation:

- The full set of the Oasis Open Technical Committee documents are available to Oasis TC members only

  `http://www.oasis-open.org/apps/org/workgroup/wss/documents.php`.

- Draft Spec for Interop1 (draft 5)

  `http://lists.oasis-open.org/archives/wss/200306/msg00002.html`

- Final Spec for Interop2 (draft 6)

  `http://lists.oasis-open.org/archives/wss/200310/msg00003.html`

Samples containing implementations of all the WSS Interop Scenarios are included in this release in the directory `<JWSDP_HOME>/xws-security/samples/interop/`. Read Understanding and Running the Interop Sample Application (page 121) for more information on these sample applications.

More information on the Web Services Architecture can be found at:

`http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/wsa.pdf`

On Which Technologies Is XWS-Security Based?

XWS-Security APIs are used for securing Web services based on JAX-RPC. This release of XWS-Security is based on non-standard XML Digital Signature and XML Encryption APIs, which are subject to change with new revisions of the technology. As standards are defined in the Web Services Security space, these nonstandard APIs will be replaced with standards-based APIs.

JSR-105 (XML Digital Signature APIs) are included in this release of the Java WSDP as well. JSR 105 is a standard API (in progress, almost at Proposed Final Draft) for generating and validating XML Signatures as specified by the W3C recommendation. It 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 (which is the goal for a future release) and by non-Web Services technologies, for example, documents stored or transferred in XML. Both JSR-105 and JSR-106 (XML Digital Encryption APIs) are core-XML security components.

XWS-Security does not use the JSR-105 or JSR-106 APIs because, currently, the Java standards for XML Digital Signatures and XML Encryption are undergoing definition under the Java Community Process. These Java standards are JSR-105-XML Digital Signature APIs, which you can read at `http://www.jcp.org/en/jsr/detail?id=105` and JSR-106-XML Digital Encryption APIs, which you can read at `http://www.jcp.org/en/jsr/detail?id=106`.

XWS-Security uses the Apache libraries for DSig and XML-Enc. In future releases, the goal of XWS-Security is to move toward using JSR-105 and JSR-106 APIs.

Table 3–2 shows how the various technologies are stacked upon one another:

Table 3–2 API/Implementation Stack Diagram

XWS-Security
JSR-105 & JSR-106 (possible in future release)
Apache XML Security implementation (current implementation, however this can easily be replaced or swapped, because the JSRs are provider-based)
J2SE Security (JCE/JCA APIs)

The *Apache XML Security* project is aimed at providing implementation of security standards for XML. Currently the focus is on the W3C standards. More information on Apache XML Security can be viewed at:

```
http://xml.apache.org/security/
```

Java security includes the *Java Cryptography Extension (JCE)* and the *Java Cryptography Architecture (JCA)*. JCE and JCA form the foundation for public key technologies in the Java platform. The JCA API specification can be viewed at `http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html`. The JCE documentation can be viewed at `http://java.sun.com/products/jce/index-14.html`.

What is the XWS-Security Framework?

You use the XWS-Security framework to secure JAX-RPC applications. This means that you use XWS-Security to secure SOAP messages (requests and responses) through signing some parts, or encrypting some parts, or sending username-password authentication info, or some combination of these. Some example applications that use the technology are discussed in Are There Any Sample Applications Demonstrating XWS-Security? (page 105).

You use the XWS-Security framework to secure JAX-RPC applications by using the `-security` option of the `wscompile` tool. When you create an `asant` (or Ant) target for JAX-RPC clients and services, you use the `wscompile` utility to generate stubs, ties, serializers, and WSDL files. XWS-Security has been integrated into JAX-RPC through the use of security configuration files. The code for performing the security operations on the client and server is generated by supplying the security configuration files to the JAX-RPC `wscompile` tool. The `wscompile` tool can be instructed to generate security code by making use of the `-security` option and supplying the security configuration file. See Configuring Security Configuration Files (page 96) for more information on creating and using security configuration files.

To use the XWS-Security framework, you need to set up the client and server-side infrastructure. A critical component of setting up your system for XWS-Security is to set up the appropriate database for the type of security (DSig, XML-Enc, User Name Token) you are planning to use. Depending on the structure of your application, these databases could be any combination of keystore files, truststore files, and username-password files. More information on setting up the infrastructure is described in Setting Up the Application Server For the Examples (page 109).

Configuring Security Configuration Files

XWS-Security makes it simple to specify client and server-side configurations describing security settings using security configuration files. In this tutorial, build, package, and deploy targets are defined and run using the `asant` tool. The `asant` tool is version of the Apache Ant Java-based build tool used specifically with the Sun Java System Application Server Platform Edition 8 (Application Server). If you are deploying to a different container, you may want to use the Apache Ant tool instead.

To configure a security configuration file, follow these steps:

1. Create a security configuration file. Creating security configuration files are discussed in more detail in Understanding Security Configuration Files (page 97). Sample security configuration files are located in the directory `<JWSDP_HOME>/xws-security/samples/simple/config/`.
2. Create an `asant` (or Ant) target in the `build.xml` file for your application that passes in and uses the security configuration file(s). This step is discussed in more detail in How Do I Specify the Security Configuration for the Build Files? (page 103).
3. Create a property in the `build.properties` file to be passed to the `build.xml` targets that points to the security configuration file. This step is discussed in more detail in How Do I Specify the Security Configuration for the Build Files? (page 103).

Understanding Security Configuration Files

Security configuration files are written in XML. The elements within the XML file that specify the security mechanism(s) to use for an application are enclosed within `<xwss:SecurityConfiguration></xwss:SecurityConfiguration>` tags. A complete set of options for values that can be placed within these elements are listed in Useful XWS-Security Command-Line Tools (page 128). This section describes a few of these options.

The first set of elements of the security configuration file contain the declaration that this file is a security configuration file. The elements that provide this declaration look like this:

```
<xwss:SecurityConfiguration
   xmlns:xwss="http://com.sun.xml.wss.configuration">
</xwss:SecurityConfiguration>
```

Within these declaration elements are elements that specify which type of security mechanism is to be applied to the SOAP message. For example, to apply XML Digital Signature, the security configuration file would include an `xwss:Sign` element, along with a keystore alias that identifies the private key/

certificate associated with the sender's signature. A simple security configuration file that uses digital signatures would look like this:

```
<xwss:SecurityConfiguration
   xmlns:xwss="http://com.sun.xml.wss.configuration">
      <xwss:Sign senderCertificateAlias="keyEntryAlias"/>
</xwss:SecurityConfiguration>
```

The xwss elements can be listed sequentially so that more than one security mechanism can be applied to the SOAP message. For example, to first sign a message and then encrypt it, you would create an xwss element with the value Sign (to do the signing first), and then create an xwss element with the value of Encrypt (to encrypt after the signing). Building on the previous example, to add encryption to the message after the message has been signed, the security configuration file would be written like this example:

```
<xwss:SecurityConfiguration
   xmlns:xwss="http://com.sun.xml.wss.configuration">
      <xwss:Sign senderCertificateAlias="s1as"/>
      <xwss:Encrypt
         receiverCertificateAlias="trustedCertAlias"/>
</xwss:SecurityConfiguration>
```

Another type of security mechanism that can be specified in the security configuration file is *user name authentication*. In the case of user name authentication, the user name and password of a client need to be authenticated against the user/password database of the server. The xwss element first specifies that the security mechanism to use is Authenticate. A second xwss element is nested within the Authenticate elements to specify which type of authentication to use (currently only UsernameAndPassword is supported). On the server-side, refer to the documentation for your server regarding how to set up a user/password database for the server, or read Setting Up The User Name/Password Database For the Application Server (page 111) for a summary. A client-side security configuration file that specifies UsernameAndPassword authentication would look like this:

```
<xwss:SecurityConfiguration
   xmlns:xwss="http://com.sun.xml.wss.configuration">
      <xwss:Authenticate>
         <xwss:UsernameAndPassword
            name="Ron"
            password="noR"/>
      </xwss:Authenticate>
</xwss:SecurityConfiguration>
```

The `simple` sample application includes a number of example security configuration files. The sample configuration files are located in the directory `<JWSDP_HOME>/xws-security/samples/simple/config/`. Further discussion of the example security configurations can be found in Sample Security Configuration File Options (page 115).

XWS-Security Configuration File Schema

When creating a security configuration file, there is a hierarchy within which the XML elements must be listed. This section contains a sketch of the schema for the data for security configuration files.

Note: In the following schema, a **?** character before an attribute or subelement indicates that zero or one instances of this attribute or subelement are allowed. A * character before an attribute or subelement indicates that zero or more instances of this attribute or subelement are allowed.

Table 3–3 shows the schema diagram on the left, and a description of the schema on the right.

Table 3–3 Security Configuration File Schema Elements and Description

Security Configuration File Schema Elements	Description
`SecurityConfiguration`	This element is the top-level XML element that indicates that what follows is a security configuration file.
`[attributes]`	
`?useTimestamps`	If `useTimestamps` is set to `true`, a timestamp is sent with the message. If `useTimestamps` is not specified (default) or is set to `false`, a `Timestamp` is not sent with the message. A timestamp says which particular point in time it marks. You may also want to consider using a nonce, which is a value that you should never receive more than once.
`?dumpMessages`	If `dumpMessages` is set to `true`, all incoming and outgoing messages are printed at the standard output. If `dumpMessages` is not specified (default) or is set to `false`, no incoming or outgoing messages are printed.

