

CHAPTER 7

JAX-RPC and JAXM

The Java API for XML Messaging (JAXM) and the Java API for XML-based RPC (JAX-RPC) are both part of the Java Web Services Developer Pack, Winter 01 release.* These APIs are a key part of Sun's plans to integrate web services interfaces into future versions of the J2EE platform. JAXM provides a common set of Java APIs for creating, consuming, and exchanging SOAP envelopes over various transport mechanisms. It is intended mainly for a document-style exchange of information because it requires the use of low-level APIs to manipulate the SOAP envelope directly. JAX-RPC provides a means for performing RMI-like Remote Procedure Calls over SOAP. In addition, JAX-RPC provides rules for such things as client code generation, SOAP bindings, WSDL-to-Java and Java-to-WSDL mappings, and data mappings between Java and SOAP.

Fundamentally, JAXM supports synchronous communications. In fact, if you don't run your JAXM provider in a J2EE web container (i.e., it is implemented as a message-driven bean or servlet), then it supports only synchronous communications. You don't get asynchronous exchanges unless you use the connection provider. Don't get hung up by the "M" versus "RPC" mislabeling. You can use JAXM to exchange document- or RPC-style SOAP messages, just as you can with JAX-RPC. The real distinction between JAXM and JAX-RPC is that JAXM forces the developer to work directly with the SOAP envelope constructs, and JAX-RPC provides a high-level, WSDL-based framework that hides details of the SOAP envelope from the developer. JAX-RPC uses WSDL to generate your messages and provides an object-oriented (i.e., RMI-like) interface to the developer. JAXM doesn't use WSDL, so the developer must construct messages by hand and send or process them explicitly. You could make an analogy in terms of database access. You can access a database using

* Sun renamed the Java XML Pack to the Java Web Services Developer Pack in February 2002. The new name is confusing—the Java XML Pack still exists and remains unchanged; the Web Services Developer Pack is the XML Pack with the addition of Tomcat, Ant, and other tools. We don't know what name Sun is likely to use in the future, so be prepared for some confusion when you go to their web site.

JDBC, in which case the developer must construct SQL queries and work with the details of the database schema. Or, the developer can use JDO, which hides details of the database schema from the developer and allows the developer to work with the data as a set of Java objects.

JAXM defines the `javax.xml.soap` package, which includes the APIs for constructing and deconstructing a SOAP envelope directly, including a MIME-encoded multipart SWA (SOAP with attachments) message. Both JAXM and JAX-RPC share this package. Even if you only care about RPC, you should still go through the JAXM section to understand the SOAP Envelope APIs.

Java API for XML Messaging (JAXM)

JAXM consists of two main areas. The “messaging” capability provides a pattern for sending and receiving SOAP messages, with or without attachments. The SOAP packaging part provides APIs for constructing and deconstructing SOAP and MIME envelopes. Generally, the functionality is separated cleanly between the `javax.xml.messaging` package and `javax.xml.soap` packages.*

Where’s the Messaging?

The word “messaging” means different things to different people. For some, it refers to instant messaging or email. For others, it means reliable, asynchronous transport of critical business data, such as with Java Message Service (JMS)[†] or ebXML Message Service. In JAXM, the “M” could be any or none of those things. Like a chameleon, JAXM can take on the personality of another existing messaging protocol through the use of profiles.

Don’t infer that JAXM doesn’t support synchronous request/response interactions. JAXM can do both asynchronous, one-way communication and a synchronous request/response with the `send()` and `call()` methods, respectively. It can even do an RPC call. We will see an RPC call later when we revisit the `GetBookPrice` example using JAXM.

Simple Servlet Deployment

There’s that word again—“simple.” The bare minimum runtime requirement for JAXM is that it be deployable in a J2SE environment. This requirement means that there is no dependency on anything, except for the ability to send something over HTTP and receive it via a servlet interface, as illustrated in Figure 7-1.

* We say “generally” because of the subtleties relating to the placement of the `send()` and `call()` methods, which we will cover in a later section.

[†] For more information on JMS, please refer to *Java Message Service*, by Richard Monson-Haefel and David Chappell (O’Reilly).

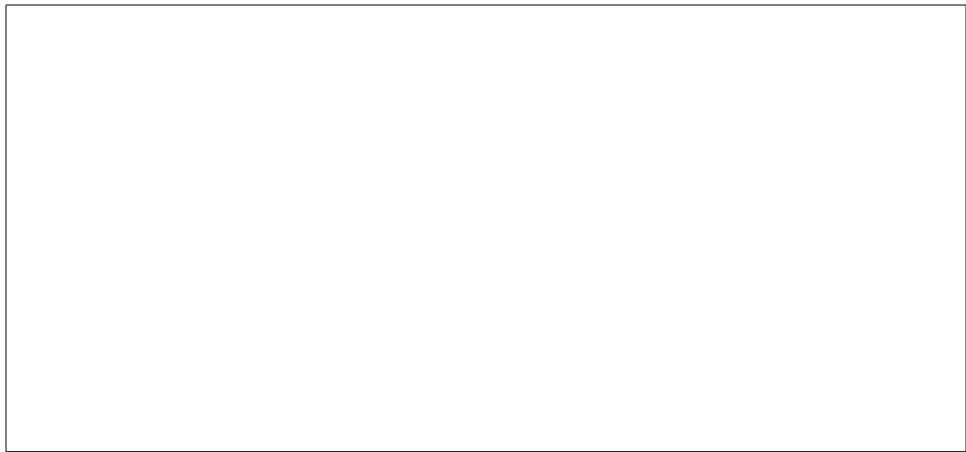


Figure 7-1. JAXM invocation in a servlet-based environment

In this type of environment, one can't rely on having a JNDI store available. Therefore, instead of performing a `lookup()` to obtain a connection, you can get a connection by calling the static `newInstance()` method on the `javax.xml.soap.SOAPConnectionFactory` object:

```
SOAPConnectionFactory scf = SOAPConnectionFactory.newInstance();
SOAPConnection connection = scf.createConnection();
```

For sending messages in this kind of environment, we use the `javax.xml.soap.SOAPConnection.call()` method. We will encounter this method again later.

Receiving the message is fairly straightforward. To receive a message, the application implements the `onMessage()` method. In a simple servlet environment, `JAXMServlet.doPost()` delegates the call to `onMessage()`. There is no concept of registering a message listener. In the provider situation, the provider may invoke the `onMessage()` method any way it likes, in accordance with its own particular delivery semantics:

```
public class ReceivingServlet extends JAXMServlet implements OnewayListener {
    ...
    public void onMessage(SOAPMessage msg) {
        System.out.println("onMessage() called in receiving servlet");
        msg.writeTo(System.out);
    }
}
```

The `onMessage()` method may return `void` or return a SOAP message, depending on whether it is intended for one-way message processing or two-way request/response. The class that implements the `onMessage()` method must extend either the `OnewayListener` or `ReqRespListener` interface to indicate its intent. Remember that you aren't allowed to overload return values; therefore, these two versions of `onMessage()` must be defined in different interfaces.

The SOAP Package

Table 7-1 shows classes and interfaces found in the `javax.xml.soap` package. These items represent the Envelope API, which is shared by both JAXM and JAX-RPC. Collectively, they provide all the functionality you need for constructing and deconstructing a SOAP or a SOAP with Attachments envelope. In Chapter 3, we used Apache SOAP, portions of the `org.w3c.dom.DocumentBuilder` interface, and portions of the JavaMail API to accomplish the same thing.

