1

Introduction to Web Services

Maydene Fisher and Eric Jendrock

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

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

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

The Role of XML and the Java™ Platform

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

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

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

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

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

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

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

What Is XML?

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

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

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

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

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

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

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

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

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

What Makes XML Portable?

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

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

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

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

Overview of the Java APIs for XML

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

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

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

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

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

JAXP

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

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

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

The SAX API

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

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

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

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

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

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

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

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

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

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

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

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

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

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

next invocation of `characters` -- prints price

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

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

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

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

The DOM API

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

XML Namespaces

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

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

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

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

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

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

The XSLT API

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

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

Transforming a DOM Tree to an XML Document

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

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

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

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

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

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

Transforming an XML Document to an HTML Document

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

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

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

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

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

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

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

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

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

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

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

JAXB

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

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

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

JAXB Binding Process

Figure 1–1 shows the JAXB data binding process.

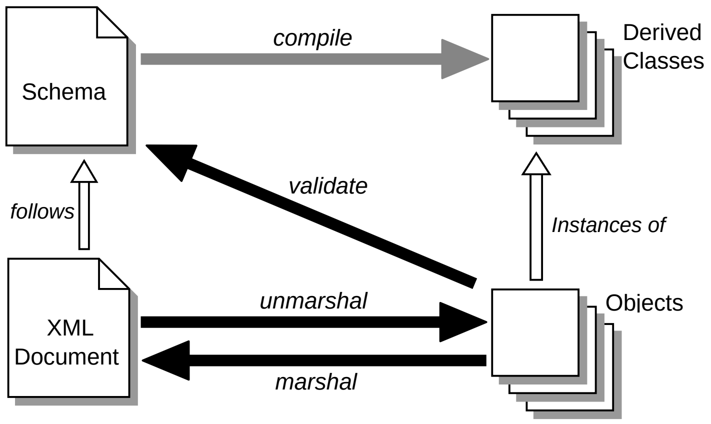

Figure 1–1 Data Binding Process

The JAXB data binding process involves the following steps:

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

Validation

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

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

Representing XML Content

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

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

Customizing JAXB Bindings

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

Example

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

Table 1–1 Schema to JAXB Bindings

XML Schema	Java Class Files

`<xsd:schema`
 	
`xmlns:xsd="http://www.w3.org/2001/XMLSchema">`	

`<xsd:element   name="purchaseOrder"`
`              type="PurchaseOrderType"/>`
	`PurchaseOrder.java`

`<xsd:element name="comment" type="xsd:string"/>`	`Comment.java`

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

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

`</xsd:schema>`	

Schema-derived Class for USAddress.java

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

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

Unmarshalling XML Content

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

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

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

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

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

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

Modifying the Content Tree

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

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

Validating the Content Tree

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

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

Marshalling XML Content

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

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

JAX-RPC

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

Overview of JAX-RPC

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

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

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

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

Interoperability

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

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

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

Ease of Use

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

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

Advanced Features

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

Using JAX-RPC

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

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

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

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

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

Creating a Web Service

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

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

```
package coffees;

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

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

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

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

```
package coffees;

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

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

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

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

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

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

Coding a Client

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

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

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

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

```
package coffees;

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

Invoking a Remote Method

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

```
java coffees.CoffeeClient
```

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

JAXM

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

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

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

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

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

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

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

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

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

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

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

Getting a Connection

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

Getting a Point-to-Point Connection

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

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

The client can use `factory` to create a `SOAPConnection` object.

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

Getting a Connection to the Messaging Provider

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

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

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

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

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

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

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

Creating a Message

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

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

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

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

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

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

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

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

Populating a Message

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

Populating the SOAP Part of a Message

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

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

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

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

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

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

soapPart.setContent(domSource);
```

Populating the Attachment Part of a Message

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

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

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

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

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

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

Sending a Message

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

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

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

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

JAXR

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

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

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

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

Using JAXR

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

Registering a Business

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

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

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

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

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

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

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

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

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

Searching a Registry

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

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

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

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

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

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

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

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

Sample Scenario

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

Scenario

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

Discovering New Distributors

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

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

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

Requesting Price Lists

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

Comparing Prices and Ordering Coffees

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

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

Selling Coffees on the Internet

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

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

Conclusion

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

Chapter 19 demonstrates a simple implementation of this scenario.