Table 3–3 Security Configuration File Schema Elements and Description (Continued)

Security Configuration File Schema Elements	**Description**
[subelements]	These subelements can appear in any order. The order in which they appear determines the order in which they are executed.
***Encrypt**	Use the `Encrypt` element to request that an encryption operation be performed.
[attributes]	
`?target`	The `target` attribute is a string representation of the `QName` of the element to be targeted for encryption. For example, if the element has a local name `Name` and a namespace URI `some-uri`, the target value is `{some-uri}Name`. The default is the `QName` of the SOAP Body.
`?encryptContentOnly`	If `encryptContentOnly` is set to `true`, only the contents of the target element are encrypted. The default is `true`.
`?receiverCertificateAlias`	The truststore alias corresponding to the certificate of the receiver of the message. This attribute is required when `Encrypt` is used.
`?keyReferenceType`	The reference mechanism of the certificate used for encryption that is to be used in preparing the message to be sent. This reference mechanism helps the receiver locate the same certificate at his end. The certificate leads the receiver to its private key. The receiver uses this private key to decrypt the message. Valid values for `keyReferenceType` are `Direct`, `Identifier`, and `IssuerSerialNumber`. If `Direct` is used, the receiver certificate is sent along with the message. `Direct` is the default value. If `Identifier` is used, the Base64-encoded subject key identifier extension value of the certificate is sent in the message. If `IssuerSerialNumber` is used, the issuer name and serial number of the certificate used are sent in the message.
***Sign**	Use the `Sign` element to request that a digital signature operation be performed.
[attributes]	

Table 3–3 Security Configuration File Schema Elements and Description (Continued)

Security Configuration File Schema Elements	**Description**
?target	The target attribute is a string representation of the QName of the element to be targeted for signature. For example, if the element has a local name Name and a namespace URI some-uri, the target value is {some-uri}Name. The default is the QName of the SOAP Body.
?senderCertificateAlias	The keystore alias corresponding to the certificate/private-key pair of the sender used for signing. If there is a single key entry in the keystore, this attribute is not required. If there is more than one key entry in the keystore, this attribute is required.
?keyReferenceType	The reference mechanism of the certificate corresponding to the private-key used for the signature that is to be used in preparing the message to be sent. This reference mechanism helps the receiver locate the same certificate at his end. The retrieved certificate helps the receiver verify the signed message. Valid values for keyReferenceType are Direct, Identifier, and IssuerSerialNumber. If Direct is used, the receiver certificate is sent along with the message. Direct is the default value. If Identifier is used, the Base64-encoded subject key identifier extension value of the certificate is sent in the message. If IssuerSerialNumber is used, the issuer name and serial number of the certificate used are sent in the message.
?signToken	If signToken is set to true, the security token (certificate data) associated with the signature is also signed in the same message. The default value is false.
***Authenticate**	Use the Authenticate element when the sender wants to be authenticated at the receiver end.
[subelements]	
?UsernameAndPassword	Use the UsernameAndPassword element when a UsernameToken should be sent with the message. This UsernameToken would contain the sender's user and password information. This subelement is mandatory when Authenticate is used.
[attributes]	

Table 3–3 Security Configuration File Schema Elements and Description (Continued)

Security Configuration File Schema Elements	**Description**
`?name`	The name of the user. The `name` attribute is mandatory when `UsernameAndPassword` is used.
`?password`	The password of the user. The `password` attribute is mandatory when `UsernameAndPassword` is used.
`?useNonce`	If `useNonce` is set to `true`, a nonce value is sent with the username token. A nonce is different from a timestamp because a timestamp says which particular point in time it marks. A nonce is a value that you should never receive more than once. WSS uses nonces to protect services such that servers can detect a replay of a previously used message authenticator (at the same service). Sending a nonce value is an effective counter-measure against replay attacks. The default value is `true`. For more information about a nonce, read `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf`.
`?digestPassword`	If `digestPassword` is set to `true`, the password is transported in digest form. The default value is `true`.
`[subelements]`	
`?Encrypt`	Use the `Encrypt` subelement of `UsernameAndPassword` if the `UsernameToken` needs to be encrypted. If the `Encrypt` subelement is used, the target of encryption is the `UsernameToken`. The complete token, not only the contents, is encrypted.
`?receiverCertificateAlias`	The truststore alias corresponding to the certificate of the receiver of the message. This attribute is required when `Encrypt` is used.

Table 3–3 Security Configuration File Schema Elements and Description (Continued)

Security Configuration File Schema Elements	Description
?keyReferenceType	The reference mechanism of the certificate used for encryption that is to be used in preparing the message to be sent. This reference mechanism helps the receiver locate the same certificate at his end. The certificate leads the receiver to its private key. The receiver uses this private key to decrypt the message. Valid values for `keyReferenceType` are `Direct`, `Identifier`, and `IssuerSerialNumber`. If `Direct` is used, the receiver certificate is sent along with the message. `Direct` is the default value. If `Identifier` is used, the Base64-encoded subject key identifier extension value of the certificate is sent in the message. If `IssuerSerialNumber` is used, the issuer name and serial number of the certificate used are sent in the message.

How Do I Specify the Security Configuration for the Build Files?

After the security configuration files are created, you can easily specify which of the security configuration files to use for your application. In the `build.properties` file for your application, create a property to specify which security configuration file to use for the client, and which security configuration file to use for the server. An example from the `simple` sample application does this using the following properties:

```
# Uncomment *1* Client Security Config. file
#client.security.config=config/dump-client.xml
#client.security.config=config/sign-client.xml
client.security.config=config/encrypt-client.xml
#client.security.config=config/sign-encrypt-client.xml
#client.security.config=config/encrypt-sign-client.xml

# Uncomment *1* Server Security Config. file
#server.security.config=config/dump-server.xml
#server.security.config=config/sign-server.xml
server.security.config=config/encrypt-server.xml
#server.security.config=config/sign-encrypt-server.xml
#server.security.config=config/encrypt-sign-server.xml
```

As you can see from this example, several security scenarios are listed in the `build.properties` file. To run a particular security configuration option, simply uncomment the security configuration you want to run, and comment all of the other options.

After the property has been defined in the `build.properties` file, you can refer to it from the file that contains the `asant` (or Ant) targets, which is `build.xml`.

When you create an `asant` (or Ant) target for JAX-RPC clients and services, you use the `wscompile` utility to generate stubs, ties, serializers, and WSDL files. XWS-Security has been integrated into JAX-RPC through the use of security configuration files. The code for performing the security operations on the client and server is generated by supplying the configuration files to the JAX-RPC `wscompile` tool. The `wscompile` tool can be instructed to generate security code by making use of the `-security` option and supplying the security configuration file. An example of the target that runs the `wscompile` utility with the `-security` option pointing to the security configuration file specified in the `build.properties` file to generate server artifacts, from the `simple` sample application, looks like this:

```
<target name="gen-server" depends="prepare"
   description="Runs wscompile to generate server artifacts">
   <echo message="Running wscompile...."/>
   <wscompile verbose="${jaxrpc.tool.verbose}"
      xPrintStackTrace="true"
      keep="true" fork="true"
      security="${server.security.config}"
      import="true"
      model="${build.home}/server/WEB-INF${model.rpcenc.file}"
      base="${build.home}/server/WEB-INF/classes"
      classpath="${app.classpath}"
      config="${config.rpcenc.file}">
      <classpath>
         <pathelement location="${build.home}/server/WEB-INF/
classes"/>
         <path refid="app.classpath"/>
      </classpath>
   </wscompile>
</target>
```

An example of the target that runs the `wscompile` utility with the `security` option pointing to the security configuration file specified in the `build.proper-`

`ties` file to generate the client-side artifacts, from the `simple` sample application, looks like this:

```
<target name="gen-client" depends="prepare"
   description="Runs wscompile to generate client side
   artifacts">
   <echo message="Running wscompile...."/>
   <wscompile fork="true" verbose="${jaxrpc.tool.verbose}"
      keep="true"
      client="true"
      security="${client.security.config}"
      base="${build.home}/client"
      features=""
      config="${client.config.rpcenc.file}">
      <classpath>
         <path refid="app.classpath"/>
      </classpath>
   </wscompile>
</target>
```

Refer to the documentation for the `wscompile` utility in Useful XWS-Security Command-Line Tools (page 128) for more information on `wscompile` options.

Are There Any Sample Applications Demonstrating XWS-Security?

This release of the Java WSDP includes two example applications that illustrate how a JAX-RPC developer can use the XML and Web Services Security framework. The example applications can be found in the `<`*`JWSDP_HOME`*`>/xws-security/samples/<`*`sample_name`*`>/` directory. Before you can run the sample applications, you must follow the setup instructions in Setting Up To Use XWS-Security With the Sample Applications (page 107).

The sample applications print out both the client and server request and response SOAP messages. The output from the server may be viewed in the appropriate container's log file. The output from the client may be viewed using `stdout`.

In these examples, the server-side code is found in the `<`*`JWSDP_HOME`*`>/xws-security/samples/<`*`sample_name`*`>/server/src/<`*`sample_name`*`>/` directory. Client-side code is found in the `<`*`JWSDP_HOME`*`>/xws-security/samples/<`*`sample_name`*`>/client/src/<`*`sample_name`*`>/` directory. The `asant` (or Ant) targets build objects under the `/build/server/` and `/build/client/` directories.

This example can be deployed onto any of the following containers. For the purposes of this tutorial, only deployment to the Sun Java System Application Server Platform Edition 8 will be shown. The `README.txt` file for each example provides more information on deploying to the other containers.