Table 7-1. The SOAP package

| Interface/class | Description |
|-----------------------|---|
| AttachmentPart | A single attachment to a SOAPMessage object |
| Detail | A container for DetailEntry objects |
| DetailEntry | The content for a Detail object, giving details for a SOAPFault object |
| MessageFactory | A factory used to create SOAPMessage objects |
| MimeHeader | An object that stores a MIME header name and its value |
| MimeHeaders | A container for MimeHeader objects, which represent the MIME headers present in a MIME part of a message |
| Name | A representation of an XML name |
| Node | A representation of a node (element) in a DOM representation of an XML document that provides tree manipulation methods |
| SOAPBody | An object that represents the contents of the SOAP body element in a SOAP message |
| SOAPBodyElement | An object that represents the contents in a SOAPBody object |
| SOAPConnection | A point-to-point connection that a client can use to send messages directly to a remote party (represented by a URL, for instance) without using a messaging provider |
| SOAPConnectionFactory | A factory used to create SOAPConnection objects |
| SOAPConstants | The definition of constants pertaining to the SOAP 1.1 protocol (e.g., <code>URI_SOAP_ACTOR_NEXT</code> and <code>URI_NS_SOAP_ENCODING</code>) |
| SOAPElement | An object representing the contents of a SOAPBody object, the contents of a SOAPHeader object, the content that can follow the SOAPBody object in a SOAPEnvelope object, or what follows the detail element in a SOAPFault object |
| SOAPElementFactory | A factory for XML fragments that eventually end up in the SOAP part |
| SOAPEnvelope | The container for the SOAPHeader and SOAPBody portions of a SOAPPart object |
| SOAPFault | An element in the SOAPBody object that contains error and/or status information |
| SOAPFaultElement | A representation of the contents in a SOAPFault object |
| SOAPHeader | A representation of the SOAP header element |
| SOAPHeaderElement | An object representing the contents in the SOAP header part of the SOAP envelope |
| SOAPMessage | The root class for all SOAP messages |
| SOAPPart | The container for the SOAP-specific portion of a SOAPMessage object |
| Text | A representation of a node whose value is text |

Let's see how these classes and interfaces fit together in some working examples.

The JAXM Sender—Request/Reply Client

This example shows how to construct a simple SOAP message and send it to a synchronous request/reply service that is expected to respond back. Let's start by running the sender and looking at the results. Run the following command in a command window:

```
java SimpleJAXMClient
```

This command produces the following output in the sender window:

```
Starting SimpleJAXMClient:
  host url      = http://localhost:8080/examples/servlet/SimpleJAXMReceive

Sending message to URL: http://localhost:8080/examples/servlet/SimpleJAXMReceive
Received reply from: http://localhost:8080/examples/servlet/SimpleJAXMReceive
Result:
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"><soap-
env:Header/><soa
p-env:Body><Response>This is the response</Response></soap-env:Body></soap-env:
Envelope>
```

In the Tomcat servlet engine window, you should see:

```
On message called in receiving servlet
There are: 0 message parts
Here's the message:
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"><soap-
env:Header/><soa
p-env:Body><Text>Some Body text</Text></soap-env:Body></soap-env:Envelope>
```

Soon we'll inspect the sending code in detail and look at the receiving code that sends the response. First, here is the sending client in its entirety:

```
import java.io.*;
import java.util.*;

public class SimpleJAXMClient {

    //Default values used if no command line parameters are set
    private static final String DEFAULT_HOST_URL =
        "http://localhost:8080/examples/servlet/SimpleJAXMReceive";
    private static final String URI = "urn:oreilly-jaws-samples";

    //Member variables
    private String m_hostURL;

    public SimpleJAXMClient(String hostURL) throws Exception
    {
        m_hostURL = hostURL;
    }
}
```

```
System.out.println();
System.out.println
    ("_____");
System.out.println("Starting SimpleJAXMClient:");
System.out.println("  host url      = " + m_hostURL);
System.out.println
    ("_____");
System.out.println();
}

public void sendJAXMMessage()
{
    try {
        javax.xml.soap.SOAPConnectionFactory scf =
            javax.xml.soap.SOAPConnectionFactory.newInstance();
        javax.xml.soap.SOAPConnection connection = scf.createConnection();

        // Get an instance of the MessageFactory class
        javax.xml.soap.MessageFactory mf =
            javax.xml.soap.MessageFactory.newInstance();

        // Create a message from the message factory. It already contains
        // a SOAP part
        javax.xml.soap.SOAPMessage message = mf.createMessage();

        // Get the message's SOAP part
        javax.xml.soap.SOAPPart soapPart = message.getSOAPPart();

        // Get the SOAP part envelope.
        javax.xml.soap.SOAPEnvelope envelope = soapPart.getEnvelope();

        // Get the Body from the SOAP envelope
        javax.xml.soap.SOAPBody body = envelope.getBody();

        // Add an element and content to the Body
        javax.xml.soap.Name name = envelope.createName("Text");
        javax.xml.soap.SOAPBodyElement bodyElement =
            body.addBodyElement (name);
        bodyElement.addTextNode ("Some Body text");

        // Send the message
        System.err.println("Sending message to URL: " + m_hostURL);

        // Synchronously send the message to the endpoint and wait for a reply
        javax.xml.soap.SOAPMessage reply =
            connection.call(message,
                new javax.xml.messaging.URLEndpoint (m_hostURL));

        System.out.println("Received reply from: " + m_hostURL);

        // Display the reply received from the endpoint
        boolean displayResult = true;
        if( displayResult ) {
```

```
        // Dump out message response.
        System.out.println("Result:");
        reply.writeTo(System.out);
    }

    connection.close();

    } catch(Throwable e) {
        e.printStackTrace();
    }
}

public static void main(String args[]) {

    ...
}
}
```

Understanding the Simple JAXM Sender

We'll start our examination of the code with the `main()` method. It's not unlike the `main()` method in the other examples we have covered in the book. This method parses incoming parameters, calls the constructor, and then calls `sendJAXMMessage()` to do the real work:

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

    . . .

    // Start the SimpleJAXMClient
    try
    {
        SimpleJAXMClient jaxmClient = new SimpleJAXMClient(hostURL);
        jaxmClient.sendJAXMMessage();
    }
    . . .
}
```

`sendJAXMMessage()` does all the interesting work; it creates and populates the SOAP envelope. First, we obtain a connection factory and use it to create a connection:

```
public void sendJAXMMessage()
{
    try {
        javax.xml.soap.SOAPConnectionFactory scf =
        javax.xml.soap.SOAPConnectionFactory.newInstance();
        javax.xml.soap.SOAPConnection connection = scf.createConnection();
```

Creating the message

Next, we obtain a message factory and create an instance of a SOAP message. The simple call to `MessageFactory.createMessage()` creates the SOAP envelope with header and body elements already in it.

Here is an example:

```
javax.xml.soap.MessageFactory mf =
    javax.xml.soap.MessageFactory.newInstance();
javax.xml.soap.SOAPMessage message = mf.createMessage();
```

If we were to look inside the SOAP message created so far, we would see that it already has the following contents:

```
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/">
<soap-env:Header/>
<soap-env:Body/>
</soap-env:Envelope>
```

These contents are accessible using the `SOAPPart`, `SOAPEnvelope`, `SOAPHeader`, and `SOAPBody` objects. In JAXM, a message is accessible using parts: either a `SOAPPart` or an `AttachmentPart`. The `SOAPPart` is the portion of the message that contains the envelope. The envelope contains the `SOAPHeader` and the `SOAPBody`:

```
// Get the message's SOAP part
javax.xml.soap.SOAPPart soapPart = message.getSOAPPart();

// Get the SOAP envelope.
javax.xml.soap.SOAPEnvelope envelope = soapPart.getEnvelope();

// Get the Body from the SOAP envelope
javax.xml.soap.SOAPBody body = envelope.getBody();
```

Adding content to the message

The pieces of the message to which we add content are `SOAPHeader` and `SOAPBody`; we can also add content indirectly by adding attachments to the message using the `AttachmentPart` object. In a later example, we will show how to add attachments. For now, we'll add some simple content to our `Body`. To do so, we must use the `addBodyElement()` method of the `Body` object. Each body element or header element must be associated with a `Name` object, which you obtain from the `SOAPEnvelope` using the `createName()` method. This method has two signatures: one takes a simple `String` argument, and the other requires a `String`, a prefix designation, and a `URI` designation. The latter approach is intended to create an element in a specific namespace.

After creating the `Name` object and using it to create a `Body` element, we add content by calling `addTextNode()` on the body element we just created:

```
// Add an element and content to the Body
javax.xml.soap.Name name = envelope.createName("Text");
javax.xml.soap.SOAPBodyElement bodyElement = body.addBodyElement (name);
bodyElement.addTextNode ("Some Body text");
```

Making the call

To execute the call, we use `SOAPConnection.call()`, passing it the message we created and a `URLEndpoint`. The `URLEndpoint` object, which inherits from `Endpoint`, specifies an absolute URL as a destination. The call blocks until a response is received:

```
// Send the message
System.err.println("Sending message to URL: " + m_hostURL);

// Synchronously send the message to the endpoint and wait for a reply
javax.xml.soap.SOAPMessage reply =
    connection.call(message,
        new javax.xml.messaging.URLEndpoint (m_hostURL));

System.out.println("Received reply from: " + m_hostURL);
```

To dump the SOAP response from the called service, we use a convenience method that JAXM provides: `writeTo()`. This method sends the raw SOAP message to the specified output stream. This method even handles attachments correctly, as we'll see later. When complete, we free resources by closing the connection explicitly:

```
// Display the reply received from the endpoint
boolean displayResult = true;
if( displayResult ) {
    // Dump out message response.
    System.out.println("Result:");
    reply.writeTo(System.out);
}
connection.close();
```

Understanding the JAXM Receiver

The JAXM Receiver used in these examples is a simple request/reply servlet. There's nothing profound here that we haven't already covered. The servlet receives the SOAP message from the sender and responds with a SOAP message. The servlet code in the following listing creates a message factory during its initialization phase:

```
import java.io.*;
import java.util.*;

public class SimpleJAXMReceive
    extends javax.xml.messaging.JAXMServlet
    implements javax.xml.messaging.ReqRespListener {

    static javax.xml.soap.MessageFactory fac = null;

    static {
        try {
            fac = javax.xml.soap.MessageFactory.newInstance();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

```

public void init(javax.servlet.ServletConfig servletConfig)
    throws javax.servlet.ServletException {
    super.init(servletConfig);
}

```

Next, `onMessage()` dumps the contents of the message to the console by using `writeTo()`; then it constructs a new message to return to the sender. Later, we will see how this same receiver and method can handle multipart messages with attachments:

```

// This is the application code for handling the message. We simply display
// the message and create and send a response.

public javax.xml.soap.SOAPMessage onMessage
    (javax.xml.soap.SOAPMessage message) {

    System.out.println("On message called in receiving servlet");
    try {

        int count = message.countAttachments();
        System.out.println("There are: " + count + " message parts");

        /// Dump the raw message out
        System.out.println("Here's the message: ");
        message.writeTo(System.out);

        /// Construct and send SOAP message response
        javax.xml.soap.SOAPMessage msg = fac.createMessage();
        javax.xml.soap.SOAPPart part = msg.getSOAPPart();
        javax.xml.soap.SOAPEnvelope env = part.getEnvelope();
        javax.xml.soap.SOAPBody body = env.getBody();
        javax.xml.soap.Name name = env.createName("Response");
        javax.xml.soap.SOAPBodyElement bodyElement =
            body.addBodyElement (name);
        bodyElement.addTextNode ("This is the response");

        return msg;

    } catch(Exception e) {
        System.out.println("Error in processing or replying to a message");
        return null;
    }
}

```

Using JAXM for SOAP with Attachments

We will now show how to modify our sending client to use the JAXM API to add attachments and headers. To see the behavior and output of this new client, execute the following command:

```
java GenericJAXMSWClient
```

This client assumes that files named *PO.xml* and *attachment.txt* are in the current directory. You should see the following output from the sending client:

```

Starting GenericJAXMSWClient:
  host url      = http://localhost:8080/examples/servlet/SimpleJAXMReceive
  data file     = PO.xml
  attachment    = Attachment.txt

```

```

Sending message to URL: http://localhost:8080/examples/servlet/SimpleJAXMReceive

Received reply from: http://localhost:8080/examples/servlet/SimpleJAXMReceive
Result:
<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"><s
oap-env:Header/><soap-env:Body><Response>This is the response</Response></soap-e
nv:Body></soap-env:Envelope>

```

The sender's output is similar to the output from the previous example. The real difference is what is seen in the Tomcat console window. The same receiver we used before generates much different results because we're now sending a multipart message. Note the MIME boundaries that separate the message's parts. The first part of the message is the SOAP envelope; the next two parts are the added attachments:

```

On message called in receiving servlet
There are: 2 attachment parts
Here's the message:
--2023334682.1010158929328.JavaMail.chappell.nbchappell13
Content-Type: text/xml

<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"><s
oap-env:Header/><soap-env:Body><PurchaseOrder><shipTo country="US"><name>Joe Smi
th</name><street>14 Oak Park</street><city>Bedford</city><state>MA</state><zip>0
1730</zip></shipTo><items><item partNum="872-AA"><productName>Candy Canes</produ
ctName><quantity>444</quantity><price>1.68</price><comment>I want candy!</commen
t></item></items></PurchaseOrder></soap-env:Body></soap-env:Envelope>
--2023334682.1010158929328.JavaMail.chappell.nbchappell13
Content-Type: text/plain

This is an attachment.
--2023334682.1010158929328.JavaMail.chappell.nbchappell13
Content-Type: text/plain; charset=ISO-8859-1

Another Part
--2023334682.1010158929328.JavaMail.chappell.nbchappell13--

```

Understanding the SwA Sender

Here's the code for the new sender, `GenericJAXMSWClient`. We will break it down and walk through the new parts in a moment.