- Sun Java System Application Server PE 8 (Application Server)
  `http://wwws.sun.com/software/products/appsrvr/home_appsrvr.html`
- Sun Java System Web Server 6.1 (Web Server)
  `http://wwws.sun.com/software/products/web_srvr/home_web_srvr.html`
- Tomcat 5 Container for Java WSDP (Tomcat)
  `http://jakarta.apache.org/tomcat/`

This example uses keystore and truststore files that are included in the `<`*`JWSDP_HOME`*`>/xws-security/etc/` directory. The container on which you choose to deploy your applications must be configured to recognize the keystore and truststore files. For more information on using keystore and truststore files, read the `keytool` documentation at `http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/keytool.html`. For more information on how to configure the Application Server to recognize these files, refer to Setting Up the Application Server For the Examples (page 109), or to the application's `README.txt` file if deploying on the Web Server or Tomcat.

The following sample applications are included:

- `simple`
  This sample application lets you plug in different client and server-side configurations describing security settings. This example has support for digital signatures, XML encryption/decryption, and user name token authentication. This example allows and demonstrates combinations of these basic security mechanisms through configuration files. See Understanding and Running the Simple Sample Application (page 113), for more information on this example.
- `interop`

  This sample application demonstrates a complete implementation of the seven WSS Interop Scenarios. More information on these scenarios can be found at `http://lists.oasis-open.org/archives/wss/200306/msg00002.html`. Although this sample makes use of the programmatic APIs of the XWS-Security framework, all of these scenarios (excluding Scenario 4) can be implemented using the security configuration file framework demonstrated in the `simple` sample. See Understanding and Running the Interop Sample Application (page 121), for more information on this example.

Setting Up To Use XWS-Security With the Sample Applications

This addendum discusses creating and running applications that use the XWS-Security framework, and deploying these applications onto the Sun Java System Application Server Platform Edition 8. For deployment onto other containers, read the `README.txt` file for the example applications for more information.

Follow these steps to set up your system to create, run, and deploy the sample applications included in this release that use the XWS-Security framework.

1. Make sure that you are running the Java WSDP 1.4 on the Java 2 Platform, Standard Edition version 1.4.2. If not, you can download the JDK from: `http://java.sun.com/j2se/1.4.2/`.
2. Setting System Properties (page 107)
3. Configuring a JCE Provider (page 108)
4. Setting Up the Application Server For the Examples (page 109)

Setting System Properties

The `asant` (or Ant) build files for the XWS-Security samples shipped with this release rely on certain environment variables being set correctly. Make sure that the following environment variables are set to the locations specified in this list. If you are not sure how to set these environment variables, refer to the file `<`*`JWSDP_HOME`*`>/xws-security/docs/samples.html` for more specific information.

1. Set `JAVA_HOME` to the location of your J2SE installation directory, for example, `/home/<`*`your_name`*`>/j2sdk1.4.2_04/`.
2. Set `JWSDP_HOME` to the location of your Java WSDP 1.4 installation directory, for example, `/home/<`*`your_name`*`>/jwsdp-1.4/`.
3. Set `SJSAS_HOME` to the location of your Application Server installation directory, for example, `/home/<`*`your_name`*`>/SUNWappserver/`. If you are

deploying onto a different container, set `SJSWS_HOME` or `TOMCAT_HOME` instead.

4. Set `ANT_HOME` to the location where the `asant` (or `ant`) executable can be found. If you are running on the Application Server, this will be `<SJSAS_HOME>/bin/`. If you are running on a different container, this location will probably be `<JWSDP_HOME>/apache-ant/bin/`.
5. Set the `PATH` variable so that it contains these directories: `<JWSDP_HOME>/jwsdp-shared/bin/`, `<SJSAS_HOME>/bin/`, `<ANT_HOME>/`, and `<JAVA_HOME>/bin/`.

Configuring a JCE Provider

The Java Cryptography Extension (JCE) provider included with J2SE 1.4.x does not support RSA encryption. Because the XWS-Security sample applications use RSA encryption, you must download and install a JCE provider that does support RSA encryption in order for these sample applications to run.

Follow these steps to add a JCE provider statically as part of your JDK environment:

1. Download and install a JCE provider JAR (Java ARchive) file. The following URL provides a list of JCE providers that support RSA encryption:

   `http://java.sun.com/products/jce/jce14_providers.html`
2. Copy the JCE provider JAR file to `<JAVA_HOME>/jre/lib/ext/`.
3. Stop the Application Server (or other container). If the Application Server is not stopped, and restarted later in this process, the JCE provider will not be recognized by the Application Server.
4. Edit the `<JAVA_HOME>/jre/lib/security/java.security` properties file in any text editor. Add the JCE provider you've just downloaded to this file. The `java.security` file contains detailed instructions for adding this provider. Basically, you need to add a line of the following format in a location with similar properties:

   ```
   security.provider.<n>=<provider class name>
   ```

   In this example, *<n>* is the order of preference to be used by the Application Server when evaluating security providers. Set *<n>* to `2` for the JCE provider you've just added. Make sure that the Sun security provider remains at the highest preference, with a value of 1.

   ```
   security.provider.1=sun.security.provider.Sun
   ```

Adjust the levels of the other security providers downward so that there is only one security provider at each level.

5. Restart the Application Server (or other container). To save time with stopping and restarting the server, you can complete the steps in Setting Up the Application Server For the Examples (page 109) before restarting the Application Server.

Setting Up the Application Server For the Examples

To set up the container for running the XWS-Security sample applications included with this release, you need to specify on which container you are running the `asant` (or Ant) build targets, and you must point the container to the keystore and truststore files to be used to run the XWS-Security sample applications. For the sample applications, these are the keystore and truststore files included in the `/xws-security/etc/` directory. For further discussion of using keystores and truststores with XWS-Security applications, read Keystore and Truststore Files with XWS-Security (page 111).

This tutorial describes deployment to the Application Server. For information on setting up other containers, refer to the `README.txt` file located in the top-level directory for each sample application.

To set up the Application Server for running the XWS-Security examples, follow these steps:

1. Open the file `/xws-security/samples/simple/build.xml`. Locate the `prepare` target. Verify that this target is configured to use the appropriate container for your system. If the target is not configured correctly for your system, replace the existing value with the appropriate one. The choices are:

   `a.&sjsas;`
   `b.&sjsws;`
   `c.&tomcat;`

2. Stop the Application Server.
3. Copy the file <*SJSAS_HOME*>`/domains/domain1/config/domain.xml` to <*SJSAS_HOME*>`/domains/domain1/config/domain.xml.default`.
4. Open the file <*SJSAS_HOME*>`/domains/domain1/config/domain.xml` in a text editor.

5. Locate and modify the following JVM options. Currently, the `keyStore` and `trustStore` options point to default keystore and truststore files for the Application Server and need to be changed to point to the keystore and truststore files specific to this example.

```
<jvm-options>-Djavax.net.ssl.keyStore=
   ${com.sun.aas.instanceRoot}/../../xws-security/etc/
   server-keystore.jks</jvm-options>
<jvm-options>-Djavax.net.ssl.trustStore=
   ${com.sun.aas.instanceRoot}/../../xws-security/etc/
   server-truststore.jks</jvm-options>
```

Note: There is a convention in SSL that keystore and key password(s) should be the same. But for the sake of completeness, the Application Server's security environment looks for the system property `com.sun.xml.wss.KeyPassword`. If this property is explicitly set, the Application Server assumes that this property contains the password for all the keys of the keystore.

---

6. In order to run sample applications that demonstrate user name/password-based authentication, you must configure the user name database of the server. To configure the user name database for the Web Server or Tomcat, refer to the `README.txt` file for the `simple` sample application. For the Application Server, follow these steps:
   a. Create a text file, for example, `xws-security-users.xml`, in the directory `<`*`SJSAS_HOME`*`>/domains/domain1/config/` to hold user names and passwords.
   b. Enter the following text:

```
<?xml version='1.0' encoding='utf-8'?>
<xws-security-users>
   <user username="Ron" password="noR"/>
   <user username="Vishal" password="changeit"/>
</xws-security-users>
```

   c. Add the following JVM option to `<`*`SJSAS_HOME`*`>/domains/domain1/config/domain.xml`:

```
<jvm-options>-Dcom.sun.xml.wss.usersFile={location of
   xws-security-users.xml file}</jvm-options>
```

7. Restart the Application Server.

You may want to change the `keyStore` and `trustStore` properties for your server back to their default values before working through other sample applications in the *Java Web Services Developer Pack 1. 4 Tutorial*. You can easily do this by renaming the file you saved as *<SJSAS_HOME>*`/domains/domain1/config/domain.xml.default` to its original name of *<SJSAS_HOME>*`/domains/domain1/config/domain.xml` after you have completed working through these examples.

Keystore and Truststore Files with XWS-Security

The default keystore shipped with the Application Server cannot always be used with XWS-Security because the certificate in the default keystore is of version 1 (v1), and some XWS-Security options work only with certificates of version 3 (v3). Version 3 includes requirements specified by X509 Token Profile.

XWS-Security can work with v1 certificates in some situations. For example, if the sender sends an Issuer/Serial pair to identify the key, and does not send the certificate itself, the receiver can locate and use an identified v1 certificate, assuming the receiver has access to it locally.

To plug in your own keystores and truststores for an application, make sure that the certificates are of version 3, and that the client truststore contains the certificate of the certificate authority that issued the server's certificate, and vice versa.

Setting Up The User Name/Password Database For the Application Server

The configurations that demonstrate username-password-based authentication require that the username database of the server be configured. To configure the username database for the Application Server, follow these steps. Steps for the other application servers are discussed in the `README.txt` file for the `simple` sample application.

1. Stop the Application Server.
2. Create a username-password file, for example, *<HOME>*`/xws-security-users.xml`.
3. Enter the following information (or similar) into the file:

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

```
      <user username="Ron" password="noR"/>
      <user username="Vishal" password="changeit"/>
   </xws-security-users>
```

4. Open the file <*SJSAS_HOME*>`/domains/domain1/config/domain.xml` and add the following `jvm` option:

   ```
   <jvm-options>
   -Dcom.sun.xml.wss.usersFile=<location of xws-security-users.xml file>
   </jvm-options>
   ```

5. Restart the Application Server.

Setting Build Properties

To run the sample applications, you must edit the sample `build.properties` file for that sample application and specify information that is unique to your system and to your installation of Java WSDP 1.4 and the Application Server (or other container).

To edit the `build.properties` file for the example you want to run, follow these steps:

1. Change to directory for the sample application you want to run: <*JWSDP_HOME*>`/xws-security/samples/simple/` or <*JWSDP_HOME*>`/xws-security/samples/interop/`.
2. Copy the `build.properties.sample` file to `build.properties`.
3. Edit the `build.properties` file, checking that the following properties are set correctly for your system:
   - `javahome`: Set this to the directory where J2SE 1.4.2 is installed.
   - `sjsas.home`: If you are running under the Application Server, set this to the directory where the Application Server is installed and make sure there is not a comment symbol (#) to the left of this entry. If you are running under a different container, set the location for its install directory under the appropriate property name (`tomcat.home` or `sjsws.home`) and uncomment that entry instead. Only one of the container home properties should be uncommented at any one time.
   - `username`, `password`: Enter the appropriate username and password values for a user assigned to the role of `admin` for the container instance being used for this sample. A user with this role is authorized to deploy applications onto the Application Server.

- `endpoint.host`, `endpoint.port`: If you changed the default host and/or port during installation of the Application Server (or other container), change these properties to the correct values for your host and port. If you installed the Application Server using the default values, these properties will already be set to the correct values.
- `VS.DIR`=If you are running under the Sun Java System Web Server, enter the directory for the virtual server. If you are running under any other container, you do not need to modify this property.
- `jwsdp.home`: Set this property to the directory where Java WSDP is installed. The keystore and truststore URL's for the client are configured relative to this property.
- `http.proxyHost`, `http.proxyPort`: If you are using remote endpoints, set these properties to the correct proxy server address and port. If you are not using remote endpoints, put a comment character (`#`) before these properties. A proxy server will follow the format of `myserver.mycompany.com`. The proxy port is the port on which the proxy host is running, for example, `8080`.

If you specify a proxy host and port, you also need to add the following lines to specify the proxy information to the targets that run the client applications in the `build.xml` file for the application.

```
<sysproperty key="http.proxyHost"
   value="${http.proxyHost}"/>
<sysproperty key="http.proxyPort"
   value="${http.proxyPort}"/>
```

4. Save and exit the `build.properties` file.

Understanding and Running the Simple Sample Application

This example is a fully-developed sample application that demonstrates various configurations that can be used to exercise XWS-Security framework code. By modifying one property in the `build.properties` file for the example, you can change the type of security that is being used for the client and/or the server. The types of security configurations possible in this example include XML Digital Signature, XML Encryption, and UserName-Token verification. This example allows and demonstrates combinations of these basic security mechanisms through the specification of the appropriate security configuration files.

The application prints out both the client and server request and response SOAP messages. The output from the server may be viewed in the appropriate container's log file. The output from the client may be viewed using `stdout`.

In this example, server-side code is found in the `/simple/server/src/simple/` directory. Client-side code is found in the `/simple/client/src/simple/` directory. The `asant` (or Ant) targets build objects under the `/build/server/` and `/build/client/` directories. You can view other useful `asant` (or Ant) targets by entering `asant (or` ant) at the command line in the `/simple/` directory.

This example uses keystores and truststores which are included in the `/xws-security/etc/` directory. For more information on using keystore and truststore files, read the `keytool` documentation at the following URL:

```
http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/key-
tool.html
```

Plugging in Security Configurations

This example makes it simple to plug in different client and server-side configurations describing security settings. This example has support for digital signatures, XML encryption/decryption, and username/token verification. This example allows and demonstrates combinations of these basic security mechanisms through configuration files. See Sample Security Configuration File Options (page 115), for further description of the security configuration options defined for the `simple` sample application.

To specify which security configuration option to use when the sample application is run (see Running the Simple Sample Application, page 120), follow these steps:

1. Open the `build.properties` file for the example. This file is located at `<`*`JWSDP_HOME`*`>/xws-security/samples/simple/build.properties`.
2. To set the security configuration that you want to run for the client, locate the `client.security.config` property, and enter one of the client security configuration options contained in the `<`*`JWSDP_HOME`*`>/xws-security/samples/simple/config/` directory. The client configuration options are listed in Sample Security Configuration File Options (page 115). This section also lists which client and server configurations work together. For example, if you want to use XML Encryption for the client, you would set this property as follows:

```
# Client Security Config. file
client.security.config=config/encrypt-client.xml
```

3. To set the security configuration that you want to run for the server, locate the `server.security.config` property, and enter one of the server security configuration options listed in the `<JWSDP_HOME>/xws-security/samples/simple/config/` directory. The server configuration options, and which server options are valid for a given client configuration, are listed in Sample Security Configuration File Options (page 115). For example, if you want to use XML Encryption for the server, you would set this property as follows:

   ```
   # Server Security Config. file
   server.security.config=config/encrypt-server.xml
   ```
4. Save and exit the `build.properties` file.
5. Run the sample application as described in Running the Simple Sample Application (page 120).

Sample Security Configuration File Options

The configuration files available for this example are located in the `/xws-security/samples/simple/config/` directory. The configuration pairs available under this sample include configurations for both the client and server side. Any combination of the `*-client.xml` and `*-server.xml` configurations can be used. Some possible combinations are discussed in more detail in the referenced sections.

- `dump-client.xml`, `dump-server.xml`
- `encrypt-client.xml`, `encrypt-server.xml`
- `sign-client.xml`, `sign-server.xml`
- `sign-encrypt-client.xml`, `sign-encrypt-server.xml`
- `encrypt-sign-client.xml`, `encrypt-sign-server.xml`
- `user-pass-authenticate-client.xml, dump-server.xml`
- `encrypted-user-pass-also-client.xml, dump-server.xml`
- `sign-ticket-also-client.xml, dump-server.xml`

Dumping the Request and/or the Response

The security configuration pair `dump-client.xml` and `dump-server.xml` have no security operations. These options enable the following tasks:

- Dump the request before it leaves the client.
- Dump the response upon receipt from the server.

The container's server logs also contain the dumps of the server request and response. See Running the Simple Sample Application (page 120) for more information on viewing the server logs.

Encrypting the Request and/or the Response

The security configuration pair `encrypt-client.xml` and `encrypt-server.xml` enable the following tasks:

- Client encrypts the request body and sends it.
- Server decrypts the request and sends back an encrypted response.
- Client decrypts the response.

The `encrypt-client.xml` file looks like this:

```
<xwss:SecurityConfiguration xmlns:xwss=
   "http://com.sun.xml.wss.configuration"
      useTimestamps="true"
      dumpMessages="true">
   <xwss:Encrypt receiverCertificateAlias="s1as"/>
</xwss:SecurityConfiguration>
```

Signing and Verifying the Signature

The security configuration pair `sign-client.xml` and `sign-server.xml` enable the following tasks:

- Client signs the request body.
- Server verifies the signature and signs its response.
- Client verifies the signature over the body.

The `sign-client.xml` file looks like this:

```
<xwss:SecurityConfiguration xmlns:xwss=
   "http://com.sun.xml.wss.configuration"
      dumpMessages="true">
   <xwss:Sign
      senderCertificateAlias="xws-security-client"/>
</xwss:SecurityConfiguration>
```

Signing then Encrypting the Request, Decrypting then Verifying the Signature

The security configuration pair `sign-encrypt-client.xml` and `sign-encrypt-server.xml` enable the following tasks:

- Client signs and then encrypts and sends the request body.
- Server decrypts and verifies the signature.
- Server signs and then encrypts and sends the response.

The `sign-encrypt-client.xml` file looks like this:

```
<xwss:SecurityConfiguration xmlns:xwss=
   "http://com.sun.xml.wss.configuration"
   useTimestamps="true"
   dumpMessages="true">
   <xwss:Sign/>
   <xwss:Encrypt keyReferenceType="Identifier"
      receiverCertificateAlias="s1as"/>
</xwss:SecurityConfiguration>
```

Encrypting then Signing the Request, Verifying then Decrypting the Signature

The security configuration pair `encrypt-sign-client.xml` and `encrypt-sign-server.xml` enable the following tasks:

- Client encrypts the request body, then signs and sends it.
- Server verifies the signature and then decrypts the request body.
- Server encrypts and then signs its response.

The encrypt-sign-client.xml file looks like this:

```
<xwss:SecurityConfiguration xmlns:xwss=
   "http://com.sun.xml.wss.configuration"
      useTimestamps="true"
      dumpMessages="true">
   <xwss:Encrypt keyReferenceType="IssuerSerialNumber"
      receiverCertificateAlias="s1as"/>
   <xwss:Sign/>
</xwss:SecurityConfiguration>
```

Adding a UserName Password Token

The security configuration pair user-pass-authenticate-client.xml and dump-server.xml enable the following tasks:

- Client adds a username-password token and sends a request.
- Server authenticates the username and password against a username-password database.
- Server dumps the response upon receipt.