First, however, look at the whole thing:

```
import java.io.*;
import java.util.*;

public class GenericJAXMSWClient {

    //Default values used if no command line parameters are set
    private static final String DEFAULT_DATA_FILENAME = "PO.xml";
    private static final String DEFAULT_HOST_URL
        = "http://localhost:8080/examples/servlet/SimpleJAXMReceive";
    private static final String URI = "urn:oreilly-jaws-samples";
    private static final String DEFAULT_ATTACHMENT_FILENAME
        = "Attachment.txt";

    //Member variables
    private String m_hostURL;
    private String m_dataFileName;
    private String m_attachment;

    public GenericJAXMSWClient(String hostURL, String dataFileName,
                               String attachment) throws Exception
    {
        m_hostURL = hostURL;
        m_dataFileName = dataFileName;
        m_attachment = attachment;

        System.out.println();
        System.out.println("-----");
        System.out.println("Starting GenericJAXMSWClient:");
        System.out.println("  host url      = " + m_hostURL);
        System.out.println("  data file    = " + m_dataFileName);
        System.out.println("  attachment   = " + m_attachment);
        System.out.println("-----");
        System.out.println();
    }

    public void sendJAXMMessage()
    {
        try {

            // for doing JAXP transformations
            javax.xml.transform.TransformerFactory tFact
                = javax.xml.transform.TransformerFactory.newInstance();
            javax.xml.transform.Transformer transformer
                = tFact.newTransformer();

            // Create an specific URL endpoint
            javax.xml.messaging.URLEndpoint endpoint
                = new javax.xml.messaging.URLEndpoint(m_hostURL);

            // Create a connection
            javax.xml.soap.SOAPConnectionFactory scf
                = javax.xml.soap.SOAPConnectionFactory.newInstance();
            javax.xml.soap.SOAPConnection connection = scf.createConnection();
```

```

// Get an instance of the MessageFactory class
javax.xml.soap.MessageFactory mf
    = javax.xml.soap.MessageFactory.newInstance();

// Create a message from the message factory.
// It already contains a SOAP part
javax.xml.soap.SOAPMessage message = mf.createMessage();

// Get the message's SOAP part
javax.xml.soap.SOAPPart soapPart = message.getSOAPPart();

// Get the SOAP envelope from the SOAP part of the message.
javax.xml.soap.SOAPEnvelope envelope = soapPart.getEnvelope();

// Read in the XML that will become the body in the SOAP envelope
javax.xml.parsers.DocumentBuilderFactory dbf =
    javax.xml.parsers.DocumentBuilderFactory.newInstance();
javax.xml.parsers.DocumentBuilder db = dbf.newDocumentBuilder();
org.w3c.dom.Document poDoc = db.parse(m_dataFileName);

// Get the empty SOAP envelope as a generic Source
// and put it into a DOMResult
javax.xml.transform.Source spSrc = soapPart.getContent();
javax.xml.transform.dom.DOMResult domResultEnv
    = new javax.xml.transform.dom.DOMResult();
transformer.transform(spSrc, domResultEnv);

// Now that we have the empty SOAP envelope in a DOMSource, we
// need to put it together with the DOM we just built from the
// input file.
// Get the document
org.w3c.dom.Node envelopeRoot = domResultEnv.getNode();
if (envelopeRoot.getNodeType() == org.w3c.dom.Node.DOCUMENT_NODE)
{
    // Get the root element of the document.
    org.w3c.dom.Element docEl
        = ((org.w3c.dom.Document)envelopeRoot).getDocumentElement();

    // Find the <SOAP-ENV:Body> tag using the envelope namespace
    org.w3c.dom.NodeList nList
        = docEl.getElementsByTagNameNS(
            javax.xml.soap.SOAPConstants.URI_NS_SOAP_ENVELOPE, "Body");
    if (nList.getLength() > 0)
    {
        // Found our <PurchaseOrder> element. Plug it in
        org.w3c.dom.Node bodyNode = nList.item(0);
        org.w3c.dom.Node poRoot = poDoc.getDocumentElement();

        // Import the node into this document.
        org.w3c.dom.Node importedNode
            = ((org.w3c.dom.Document)envelopeRoot).importNode(poRoot,
                true);
        bodyNode.appendChild(importedNode);
    }
}

```

```
        // Now shove it all back into the envelope.
        javax.xml.transform.dom.DOMSource domSource
            = new javax.xml.transform.dom.DOMSource(envelopeRoot);
        soapPart.setContent(domSource);
    }
}
else if (envelopeRoot.getNodeType() == org.w3c.dom.Node.ELEMENT_NODE)
    System.out.println("ElementNode");
else
    System.out.println("Unknown Node type");

// Get the Header from the SOAP envelope
javax.xml.soap.SOAPHeader header = envelope.getHeader();

// Add an element and content to the Header
javax.xml.soap.Name name
    = envelope.createName("MessageHeader",
        "jaxm", "urn:oreilly-jaws-samples");
javax.xml.soap.SOAPHeaderElement headerElement
    = header.addHeaderElement(name);

// Add an element and content to the Header
name = envelope.createName("From");
javax.xml.soap.SOAPElement childElement
    = headerElement.addChildElement(name);
childElement.addTextNode("Me");

// Add an element and content to the Header
name = envelope.createName("To");
childElement = headerElement.addChildElement(name);
childElement.addTextNode("You");

// Add additional Parts to the message
javax.activation.FileDataSource fds
    = new javax.activation.FileDataSource(m_attachment);
javax.activation.DataHandler dh
    = new javax.activation.DataHandler(fds);
javax.xml.soap.AttachmentPart ap1
    = message.createAttachmentPart(dh);
message.addAttachmentPart(ap1);

javax.xml.soap.AttachmentPart ap2
    = message.createAttachmentPart("Another Part",
        "text/plain; charset=ISO-8859-1");
message.addAttachmentPart(ap2);

// Save the changes made to the message
message.saveChanges();

System.err.println("Sending message to URL: "+ endpoint.getURL());

// Send the message to the endpoint and wait for a reply
javax.xml.soap.SOAPMessage reply
    = connection.call(message, endpoint);
```

```
System.out.println("Received reply from: " + endpoint);

// Display the reply received from the endpoint
boolean displayResult = true;

if( displayResult ) {
    // Document source, do a transform.
    System.out.println("Result:");
    javax.xml.soap.SOAPPart replyPart = reply.getSOAPPart();
    javax.xml.transform.Source src = replyPart.getContent();
    javax.xml.transform.stream.StreamResult result
        = new javax.xml.transform.stream.StreamResult( System.out );
    transformer.transform(src, result);
    System.out.println();
}
connection.close();

} catch(Throwable e) {
    e.printStackTrace();
}
}

//
// NOTE: the remainder of this deals with reading arguments
//
/** Main program entry point. */

public static void main(String args[]) {

    // Process command line, etc
    ...
    // Start the GenericJAXMSWAClient
    try
    {
        GenericJAXMSWAClient jaxmClient =
            new GenericJAXMSWAClient(hostURL, dataFileName, attachment);
        jaxmClient.sendJAXMMessage();

    }
    catch(Exception e)
    {
        System.out.println(e.getMessage());
    }
}
...
}
```

Attaching an XML fragment to the SOAP envelope

Much of the code in this chapter deals with attaching *PO.xml* to the SOAP envelope. When trying to port our Apache SOAP example from the previous chapter, we ran into a bit of a gotcha. Attaching an existing XML document to a SOAP envelope is reasonable—but you can't do it, at least not simply. This flaw is by far the biggest we

have encountered in the API. The MessageFactory creates an envelope with empty <Header> and <Body> elements. APIs exist for creating and manipulating elements individually, but nothing lets you take a whole document and attach it. A SOAPPart.setContent() method takes a document source and attaches it as the SOAP part of the message, but the document that you give it must have the envelope structure in place already. This sort of defeats the purpose. If we had corporate data that was already packaged in full SOAP envelopes, we wouldn't need an API at all, would we?

Evidence suggests that this package was created solely for the purpose of providing an API for connecting to an ebXML infrastructure. In ebXML, the body of the envelope is intended to be a manifest and the actual payload of the message is intended to be an attachment. What if you don't want to use it in that way?

Enough of the soapbox. Our solution converts both the envelope and the XML document into DOM trees, plugs them together, and assigns the whole thing back into the envelope. To help do this, we use a JAXP Transformer object. The javax.xml.transform package is an API in JAXP, which is mainly intended for transforming documents using XSLT stylesheets. We use it as a utility to convert the envelope and the XML document between a DOM tree representation and the javax.xml.transform.Stream datatype that is required by some JAXM envelope methods we will use:

```
// for doing JAXP transformations
javax.xml.transform.TransformerFactory tFact
    = javax.xml.transform.TransformerFactory.newInstance();
javax.xml.transform.Transformer transformer
    = tFact.newTransformer();
```

First, read in the XML document (*PO.xml*) from a disk and put it into a document:

```
// Read in the XML that will become the body in the SOAP envelope
javax.xml.parsers.DocumentBuilderFactory dbf =
    javax.xml.parsers.DocumentBuilderFactory.newInstance();
javax.xml.parsers.DocumentBuilder db = dbf.newDocumentBuilder();
org.w3c.dom.Document poDoc = db.parse(m_dataFileName);
```

After creating the message and getting its SOAPPart, getContent() retrieves the envelope as a generic javax.xml.transform.Source object. The Source object is the superclass of either DOMSource, SAXSource, or StreamSource. Likewise, the Result object is the superclass of DOMResult, SAXResult, and StreamResult. The transformer can take any Source object and do the right thing automatically, regardless of its subtype, and convert it to the desired Result, as shown in the following listing:

```
// Get the empty SOAP Envelope as a generic Source
// and put it into a DOMResult
javax.xml.transform.Source spSrc = soapPart.getContent();
javax.xml.transform.dom.DOMResult domResultEnv
    = new javax.xml.transform.dom.DOMResult();
transformer.transform(spSrc, domResultEnv);
```

Now that we have the envelope as a DOMResult, retrieve the Document and its root element:

```
org.w3c.dom.Node envelopeRoot = domResultEnv.getNode();
if (envelopeRoot.getNodeType() == org.w3c.dom.Node.DOCUMENT_NODE)
{
    // Get the root element of the document.
    org.w3c.dom.Element docEl
        = ((org.w3c.dom.Document)envelopeRoot).getDocumentElement();
```

Next, find the <SOAP-ENV:Body> tag using the namespace of the envelope and plug in the purchaseOrder document:

```
// Find the <SOAP-ENV:Body> tag using the envelope namespace
org.w3c.dom.NodeList nList
    = docEl.getElementsByTagNameNS(
        javax.xml.soap.SOAPConstants.URI_NS_SOAP_ENVELOPE, "Body");
if (nList.getLength() > 0)
{
    // Found our <PurchaseOrder> element. Plug it in
    org.w3c.dom.Node bodyNode = nList.item(0);
    org.w3c.dom.Node poRoot = poDoc.getDocumentElement();
```

Now we have the two elements that need to be attached to one another: the SOAP Body element and the PurchaseOrder element. However, you can't just reparent a Node from one document to another; doing so causes an error. Each element keeps track of its owning Document, which is checked by the individual routines that insert elements. Move a Node into another document properly by importing the Node and its subelements into the Document first. This operation performs a copy. The second parameter to `import()` is a Boolean that indicates whether this is a deep copy of all subnodes or just the current one. Once the nodes are imported into the Document that represents the envelope, we can simply attach the root Node as the immediate child of the <Body> element:

```
org.w3c.dom.Node importedNode
    = ((org.w3c.dom.Document)envelopeRoot).importNode(poRoot,
        true);
bodyNode.appendChild(importedNode);
```



If you use DOM level 3, there is an alternative to doing a copy. An experimental `adoptNode()` method reassigns an actual instance of a Node from one Document to another.

Now that the envelope is joined to the PO document, we take the root Node, convert it to a DOMSource, and place the whole thing into the messages's SOAPPart:

```
javax.xml.transform.dom.DOMSource domSource
    = new javax.xml.transform.dom.DOMSource(envelopeRoot);
soapPart.setContent(domSource);
```

Adding a header dynamically

You will encounter no surprises here. This code is symmetric to the Body APIs that we saw at the beginning of this section:

```
// Add an element and content to the Header
javax.xml.soap.Name name
    = envelope.createName("MessageHeader",
        "jaxm", "urn:oreilly-jaws-samples");
javax.xml.soap.SOAPHeaderElement headerElement
    = header.addHeaderElement(name);

// Add an element and content to the Header
name = envelope.createName("From");
javax.xml.soap.SOAPElement childElement
    = headerElement.addChildElement (name);
childElement.addTextNode ("Me");

// Add an element and content to the Header
name = envelope.createName("To");
childElement = headerElement.addChildElement(name);
childElement.addTextNode ("You");
```

Adding MIME attachments

Next, let's look at two (of many) ways to add attachments to our SOAP message. The simple sender uses both methods. No matter how you add an attachment, though, it requires two steps: call `createAttachmentPart()` with the appropriate content to get an attachment and `addAttachmentPart()` to add the attachment to the message.

The first method is arguably more complex; we use it to insert an external file (in this case, a purchase order, formatted as XML) as an attachment. First, we use the Activation Framework to create a `FileDataSource` that points at our external purchase order. We then convert the `FileDataSource` to a `DataHandler`; in turn, we use the `DataHandler` to create our first attachment, `ap1`, by calling `createAttachmentPart()`. Finally, we call `addAttachmentPart()` to add the attachment to the message.

Note that we don't need to specify the content type anywhere; the content type is provided automatically by the `DataHandler` object:

```
// Add additional Parts to the message
javax.activation.FileDataSource fds
    = new javax.activation.FileDataSource(m_attachment);
javax.activation.DataHandler dh
    = new javax.activation.DataHandler(fds);
javax.xml.soap.AttachmentPart ap1
    = message.createAttachmentPart(dh);
message.addAttachmentPart(ap1);
```

Perhaps a more intuitive way to create an attachment is to call `createAttachmentPart()` with the content and content type as arguments, as we've done here in the attachment `ap2`:

```
javax.xml.soap.AttachmentPart ap2
    = message.createAttachmentPart("Another Part",
        "text/plain; charset=ISO-8859-1");
message.addAttachmentPart(ap2);
```

JAXM Profiles

JAXM is capable of morphing itself into an API that frontends any number of SOAP-based messaging frameworks through the use of "profiles." A key part of a message profile is the ability to automate the creation of message headers and body elements that may be specific to Framework. We will describe this concept in more detail in a moment. In the 1.0 reference implementation, an ebXML MS profile and a SOAP-RP profile are provided as examples.

A profile consists of a `ProviderConnectionFactory`, a `ProviderMetaData` object that provides a list of profiles via a `getSupportedProfiles()` method, and a custom `MessageFactory` used to create messages specific to the profile being used. Let's look at how these are used.