The username-password database must be set up before this security configuration pair will run properly. Refer to Setting Up the Application Server For the Examples (page 109) for instructions on setting up this database.

The user-pass-authenticate-client.xml file looks like this:

```
<xwss:SecurityConfiguration xmlns:xwss=
   "http://com.sun.xml.wss.configuration"
      dumpMessages="true">
   <xwss:Authenticate>
      <xwss:UsernameAndPassword
         name="Ron"
         password="noR"
      />
   </xwss:Authenticate>
</xwss:SecurityConfiguration>
```

Adding a UserName Password Token, then Encrypting the UserName Token

The security configuration pair `encrypted-user-pass-also-client.xml` and `dump-server.xml` enable the following tasks:

- Client adds a `UsernamePassword` token.
- Client encrypts the `UsernamePassword` token before sending the request.
- Server decrypts the `UsernamePassword` token.
- Server authenticates the user name and password against a username-password database.
- Server dumps the response upon receipt.

The user name-password database must be set up before this security configuration pair will run properly. Refer to Setting Up the Application Server For the Examples (page 109) for instructions on setting up this database.

The `encrypted-user-pass-also-client.xml` file looks like this:

```
<xwss:SecurityConfiguration xmlns:xwss=
   "http://com.sun.xml.wss.configuration"
      dumpMessages="true">
   <xwss:Authenticate>
      <xwss:UsernameAndPassword name="Ron" password="noR">
         <xwss:Encrypt
            keyReferenceType="Identifier"
            receiverCertificateAlias="s1as"/>
      </xwss:UsernameAndPassword>
   </xwss:Authenticate>
</xwss:SecurityConfiguration>
```

Signing the Ticket Element

The security configuration pair `sign-ticket-also-client.xml` and `dump-server.xml` enable the following tasks:

- Client signs the ticket element present in the message body and then signs the message body itself.
- Client sends request.
- Server verifies these signatures.
- Server dumps the response upon receipt.

The `sign-ticket-also-client.xml` file looks like this:

```
<xwss:SecurityConfiguration xmlns:xwss=
   "http://com.sun.xml.wss.configuration"
      useTimestamps="true"
      dumpMessages="true">
   <xwss:Sign target="{http://xmlsoap.org/Ping}ticket"
      keyReferenceType="Identifier"/>
   <xwss:Sign/>
</xwss:SecurityConfiguration>
```

Running the Simple Sample Application

Before the sample application will run correctly, you must have completed the tasks defined in the following sections of this addendum:

- Setting System Properties (page 107)
- Configuring a JCE Provider (page 108)
- Setting Up the Application Server For the Examples (page 109)
- Setting Build Properties (page 112)

To run the `simple` sample application, follow these steps:

1. Start the selected container and make sure the server is running. To start the Application Server,
   a. From a Unix machine, enter the following command from a terminal window: `asadmin start-domain domain1`
   b. From a Windows machine, choose Start→Programs→Sun Microsystems→J2EE 1.4 SDK→Start Default Server.
2. Modify the `build.properties` file to set up the security configuration that you want to run for the client and/or server. See Sample Security Configuration File Options (page 115) for more information on the security configurations options that are already defined for the sample application.
3. Build and run the application from a terminal window or command prompt.
   - On the Application Server, the command to build and run the application is: `asant run-sample`
   - On the other containers, the command to build and run the application is: `ant run-sample`

Note: To run the sample against a remote server containing the deployed endpoint, use the `run-remote-sample` target in place of the `run-sample` target. In this situation, make sure that the `endpoint.host`, `endpoint.port`, `http.proxyHost`, `http.proxyPort`, and `service.url` properties are set correctly in the `build.properties` file (as discussed in Setting Build Properties, page 112) before running the sample.

If the application runs successfully, you will see a message similar to the following:

```
[echo] Running the client program....
[java] ==== Sending Message Start ====
...
[java] ==== Sending Message End ====
[java] ==== Received Message Start ====
...
[java] ==== Received Message End ====
[java] Hello to Duke!
```

You can view similar messages in the server logs:

```
<SJSAS_HOME>/domains/<domain-name>/logs/server.log
<TOMCAT_HOME>/logs/launcher.server.log
<SJSWS_HOME>/<Virtual-Server-Dir>/logs/errors
```

Understanding and Running the Interop Sample Application

This example is a fully-developed sample application that demonstrates various configurations that can be used to exercise XWS-Security framework code. The types of security configurations possible in this example include Digital Signature, XML Encryption, and UserName-Token verification. This example makes use of the XWS-Security framework to implement all WSS interop scenarios.

The interop sample application demonstrates a complete implementation of the seven WSS interop scenarios, as defined in the WSS Interop Scenarios document, which can be found at the following URL:

```
http://lists.oasis-open.org/archives/wss/200306/msg00002.html
```

The application prints out both the client and server request and response SOAP messages. The output from the server may be viewed in the appropriate container's log file. The output from the client may be viewed using `stdout`.

In this example, server-side code is found in the `/interop/server/src/interop/` directory. Client-side code is found in the `/interop/client/src/interop/` directory. The `asant` (or Ant) targets build objects under the `/build/server/` and `/build/client/` directories. You can view other useful `asant` (or Ant) targets by entering `asant` (or ant) at the command line in the `/interop/` directory.

This example uses keystores and truststores which are included in the `/xws-security/etc/` directory. These are the only keystore and truststore files that will work with this example.

For this example, the keystore and truststore required on the server side have been packaged into the WAR (Web ARchive) file itself. It is therefore sufficient to ensure that the client sets the correct locations to the `client-keystore.jks` and `client-truststore.jks` in the `build.properties` file in order to run this sample.

Web Services Security Scenarios

This section discusses the `interop` sample application, which implements the seven existing WSS interop scenarios. Using these scenarios, developers will be able to send and receive messages compliant with the WSS Soap Message Security specification. Developers can use the framework to implement applications that have security requirements similar to those defined in the WSS interop scenarios.

Table 3–4 lists each of the WSS Interop Scenarios and provides a brief description of the security configuration used for each.

Table 3–4 WSS Interop Scenarios

Scenario	**Description**
Scenario 1	The Request header contains a user name and password. The Response does not contain a security header. More information on Scenario 1 can be found at `http://lists.oasis-open.org/archives/wss/200306/msg00002.html`, *Draft Spec for Interop1 (draft 5)*. The sample client code that demonstrates this interop scenario is found in *<JWSDP_HOME>*`/xws-security/samples/interop/client/src/interop/Scenario1Client.java`. The sample server code that demonstrates this interop scenario is found in *<JWSDP_HOME>*`/xws-security/samples/interop/server/src/interop/Scenario1Impl.java`.
Scenario 2	The Request header contains a user name and password that have been encrypted using a public key provided out-of-band. The Response does not contain a security header. More information on Scenario 2 can be found at `http://lists.oasis-open.org/archives/wss/200310/msg00003.html`, *Final Spec for Interop2 (draft 6)*. The sample client code that demonstrates this interop scenario is found in *<JWSDP_HOME>*`/xws-security/samples/interop/client/src/interop/Scenario2Client.java`. The sample server code that demonstrates this interop scenario is found in *<JWSDP_HOME>*`/xws-security/samples/interop/server/src/interop/Scenario2Impl.java`.
Scenario 3	The Request body contains data that has been signed and encrypted. The certificate used to verify the signature is provided in the header. The certificate associated with the encryption is provided out-of-band. The Response body is also signed and encrypted, reversing the roles of the key pairs identified by the certificates. The sample client code that demonstrates this interop scenario is found in *<JWSDP_HOME>*`/xws-security/samples/interop/client/src/interop/Scenario3Client.java`. The sample server code that demonstrates this interop scenario is found in *<JWSDP_HOME>*`/xws-security/samples/interop/server/src/interop/Scenario3Impl.java`.

Table 3–4 (Continued)WSS Interop Scenarios

Scenario	**Description**
Scenario 4	The Request body contains data that has been signed and encrypted. The certificate used to verify the signature is provided in the header. The symmetric key used to perform the encryption is provided out-of-band. The Response body is also signed and encrypted. The same symmetric key is used to perform the encryption. The certificate used to verify the signature is provided out-of-band. The sample client code that demonstrates this interop scenario is found in `<JWSDP_HOME>/xws-security/samples/interop/client/src/interop/Scenario4Client.java`. The sample server code that demonstrates this interop scenario is found in `<JWSDP_HOME>/xws-security/samples/interop/server/src/interop/Scenario4Impl.java`.
Scenario 5	The Request body contains data that has been signed twice. First the first element of the body is signed. The certificate used to verify this signature is provided out-of-band. Next the entire body is signed. The certificate used to verify this signature is provided in the header. The Response body is not signed or encrypted. The sample client code that demonstrates this interop scenario is found in `<JWSDP_HOME>/xws-security/samples/interop/client/src/interop/Scenario5Client.java`. The sample server code that demonstrates this interop scenario is found in `<JWSDP_HOME>/xws-security/samples/interop/server/src/interop/Scenario5Impl.java`.
Scenario 6	The Request body contains data that has been encrypted and signed. The certificate associated with the encryption is provided out-of-band. The certificate used to verify the signature is provided in the header. The Response body is also encrypted and signed, reversing the roles of the key pairs identified by the certificates. The sample client code that demonstrates this interop scenario is found in `<JWSDP_HOME>/xws-security/samples/interop/client/src/interop/Scenario6Client.java`. The sample server code that demonstrates this interop scenario is found in `<JWSDP_HOME>/xws-security/samples/interop/server/src/interop/Scenario6Impl.java`.
Scenario 7	The Request body contains data that has been signed and encrypted. The signature also protects an enclosed security token by means of the STR Dereference Transform. The certificate used to verify the signature is provided in the header. The certificate associated with the encryption is provided out-of-band. The Response body is also signed and encrypted, reversing the roles of the key pairs identified by the certificates. The sample client code that demonstrates this interop scenario is found in `<JWSDP_HOME>/xws-security/samples/interop/client/src/interop/Scenario7Client.java`. The sample server code that demonstrates this interop scenario is found in `<JWSDP_HOME>/xws-security/samples/interop/server/src/interop/Scenario7Impl.java`.