ProviderConnectionFactory

`ProviderConnectionFactory` allows a JAXM client to obtain a `ProviderConnection` to a messaging provider, such as an ebXML Message Service, or a JMS provider that supports SOAP over JMS. A `ProviderConnectionFactory` can be configured administratively and retrieved via a `JNDI lookup()`. From there, a `ProviderConnection` is established:

```
ctx = new InitialContext();
ProviderConnectionFactory pcf =
    (ProviderConnectionFactory)ctx.lookup("GuaranteedMessaging");
ProviderConnection pc = pcf.createConnection();
```

Obtaining the profile via ProviderMetaData

Once the `ProviderConnection` is instantiated, the `ProviderMetaData` class can be queried to discover whether this connection supports a desired profile. The `getSupportedProfiles()` returns an array of `Strings` that lists the profiles that the `ProviderConnection` supports. For example, if ebXML is supported, the array will contain the string "ebXML":

```
ProviderMetaData pMetaData = pc.getMetaData();
String[] supportedProfiles = pMetaData.getSupportedProfiles();
String desiredProfile = null;

for(int i=0; i < supportedProfiles.length; i++) {
    if(supportedProfiles[i].equalsIgnoreCase("ebxml")) {
```

```

        desiredProfile = supportedProfiles[i];
        break;
    }
}

```

Using the custom MessageFactory to create profile-specific messages

It is possible to plug in a custom MessageFactory that creates a message in the form expected by the transport being used. For example, a MessageFactory for an ebXML profile might create a message with a SOAP envelope, which is prepopulated with the <MessageHeader> element in the SOAP header. In the following code, the EbXMLMessageImpl is a custom extension of the javax.xml.soap.SOAPMessage:

```

MessageFactory mf = pc.createMessageFactory(desiredProfile);
EbXMLMessageImpl ebxmlMsg = (EbXMLMessageImpl)mf.createMessage();

```

Sending the message

You can send a message with JAXM in two ways. One way uses the ProviderConnection.send() method. The other uses SOAPConnection.call(). The two methods have different purposes and different semantics. Since we are on the subject of ProviderConnection, we will talk about send() first and defer SOAPConnection.call() to the section “Simple Servlet Deployment.”

The ProviderConnection.send() method assumes that one-way asynchronous sending can occur. Whether the send() method blocks and waits for the operation to occur depends on the provider’s underlying message delivery semantics.

If you look at the API document for ProviderConnection.send(), you may notice that there is no way to specify a destination as part of the method signature. It assumes that the destination is established and somehow already associated with the SOAP message. There are a number of reasons for this design:

- The JAXM API is intended to be agnostic with regard to the underlying workings of the provider. The API is designed to work with many different providers, and specifying the destination as a parameter to send() may not always be appropriate.
- Whether the destination is a URI, URN, or an absolute URL is a function of the underlying infrastructure to which the JAXM API is attached. For instance, the messaging provider may provide an administration piece that maps generic URIs to specific destinations such as a URL or a JMS Topic or Queue.
- The SOAP header element stores the destination (or destinations). The SOAP header is constructed using the Envelope APIs (just like any other part of the message).

To facilitate setting the destination, JAXM provides an Endpoint class that specifies a URI as a destination. The following code assumes that the profile-specific message has

defined additional `setFrom()` and `setTo()` methods, and that the underlying infrastructure knows how to interpret the URI string used to construct the `Endpoint` object:

```
ebxmlMsg.setFrom(new Endpoint(from));  
ebxmlMsg.setTo(new Endpoint(to));  
pc.send(ebxmlMsg);
```

A strange inconsistency seems to exist in the API here: the code one would use to connect and send SOAP messages depends on how the client is deployed. When using a `ProviderConnection`, you send a message using the `java.xml.messaging.ProviderConnection.send()` method. Otherwise, you send the message using the `javax.xml.soap.SOAPConnection.call()` method. One could argue that the two scenarios are sufficiently different and don't warrant consistent APIs. If you write an application that is intended to connect to a larger framework, which implies using the `ProviderConnection` approach, many things specific to that framework have to be coded into the application (beyond just the connect and send operations).

JAX-RPC

JAX-RPC is a specification that is developing through the Java Community Process (JCP). It aims to provide a JCP-sanctioned standard set of Java APIs for both a client-side and server-side programming model.

These APIs leverage interoperable communications within Java applications with a protocol design center based on, but not limited to, SOAP. It covers the following areas:

- A Java code generation model for client-side stubs and server-side tie classes, based on a set of conventions for mapping WSDL to Java and Java to WSDL.
- An API for dynamic SOAP-RPC and a `Call` interface that is conceptually similar to Apache SOAP. `Call` semantics include synchronous invoke and synchronous invoke/one-way. JAX-RPC does not address asynchronous invocation in its 1.0 rendition. A true asynchronous model would require callbacks (`onMessage`, etc). The one-way invocation model defined in the API is considered synchronous.
- A model for defining a service, registering it, and invoking it within the J2EE and J2SE environments. This model covers typical J2EE/J2SE deployment issues such as creating deployment descriptors and packaging Web Application Archive (WAR) files.
- A binding to SOAP, including SOAP Fault handling through Java exceptions and `HeaderFault` processing.
- Type mappings between Java and XML datatypes.
- A service-side invocation handler mechanism used to chain together service method invocations.

- A reference implementation (RI) that provides a runtime implementation and a code generation tool, *xrpcc*.
- A serialization framework for marshalling and unmarshalling data between Java and XML based on soap-encoding rules. The RI from Sun includes an implementation of this framework.

JAX-RPC is a funny kind of animal. Finding the right kinds of things to write about it in the context of this book was a challenge. At the time of writing, the specification was still evolving. This chapter is based on a Version 0.6 snapshot of the specification's first public draft, issued as part of the Winter 01 Java Web Services Developer Pack. There are no known implementations, except for the RI (which is evolving with the specification) and the Apache Axis project (which is in alpha stage and based on Version 0.5 of the specification). Therefore, many examples in this section are repurposed directly from the specification. Several areas in the specification are in flux, need more definition, and are extremely likely to change before it hits a 1.0 status. Currently, the specification is 152 pages long and has enough information in it to comprise a whole book when it's finally finished.

In spite of its al dente status, the JAX-RPC specification has a brief working tutorial (as do all JAX products) that walks you through installation and setup and guides you through building a simple "Hello World" example. The RI runtime requires Tomcat; its installation, code generation tool, and deployment tool are based on Ant.

Most of the specification iterates over details and rules for mapping things between Java, XML, SOAP, and WSDL—not only for infrastructure provider runtime interactions, but also to provide directives and guidelines for code generation tools. If you are building a code generation tool, you will need more than we are presenting in this book.

Considering these factors, we still chose to write about JAX-RPC because it is a significant piece of functionality that is slated for J2EE 1.4. Therefore, its "baketime" will have to be short. We'll focus on the things that are baked and concentrate on the things that are exposed to the application developer. As the specification progresses, we will update our examples and provide new examples to go with them. Check this book's home page on the O'Reilly web site (<http://www.oreilly.com/catalog/javaweb-serv>) occasionally to see what we have placed there.

Stubs and Tie Classes

The concepts of "stub" and "tie" are not unique to JAX-RPC. The terms have been used in many other distributed-computing technologies, such as RMI, CORBA, and DCE RPC. Some of you may also be familiar with the terminology "stubs and skeletons." To understand stubs and ties, it is important to understand what's so exciting about remote procedure calls in the first place. The idea behind a remote procedure call is that an application makes a call to a method on an object, and the actual implementation of that object exists in another process space. The processes are

typically located on different machines separated by a network connection. The application making the method call (the client) does not need to know that it actually makes a call to a remote object (the service), nor does it need to worry about the details of how that happens.

The client application has a local object, the “stub,” that acts as a proxy for the remote object. The stub object has the same methods as the remote object, but does not implement the business logic. Instead, the stub represents an interface to an underlying infrastructure that is responsible for packaging the method name and its parameters into an agreed-upon wire format. This operation is sometimes referred to as *marshalling* or *encoding*. Upon reaching its destination, the skeleton or tie class is responsible for unmarshalling the wire format and reconstructing the data into a form that is recognizable by the server application. In the case of a remote procedure call, this means invoking the actual method with the expected parameters and datatypes, then marshalling return values back to the sender.

An assumption here is that a tool generates the stub and tie code, isolating the developer from the details of the marshalling and the transport protocol. In other distributed-computing technologies, in which both sides of the conversation are under the domain of a single vendor, a tool typically generates the stub and the skeleton at the same time, based on a generic description of an interface. With SOAP and web services, we break tradition because we can’t assume that the invoking client and the receiving service are part of the same software infrastructure. It is likely that each end of the conversation is built upon different software platforms. The one constant in the picture is that each side needs to interact with the same interface definition defined in WSDL and speak the same interoperable protocol, such as SOAP, which is also specified in WSDL as a binding.

Stub generation by a tool is only one option in JAX-RPC. We will discuss other options in the “JAX-RPC Client Invocation Models” section.

WSDL to Java, Java to WSDL

JAX-RPC defines a mapping of datatypes between WSDL and Java. The mappings cover simple datatypes, such as `short`, `int`, `long`, `float`, and `double`. It also has some fairly dry and boring rules about the mapping of arrays, structs and complex types, and enumerations. The good news is that this part of the specification is intended for vendors who build code generation tools that hide all these details behind a stub class.

You should also be familiar with definitions of parameter-passing modes. Some highlights that we will cover include remote references, pass-by-copy, and Holder classes.

Remote references

The generally accepted definition of a remote reference is an instance of a proxy that represents a particular instance of a remote object or service that can be transferred from one client to another. Remember that JAX-RPC doesn’t support remote

references in its pre-1.0 rendition, largely because SOAP doesn't have a model for remote references, either. A JAX-RPC client or server must be able to support any arbitrary SOAP message to or from a non-JAX-RPC entity.

Pass-by-copy and Holder classes

In a WSDL operation, a message part can be specified to appear within the input message only. This message part is considered an In parameter. A WSDL message part appearing only within an operation's output message can be thought of as an out parameter. Message parts appearing in both input and output messages within a WSDL operation can be called inout parameters.

A parameter marked as in indicates that the sending client has no further concern with what that remote procedure does with it once it is sent over the wire. This concept is commonly referred to as *pass-by-value* or *pass-by-copy*. An out parameter is a return value. An inout parameter is expected to be modified by the remote procedure, causing the calling client to see the modified value after the method invocation returns.

Generally speaking, you can support inout parameters in two ways: with pass-by-reference and Holder classes. In distributed-computing environments that support pass-by-reference, an object reference is a specific datatype that can be passed as a parameter to a remote object. It can be treated as the handle to an actual instance of the parameter. Once the remote method call is complete, it may be dereferenced by the client program to obtain the newly modified value. The underlying infrastructure marshalls the data over the wire to get the right result.

Java doesn't support pass-by-reference natively for primitive types. Instead, JAX-RPC uses Holder classes, which are classes used by the underlying infrastructure to act as a place to hold the values. A stub uses a Holder to store the modified values after the method invocation so the calling client can then access them. Here's an example illustrating the use of Holders:

```
public interface StockQuoteProvider extends java.rmi.Remote {
    // Method returns last trade price
    float getStockQuote(String tickerSymbol,
        javax.xml.rpc.holders.IntHolder volume,
        javax.xml.rpc.holders.FloatHolder bid,
        javax.xml.rpc.holders.FloatHolder ask)
        throws java.rmi.RemoteException;
}

//Java
package javax.xml.rpc.holders;
public final class IntHolder {
    public int value;
    public IntHolder() { }
    public IntHolder(int value) {
        this.value = value;
    }
}
```

Holders are also used to represent complex data structures as parameters. The `java.xml.rpc` Holders package defines Holders for all the built-in Java primitives. Beyond that, the code generation tool is responsible for creating Holders for the parameters. In either case—whether using the supplied Holder classes or the custom generated ones—the code generation tool is also responsible for creating the code that serializes and deserializes data sent across the wire.

Generated service interface

JAX-RPC allows a code generation tool to create an implementation class for the Service interface based on an existing WSDL document. While the implementation details of the generated class are vendor-specific, the generated interface is required to adhere to the following design pattern:

```
public interface <ServiceName> extends javax.xml.rpc.Service {
    public <ServiceDefInterface> get<Name of the wsdl:port>()
        throws JAXRPCException;
    ...
}
```

In the previous listing, the `<ServiceName>` of the generated service interface is mapped from the name attribute of the corresponding `wsdl:service` definition. The `<ServiceDefInterface>` name is mapped from the `portType`, and the `get<Name of the wsdl:port>()` method is mapped from the—you guessed it—`wsdl:port` definition's name attribute. For example, given the following WSDL definition:

```
<service name="StockQuoteService">
  <port name="StockQuoteProviderPort" binding="tns:somebinding">
    <http:address location="http://example.com/" />
  </port>
</service>

<portType name="StockQuoteProvider">
  <operation name="GetLastTradePrice"
    parameterOrder="tickerSymbol">
    ...
  </operation>
</portType>
```

a JAX-RPC implementation creates the following Java interface; the `get` method returns an instance of a stub class that implements the `<ServiceDefInterface>`:

```
package com.example;
public interface StockQuoteService extends javax.xml.rpc.Service {
    public StockQuoteProvider getStockQuoteProviderPort()
        throws JAXRPCException;
    ...
}
```

Value types

Value types are specific to the generation of WSDL from a Java class. A value type is a special form of serializable class, containing data values that are capable of being marshaled between a client and a service. It must implement `Serializable`, but does not follow the standard rules for Java serialization. Only public, nontransient data members are mapped to the WSDL. A value type may also be a `JavaBean`, in which case, the WSDL generation may use bean introspection to identify the properties and map them to the WSDL. The methods in the value class are not mapped to the WSDL; it is strictly for data. Whether a value type should use the `Serializable` marker interface or whether a different interface should be defined is still undecided.

SOAPElement API

JAX-RPC also allows the use of a `javax.xml.soap.SOAPElement` object as a parameter value for a remote method. This object is intended for when you want to bypass an existing datatype mapping or when a mapping doesn't exist for the data you are using. You can also use it if you just want to plug in the element by hand. Regardless of the reasons, whatever you place in the `SOAPElement` parameter becomes the request envelope's body.

If you recall the JAXM section, we used the `SOAPElement` like this:

```
// Add an element and content to the Header
name = envelope.createName("From");
javax.xml.soap.SOAPElement childElement
    = headerElement.addChildElement (name);
childElement.addTextNode ("Me");
```

In this code, we create the element as a side effect of appending the name to the `headerElement`. A more direct way to create an element uses the `SOAPElementFactory` to create an element and populate its contents. To create a `SOAPElementFactory`, call its static method `newInstance()`:

```
javax.xml.soap.SOAPElementFactory sef
    = javax.xml.soap.SOAPElementFactory.newInstance();
javax.xml.soap.SOAPElement sel = sef.createElement(...);
```

The interface for the `SOAPElementFactory` is:

```
package javax.xml.soap;
public abstract class SOAPElementFactory {
    public abstract SOAPElement create(Name name)
        throws SOAPException;
    public abstract SOAPElement create(String localName)
        throws SOAPException;
    public abstract SOAPElement create(String localName, String prefix,String uri)
        throws SOAPException;
    public static SOAPElementFactory newInstance()
        throws SOAPException;
}
```

The first public draft of the JAX-RPC specification identifies some problems with this approach, which may mean that it's likely to change in a future draft. One problem is that the `Name` object is created using the `Envelope` object, which is not available to the JAX-RPC programmer. "Not available" means that nothing exposed in the JAX-RPC client interface lets you peek at the envelope that will eventually be sent. The lack of a factory for a `Name` is not that big of an issue in itself. The `Name` interface contains a local name, a namespace prefix, and a URI. An alternate method signature lets you create the `SOAPElement` with the desired result. More significant issues are probably also behind this problem.