How is XWS-Security Implemented in the Interop Sample Application?

The SecurityConfigurator API adds a handler to the front of a handler chain on the server side. The handler contains an initially empty list of filters. Each of these filters can perform a specific, security-related, unit of work on a message. The SecurityConfigurator allows filters to be added to this list in order to provide security services. The interop samples all use this mechanism and, taken together, illustrate the full breadth of available functionality.

The following example code is from the Scenario4Client.java file from the interop example:

```
private static void scenario4(PingService service) throws
Exception {
  try {
    SecurityConfigurator secCfg =
      new SecurityConfigurator(service, portName);
    SecurityEnvironment secDomain =
      initializeSecurityEnvironment();
    secCfg.setSecurityEnvironment(secDomain);
    X509Certificate certificate = (X509Certificate)
      ((DefaultSecurityEnvironmentImpl)
      secDomain).getKeyStore().getCertificate(aliases[0]);
    secCfg.addRequestTimestamp();
    secCfg.addSignRequest("//SOAP-ENV:Body",certificate,
      SecurityConfigurator.DIRECT_REFERENCE_STRATEGY);
    secCfg.addFilterForOutgoingMessages(new
      ExportReferenceListFilter());
    secCfg.addFilterForOutgoingMessages(new
      EncryptElementFilter("//SOAP-ENV:Body",true,
      new KeyNameStrategy()));

    if (debug) {
      // see what the request and response look like
      secCfg.addDumpRequest().addDumpResponse();
    }

    /* Add filters for Incoming messages */

    secCfg.addFilterForIncomingMessages(new
      ProcessSecurityHeaderFilter());

    } catch (Exception e) {
            e.printStackTrace();
            throw e;
        }
    }
```

This mechanism does not handle the `mustUnderstand` portion of SOAP processing properly for encrypted headers. Therefore, because the `wscompile` utility with the
`-security` option does handle this portion of SOAP processing properly, `wscompile` is the preferred method for handling security in this release. Using the `wscompile` utility is the method described in Understanding and Running the Simple Sample Application (page 113).

Running the WSS Interop Scenario Sample Applications

Before the sample application will run correctly, you must have completed the tasks defined in the following sections of this addendum:

- Setting System Properties (page 107)
- Configuring a JCE Provider (page 108)
- Setting Up the Application Server For the Examples (page 109)
- Setting Build Properties (page 112)

To run the `interop` sample application, follow these steps:

1. Start the selected container and make sure the server is running. To start the Application Server,
   a. From a Unix machine, enter the following command from a terminal window: `asadmin start-domain domain1`
   b. From a Windows machine, choose Start→Programs→Sun Microsystems→J2EE 1.4 SDK→Start Default Server.
2. If you are using a remote proxy server, add the proxy information to the `run-client` targets in the `build.xml` file. There are seven `run-client` targets, one for each of the interop scenarios, and each of these targets must have the following lines added if you are using a proxy server.

   To modify the `run-client` targets,

   a. Open the file `/interop/build.xml` in a text editor.

b. Locate the run-client<*x*> target definitions, which follow this format:

```
<target name="run-client1" ....>
<target name="run-client2" ....>
...
<target name="run-client7" ....>
```

c. Add the following lines to specify the proxy information to each of the seven `run-client` targets. The `http.proxyHost` and `http.proxyHost` properties should have been defined as specified in Setting Build Properties (page 112).

```
<sysproperty key="http.proxyHost"
   value="${http.proxyHost}"/>
<sysproperty key="http.proxyPort"
   value="${http.proxyPort}"/>
```

3. Build and run the application from a terminal window or command prompt, for example, on the Application Server,

   ```
   asant run-all
   ```

   or on Tomcat or the Web Server,

   ```
   ant run-all
   ```

4. Build and run an individual interop scenario using the following command, where <*x*> is replaced by the number of the scenario to be run:

   ```
   asant run-client<x>
   ```

   or on a remote proxy server,

   ```
   asant interop-client<x>
   ```

   For example, to run WSS Interop scenario 1, the command would be:

   ```
   asant run-client1
   ```

Note: When the server is a remote server and the application is already deployed on the remote server, the `interop-client<x> asant` (or Ant) targets can be used instead of the `run-Client<x>` targets.

If the application runs successfully, you will see a message similar to the following:

```
[echo] Running the client program....
[java] ==== Request Start ====
...
[java] ==== Request End ====
```

```
[java] ==== Response Start ====
...
[java] ==== Response End ====
[java] Hello to Duke!
```

You can view similar messages in the server logs:

```
<SJSAS_HOME>/domains/<domain-name>/logs/server.log
<TOMCAT_HOME>/logs/launcher.server.log
<SJSWS_HOME>/<Virtual-Server-Dir>/logs/errors
```

Useful XWS-Security Command-Line Tools

In this release, the following command-line tools are included. These tools provide specialized utilities for keystore management or for specifying security configuration files.

- `pkcs12import`

  The `pkcs12import` command allows *Public-Key Cryptography Standards version 12* (PKCS-12) files (sometimes referred to as PFX files) to be imported into a keystore, typically a keystore of type *Java KeyStore* (JKS).

  When would you want to do this? One example would be a situation where you want to obtain a new certificate from a certificate authority. In this scenario, one option is to follow this sequence of steps:

  a. Generate a key-pair.

  b. Generate a certificate request

  c. Send the request to the authority for its signature

  d. Get the signed certificate and import it into his keystore.

  Another option is to let the certificate authority generate a key-pair. The authority would return a generated certificate signed by itself along with the corresponding private key. One way the certificate authority can return this information is to bundle the key and the certificate in a PKCS-12 formatted file (generally `.pfx` extension files). The information in the PKCS-12 file would be encrypted using a password that would be conveyed to the user by the authority. After receiving the PKCS-12 formatted file, you would import this key-pair (certificate/private-key pair) into your private keystore using the `pkcs12import` tool. The result of the import is that the private-key and the corresponding certificate in the PKCS-12 file are stored as a key entry inside the keystore, associated with some alias.

The `pkcs12import` tool can be found in the directory `<JWSDP_HOME>/xws-security/bin/`, and can be run from the command line by executing `pkcs12import.sh` (on Unix systems) or `pkcs12import.bat` (on Windows systems). The options for this tool listed in Table 3–5.

Table 3–5 Options for `pkcs12import` tool

Option	Description
`-file` *`pkcs12-file`*	Required. The location of the PKCS-12 file to be imported.
`[ -pass` *`pkcs12-password`* `]`	The password used to protect the PKCS-12 file. The user is prompted for this password if this option is omitted.
`[ -keystore` *`keystore-file`* `]`	Location of the keystore file into which to import the contents of the PKCS-12 file. If no value is given, defaults to `${`*`user-home`*`}/.keystore`.
`[ -storepass` *`store-password`* `]`	The password of the keystore. User is prompted for the password of the truststore if this option is omitted.
`[ -keypass` *`key-password`* `]`	The password to be used to protect the private key inside the keystore. The user is prompted for this password if this option is omitted.
`[ -alias` *`alias`* `]`	The alias to be used to store the key entry (private key and the certificate) inside the keystore.

Due to a bug in this release of the `pkcs12import` program that prevents it from working on files that do not have a private key as well as a certificate, use these steps to import PKCS#12 files that contain only certificates:

```
a. keytool -list -storepass wsi -storetype PKCS12 -keystore
   "${cert}"
      alias = alias
```

b. `keytool -export -keystore "${cert}" -alias ${alias} -storetype PKCS12 -file temp.cer`
   Enter `wsi` when prompted for the password

c. `keytool -import -alias <meaningful-alias> -keystore keystore.jks -file temp.cer`
   Enter `wsisun` when prompted for the password.

- `keyexport`

  This tool is used to export a private key in a keystore (typically of type Java Keystore (JKS)) into a file.

Note: The exported private key is not secured with a password, so it should be handled carefully. For example, you can export a private key from a keystore and use it to sign certificate requests obtained through any means using other key/certificate management tools. These certificate requests are then sent to a certificate authority for validation and certificate generation.

The `keyexport` tool can be found in the directory `<JWSDP_HOME>/xws-security/bin/`, and can be run from the command line by executing `keyexport.sh` (on Unix systems) or `keyexport.bat` (on Windows systems). The options for this tool are listed in Table 3–6.

Table 3–6 Options for `keyexport` tool

Option	**Description**
`-keyfile` *`key-file`*	Required. The location of the file to which the private key will be exported.
`[ -outform` *`output-format`* `]`	This specifies the output format. The options are DER and PEM. The DER format is the DER encoding (binary format) of the certificate. The PEM format is the base64-encoding of the DER encoding with header and footer lines added.
`[ -keystore` *`keystore-file`* `]`	Location of the keystore file containing the key. If no value is given, this option defaults to `${`*`user-home`*`}/.keystore`.
`[ -storepass` *`store-password`* `]`	Password of the keystore. User is prompted for the password if this option is omitted.

Table 3–6 (Continued)Options for `keyexport` tool

`[ -keypass` *`key-password`* `]`	The password used to protect the private key inside the keystore. User is prompted for the password if this option is omitted.
`[ -alias` *`alias`* `]`	The alias of the key entry inside the keystore.

- `wscompile`

  The `wscompile` tool generates the client stubs and server-side ties for the service definition interface that represents the Web service interface. Additionally, it generates the WSDL description of the Web service interface which is then used to generate the implementation artifacts.

  XWS-Security has been integrated into JAX-RPC through the use of security configuration files. The code for performing the security operations on the client and server is generated by supplying the configuration files to the JAX-RPC `wscompile` tool. The `wscompile` tool can be instructed to generate security code by making us of the `-security` option to specify the location of the security configuration file that contains information on how to secure the messages to be sent. An example of using the `-security` option with `wscompile` is shown in How Do I Specify the Security Configuration for the Build Files? (page 103).

  The syntax for this option is as follows:

  `wscompile [-security {`*`location of security configuration file`*`}]`

  For more description of the `wscompile` tool, its syntax, and examples of using this tool, read:
  `http://docs.sun.com/source/817-6092/hman1m/wscompile.1m.html`

Troubleshooting XWS-Security Applications

Error: at XMLCipher.getInstance (Unknown Source)

```
[java] Exception in thread "main"
java.lang.NullPointerException
[java] at
com.sun.org.apache.xml.security.encryption.XMLCipher.getInstan
ce(Unknown Source)
```

Solution: Configure a JCE provider as described in Configuring a JCE Provider (page 108).

Error: UnsupportedClassVersionError

```
java.lang.UnsupportedClassVersionError: com/sun/tools/javac/
Main (Unsupported major.minor version 49.0)
```

Solution: Install version 1.4.2 of Java 2 Standard Edition (J2SE). If you had an older version of the JDK, you will also have to reinstall the Application Server so that it recognizes this as the default version of the JDK. If you've installed version 1.5 of the JDK, you must use version 1.4.2 as the target JDK for XWS-Security.

Error: DeployTask not found

Solution: Verify that the `jwsdp.home` property in the `build.properties` file for the sample is set correctly to the location where you installed the Java WSDP version 1.4, as described in Setting Build Properties (page 112).

Further Information

- Java 2 Standard Edition, v.1.4.2 security information
  `http://java.sun.com/j2se/1.4.2/docs/guide/security/index.html`
- Java Servlet specification
  `http://java.sun.com/products/servlet/`
- Information on SSL specifications
  `http://wp.netscape.com/eng/security/`
- 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`

4

Java XML Digital Signature API

THE Java XML Digital Signature API is a standard Java API for generating and validating XML Signatures. This API is being defined under the Java Community Process as JSR 105 (see `http://jcp.org/en/jsr/detail?id=105`). This JSR is currently at Proposed Final Draft stage and this release of Java WSDP contains an early access implementation of the Proposed Final Draft 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 4–1 shows how the XML Digital Signature API (JSR 105) interacts with other security components today, including JSR 106 (XML Digital Encryption APIs), and how it will interact in future releases.

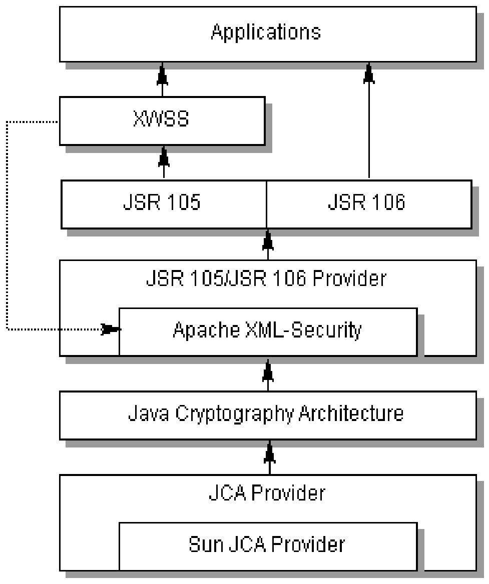

Figure 4–1 Java WSDP v1.4 Security Components

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. The JSR 105/JSR 106 layer will be standard after the two JSRs become 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 KeyInfoFactory 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 v1.4 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.

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 followingcommands after you installed Java WSDP 1.4:

For Solaris/Linux:

```
1.% export JAVA_HOME=<your J2SE installation directory>
```

2. `% export JWSDP_HOME=<`*`your Java WSDP1.4 installation directory`*`>`
3. `% export ANT_HOME=$JWSDP_HOME/apache-ant`
4. `% export PATH=$ANT_HOME/bin:$PATH`
5. `% cd $JWSDP_HOME/xmldsig/samples/<`*`sample-name`*`>`

For Windows 2000/XP:

1. `> set JAVA_HOME=<`*`your J2SE installation directory`*`>`
2. `> set JWSDP_HOME=<`*`your Java WSDP1.4 installation directory`*`>`
3. `> set ANT_HOME=%JWSDP_HOME%\apache-ant`
4. `> set PATH=%ANT_HOME%\bin;%PATH%`
5. `> 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.

If you are behind a firewall and use an HTTP proxy server, you will need to modify the `build.properties` file before you can run this example.

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 DOMValidate-

`Context` 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 146).

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 `envelo-pedSignature.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, 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, 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));
```

A
The Java WSDP Registry Server

A registry offers a mechanism for humans or software applications to advertise and discover Web services. The Java Web Services Developer Pack (Java WSDP) Registry Server implements Version 2 of the Universal Description, Discovery and Integration (UDDI) project to provide a UDDI registry for Web services in a private environment. You can use it with the Java WSDP APIs as a test registry for Web services application development.

You can use the Registry Server to test applications that you develop that use the Java API for XML Registries (JAXR). (See the JAXR chapter of the *J2EE Tutorial* for more information.) You can also use the JAXR Registry Browser sample application provided with the Java WSDP to perform queries and updates on Registry Server data; see Registry Browser (page 159) for details.

The release of the Registry Server that is part of the Java WSDP includes the following:

- A Web application, a servlet, that implements UDDI Version 2 functionality
- A database based on the native XML database Xindice, which is part of the Apache XML project. This database provides the persistent store for registry data.

The Registry Server does not support messages defined in the UDDI Version 2.0 Replication Specification.

This chapter describes how to start the Registry Server and how to use JAXR to access it. It also describes how to add and delete Registry Server users by means of a script.

Starting the Registry Server

In order to use the Java WSDP Registry Server, you must start the Application Server. Starting the Application Server automatically starts both the Registry Server and the Xindice database.

To start the Application Server on Windows, choose Sun Microsystems→J2EE 1.4 SDK→Start Default Server from the Start menu.

To start the Application Server on a UNIX system, use the following command:

```
<J2EE_HOME>/bin/asadmin start-domain domain1
```

To stop the Application Server on Windows, choose Sun Microsystems→J2EE 1.4 SDK→Stop Default Server from the Start menu.

To stop the Application Server on a UNIX system, use the following command:

```
<J2EE_HOME>/bin/asadmin stop-domain domain1
```

Changing the Port for the Registry Server

Normally you run the Application Server on port 8080. If another application uses this port, you can change the port by editing the `<J2EE_HOME>/domains/domain1/config/domain.xml` file. Open the file in a text editor and find the `http-listener` element that uses port 8080 (its `id` attribute has the value `http-listener-1`). Change this attribute to some other port value, such as 8082 or 8083:

```
port=”8082”
```

In order to run the Registry Server on a changed Application Server port, you must also edit the file `<JWSDP_HOME>/jwsdp-shared/bin/launcher.xml`. Find the following lines (they are all on one line):

```
<sysproperty key="org.apache.xindice.host"
value="desired Xindice host"/>
<sysproperty key="org.apache.xindice.port"
value="desired Xindice port"/>
```

Make the host and port the same as those for the Application Server HTTP listener. Uncomment these properties before you save the file.

Adding and Deleting Users

To add a new user to the Registry Server database, you use the script `registry-server-test.bat` (Windows) or `registry-server-test.sh` (UNIX), in the directory `<JWSDP_HOME>/registry-server/samples/`. This script uses files in the directory `<JWSDP_HOME>/registry-server/samples/xml/`. You use the same script to delete a user.

Adding a New User to the Registry

To add a new user to the Registry Server database, you use the file `UserInfo.xml` in the `xml` subdirectory. Perform the following steps:

1. Go to the directory `<JWSDP_HOME>/registry-server/samples/`.
2. Open the file `xml/UserInfo.xml` in an editor.
3. Change the values in the `<fname>`, `<lname>`, and `<uid>` tags to the first name, last name, and unique user ID (UID) of the new user. The `<uid>` tag is commonly the user's login name. It must be unique.
4. Change the value in the `<passwd>` tag to a password of your choice. This is the password for the new user. Do not modify the `<tokenExpiration>` or `<authInfo>` tag.
5. Save and close the `UserInfo.xml` file.
6. Type the following command (all on one line):

   Windows:

   ```
   registry-server-test run-cli-request
     -Drequest=xml\UserInfo.xml
   ```

UNIX:

```
registry-server-test.sh run-cli-request
  -Drequest=xml/UserInfo.xml
```

Deleting a User from the Registry

To delete a user from the registry, you use the file `UserDelete.xml` in the `xml` subdirectory.

Before you run the script this time, edit this file by modifying the values in the `<fname>`, `<lname>`, `<uid>`, and `<passwd>` tags.

To delete the user, use the following command:

Windows:

```
registry-server-test run-cli-request
  -Drequest=xml\UserDelete.xml
```

UNIX:

```
registry-server-test.sh run-cli-request
  -Drequest=xml/UserDelete.xml
```

Further Information

For more information about UDDI registries, JAXR, and Web services, see the following:

- Universal Description, Discovery, and Integration (UDDI) project:
  `http://www.uddi.org/`
- JAXR home page:
  `http://java.sun.com/xml/jaxr/`
- J2EE 1.4 Tutorial:
  `http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html`
- Java Web Services Developer Pack (Java WSDP):
  `http://java.sun.com/webservices/webservicespack.html`

- Java Technology and XML:
  `http://java.sun.com/xml/`
- Java Technology & Web Services:
  `http://java.sun.com/webservices/index.html`

B

Registry Browser

THE Registry Browser is both a working example of a JAXR client and a simple GUI tool that enables you to search registries and submit data to them. See the JAXR chapter of the *J2EE Tutorial* for more information.

The Registry Browser source code is in the directory `<JWSDP_HOME>/jaxr/samples/jaxr-browser/`. Much of the source code implements the GUI. The JAXR code is in the file `JAXRClient.java`.

The Registry Browser allows access to any registry, but includes as preset URLs the IBM and Microsoft UDDI test registries and the Registry Server (see The Java WSDP Registry Server, page 153).

Starting the Browser

To start the browser, go to the directory `<JWSDP_HOME>/jaxr/bin/` or place this directory in your path.

The following commands show how to start the browser on a UNIX system and a Microsoft Windows system, respectively:

```
jaxr-browser.sh
```

```
jaxr-browser
```

In order to access the Registry Server through the browser, you must make sure to start the Application Server before you perform any queries or submissions to the browser; see Starting the Registry Server (page 154) for details.

In order to access external registries, the browser needs to know your Web proxy settings. By default, the browser uses the settings you specified when you installed the Java WSDP. These are defined in the file `<JWSDP_HOME>/conf/jwsdp.properties`. If you want to override these settings, you can edit this file or specify proxy information on the browser command line.

To use the same proxy server for both HTTP and HTTPS access, specify a non-default proxy host and proxy port as follows. The port is usually 8080. The following command shows how to start the browser on a UNIX system:

```
jaxr-browser.sh httpHost httpPort
```

For example, if your proxy host is named `websys` and it is in the `south` subdomain, you would type

```
jaxr-browser.sh websys.south 8080
```

To use different proxy servers for HTTP and HTTPS access, specify the hosts and ports as follows. (If you do not know whether you need two different servers, specify just one. It is relatively uncommon to need two.) On a Microsoft Windows system, the syntax is as follows:

```
jaxr-browser httpHost httpPort httpsHost httpsPort
```

After the browser starts, type the URL of the registry you want to use in the Registry Location combo box, or select a URL from the drop-down menu in the combo box. The menu allows you to choose among the IBM and Microsoft registries and the default Registry Server URL:

```
http://localhost:8080/RegistryServer/
```

If you are accessing the Registry Server on a remote system, replace `localhost` with the fully qualified hostname of the system where the Registry Server is running. If Tomcat is running on a nondefault port, replace `8080` with the correct port number. You specify the same URL for both queries and updates.

There may be a delay of a few seconds while a busy cursor is visible.

When the busy cursor disappears, you have a connection to the URL. However, you do not establish a connection to the registry itself until you perform a query or update, so JAXR will not report an invalid URL until then.

The browser contains two main panes, Browse and Submissions.

Querying a Registry

You use the Browse pane to query a registry.

Note: In order to perform queries on the Microsoft registry, you must be connected to the `inquire` URL. To perform queries on the IBM registry, you may be connected to either the `inquiryapi` URL or the `publishapi` URL.

Querying by Name

To search for organizations by name, perform the following steps.

1. Click the Browse tab if it is not already selected.
2. In the Find By panel on the left side of the Registry Browser window, do the following:
   a. Select Name in the Find By combo box if it is not already selected.
   b. Type a string in the text field.
   c. Press Enter or click the Search button in the toolbar.

After a few seconds, the organizations whose names match the text string appear in the right side of the Registry Browser window. An informational dialog box appears if no matching organizations are found.

Queries are not case-sensitive. If you type a plain text string (*`string`*), organization names match if they *begin* with the text string you entered. Enclose the string in percent signs (*`%string%`*) for wildcard searches.

Double-click on an organization to show its details. An Organization dialog box appears. In this dialog box, you can click Show Services to display the Services dialog box for the organization. In the Services dialog box, you can click Show ServiceBindings to display the ServiceBindings dialog box for that service.

Querying by Classification

To query a registry by classification, perform the following steps.

1. Select Classification in the Find By combo box.
2. In the Classifications pane that appears below the combo box, double-click a classification scheme.
3. Continue to double-click until you reach the node you want to search on.
4. Click the Search button in the toolbar.

After a few seconds, one or more organizations in the chosen classification may appear in the right side of the Registry Browser window. An informational dialog box appears if no matching organizations are found.

Managing Registry Data

You use the Submissions pane to add organizations to the registry.

To go to the Submissions pane, click the Submissions tab.

Adding an Organization

To add an organization, use the Organization panel on the left side of the Submissions pane.

Use the Organization Information fields as follows:

- Name: Type the name of the organization.
- Id: You cannot type or modify data in this field; the ID value is returned by the registry when you submit the data.
- Description: Type a description of the organization.

Use the Primary Contact Information fields as follows:

- Name: Type the name of the primary contact person for the organization.
- Phone: Type the primary contact's phone number.
- Email: Type the primary contact's email address.

Note: With the Registry Server, none of these fields is required; it is possible (though not advisable) to add an organization that has no data. With the IBM and Microsoft registries, an organization must have a name.

For information on adding or removing classifications, see Adding and Removing Classifications (page 164).

Adding Services to an Organization

To add information about an organization's services, Use the Services panel on the right side of the Submissions pane.

To add a service, click the Add Services button in the toolbar. A subpanel for the service appears in the Services panel. Click the Add Services button more than once to add more services in the Services panel.

Each service subpanel has the following components:

- Name, Id, and Description fields
- Edit Bindings and Remove Service buttons
- A Classifications panel

Use these components as follows:

- Name field: Type a name for the service.
- Id field: You cannot type or modify data in this field for a level 0 JAXR provider.
- Description field: Type a description of the service.
- Click the Edit Bindings button to add service bindings for the service. An Edit ServiceBindings dialog box appears. See the next section, Adding Service Bindings to a Service, for details.
- Click the Remove Service button to remove this service from the organization. The service subpanel disappears from the Services panel.
- To add or remove classifications, use the Classifications panel. See Adding and Removing Classifications (page 164) for details.

Adding Service Bindings to a Service

To add service bindings for a service, click the Edit Bindings button in a service subpanel in the Submissions pane. The Edit ServiceBindings dialog box appears.

If there are no existing service bindings when the dialog box first appears, it contains an empty Service Bindings panel and two buttons, Add Binding and Done. If the service already has service bindings, the Service Bindings panel contains a subpanel for each service binding.

Click Add Binding to add a service binding. Click Add Binding more than once to add multiple service bindings.

After you click Add Binding, a new service binding subpanel appears. It contains three text fields and a Remove Binding button.

Use the text fields as follows:

- Description: Type a description of the service binding.
- Access URI: Type the URI used to access the service. The URI must be valid; if it is not, the submission will fail.

Use the Remove Binding button to remove the service binding from the service.

Click Done to close the dialog box when you have finished adding or removing service bindings.

Adding and Removing Classifications

To add classifications to, or remove classifications from, an organization or service, use a Classifications panel. A Classifications panel appears in an Organization panel or service subpanel.

To add a classification:

1. Click Add.
2. In the Select Classifications dialog, double-click one of the classification schemes.
   - If you clicked ntis-gov:naics:1997 or unspsc-org:unspsc:3-1, you can add the classification at any level of the taxonomy hierarchy. When you reach the level you want, click Add.
   - If you clicked uddi-org:iso-ch:3166:1999 (geography), locate the appropriate leaf node (the country) and click Add.

The classification appears in a table in the Classifications panel below the buttons.

To add multiple classifications to the organization or service, you can repeat these steps more than once. Alternatively, you can click on the classification schemes while pressing the control or shift key, then click Add.

Click Close to dismiss the window when you have finished.

To remove a classification, select the appropriate table row in the Classifications panel and click Remove. The classification disappears from the table.

Submitting the Data

When you have finished entering the data you want to add, click the Submit button in the toolbar.

An authentication dialog box appears. To continue with the submission, type your user name and password and click OK. To close the window without submitting the data, click Cancel.

If you are using the Registry Server, the default username and password are both `testuser`.

If the submission is successful, an information dialog box appears with the organization key in it. Click OK to continue. The organization key also appears in the ID field of the Submissions pane.

Note: If you submit an organization, return to the Browse pane, then return to the Submissions pane, you will find that the organization is still there. If you click the Submit button again, a new organization is created, whether or not you modify the organization data.

Deleting an Organization

To delete an organization:

1. Use the Browse pane to locate an organization you wish to delete.
2. Connect to a URL that allows you to publish data. If you were previously using a URL that only allows queries, change the URL to the publish URL.

3. Right-click on the organization and choose Delete RegistryObject from the pop-up menu.
4. In the authentication dialog box that appears, type your user name and password and click OK. To close the window without deleting the organization, click Cancel.

Stopping the Browser

To stop the Registry Browser, choose Exit from the File menu.

Index