JAX-RPC Client Invocation Models

JAX-RPC defines three different client models used to invoke a remote method: one static model and two dynamic models. The statically defined stub model is typically based on a code generation tool. The dynamic proxy invocation model is based on building a proxy object dynamically using the reflection APIs (`java.lang.reflect`). The Dynamic Invocation Interface (DII) is based on a `Call` object similar to the Apache SOAP `Call` interface we saw in Chapter 5.

Statically Generated Stubs

A tool can generate a class that implements the `javax.xml.rpc.stub` interface, which contains the following methods:

```
package javax.xml.rpc;  
public interface Stub {  
    public void _setProperty(String name, Object value);  
    public Object _getProperty(String name);  
    public java.util.Iterator _getPropertyNames();  
}
```

In addition to implementing these methods, the generated stub would have a method that matches the name of the actual service method, such as `getLastTradePrice()`. The underlying implementation of this method can be anything the tool and the infrastructure agree upon. It is not required to be transport-independent; in fact, stubs are usually bound directly to a transport. The methods in the `Stub` interface exist to allow dynamic capabilities in the static stub.

Table 7-2 lists the handful of predefined properties that can be set on the `Stub` class. These properties are expected to be set prior to making a method call, based on the assumption that the underlying transport or infrastructure might need the information contained in these properties to reach the service successfully.

Table 7-2. Stub properties

| Property name | Value | Description |
|--|------------------|---|
| http.auth.username | java.lang.String | Username for the HTTP Basic authentication. |
| http.auth.password | java.lang.String | Password for the HTTP Basic authentication. |
| javax.xml.rpc.service.endpoint.address | java.lang.String | Target service endpoint address. The URI scheme for the endpoint address specification must correspond to the protocol/transport binding for this stub class. |

By convention, the fully qualified package name should be part of any property name. Vendor-specific properties should follow the same naming convention, using the vendor's package-naming style.

As of JAX-RPC 1.0 Public Draft, whether there should be a standard set of accessor methods for these standard properties was an issue under discussion.

Dynamic Invocation Using the Service Interface

The `javax.xml.rpc.Service` interface encapsulates two flavors of dynamic invocation that do not require any generated code. These methods are dynamic proxy invocation via the `getPort()` method and the DII using the `Call` interface. Here is the `Service` interface definition:

```
package javax.xml.rpc;
public interface Service {
    public Call createCall()
        throws JAXRPCException;
    public Call createCall(QName portName)
        throws JAXRPCException;
    public Call createCall(QName portName, java.lang.String operationName)
        throws JAXRPCException;
    public java.rmi.Remote getPort(QName portName,
        java.lang.Class serviceDefInterface)
        throws JAXRPCException;
    public java.util.Iterator getPorts();
    public QName getServiceName();
    public TypeMappingRegistry getTypeMappingRegistry()
        throws JAXRPCException;
    public java.net.URL getWSDLDocumentLocation();
    public void setTypeMappingRegistry(TypeMappingRegistry registry)
        throws JAXRPCException;
}
```

The dynamic proxy approach doesn't require any generated code. `javax.xml.rpc.Service.getPort()` returns a dynamic proxy for the object being operated on. Only a `QName` to identify the port and a compiled interface definition class are required:

```
com.example.StockQuoteProvider sqp =
    (com.example.StockQuoteProvider)service.getPort(portName,
        StockQuoteProvider.class);

float price = sqp.getLastTradePrice("ACME");
```

In this code, the `getPort()` method is passed in an interface definition that will be used as a template for building a runtime instance of a dynamic proxy. Thus, the returned object is not the same as the passed-in object. A typical dynamic proxy implementation uses the `java.lang.reflect.Proxy` object to build a table of method objects, which are dispatched using `java.lang.reflect.InvocationHandler.invoke()`. In the context of JAX-RPC, that validation would result in constructing the appropriate SOAP envelope and sending it on its way. `getPort()` may choose to validate the interface and the port name against its WSDL if it likes, but this is not required by the specification.

For more information on dynamic proxies, refer to the J2SE documentation for `java.lang.reflect.Proxy` and `java.lang.reflect.InvocationHandler`, found at <http://java.sun.com/j2se/1.3/docs/guide/reflection/>.

Dynamic Invocation Interface (DII)

DII is based on a `Call` object. To get a `Call` object, use the `javax.xml.rpc.Service.createCall()` method. There are three overloaded versions of `createCall()`:

```
package javax.xml.rpc;
public interface Service {
    public Call createCall() throws JAXRPCException;
    public Call createCall(QName portName) throws JAXRPCException;
    public Call createCall(QName portName, String operationName)
        throws JAXRPCException;
    ...
}
```

When the `Call` object is created, it has little or no information about the invocation that needs to be made. The information required to execute the call (parameters, types, etc.) is constructed using its set methods. Here's the definition of the `Call` interface; note that the methods that are not highlighted are listed in the specification, but are not yet implemented in the reference implementation or listed in the javadoc:

```
public interface Call {
    public boolean isParameterAndReturnSpecRequired();

    // Parameter passing and return type handling
    public void addParameter(String paramName,
        QName xmlType, ParameterMode parameterMode)
        throws JAXRPCException;
    public QName getParameterTypeByName(String paramName);
    public void setReturnType(QName xmlType)
        throws JAXRPCException;
    public QName getReturnType();
    public void removeAllParameters();

    // WSDL operation
    public QName getOperationName();
    public void setOperationName(QName operationName);
}
```

```

// WSDL portType
public QName getPortTypeName();
public void setPortTypeName(QName portType);

// WSDL endpoint
public String getTargetEndpointAddress();
public void setTargetEndpointAddress(String address);

// properties
public void setProperty(String name, Object value)
    throws JAXRPCException;
public Object getProperty(String name);
public void removeProperty(String name);
public java.util.Iterator getPropertyNames();

// Remote Method Invocation methods
public Object invoke(QName operationName, Object[] inputParams)
    throws java.rmi.RemoteException, JAXRPCException;
public Object invoke(Object[] inputParams)
    throws java.rmi.RemoteException, JAXRPCException;
public void invokeOneWay(Object[] inputParams)
    throws javax.xml.rpc.JAXRPCException;
public java.util.Map getOutputParams()
    throws javax.xml.rpc.JAXRPCException;
}

```

The Call object in JAX-RPC differs from the Apache SOAP Call object; it encapsulates both message-style invocation and RPC-style invocation in a single interface. Unlike ApacheSOAP, Call.invoke() and Message.send() operations aren't separate.

Building the method signature

When using a Call object, you can build parameters dynamically by using the addParameter() method. Two modes of invocation are available: synchronous request/response (using the invoke() method) and asynchronous fire-and-forget (using invokeOneWay()). The addParameter() method has the following signature:

```

public void addParameter(String paramName,
    QName xmlType, ParameterMode parameterMode)

```

The ParameterMode class evaluates to one of three possible values: in, out, or inout. The definition of the class is:

```

// Typesafe Enumeration for ParameterMode
public class ParameterMode {
    private final String mode;
    private ParameterMode(String mode) {
        this.mode = mode;
    }
    public String toString() { return mode; }
    public static final ParameterMode PARAM_MODE_IN =
        new ParameterMode("PARAM_MODE_IN");
    public static final ParameterMode PARAM_MODE_OUT =
        new ParameterMode("PARAM_MODE_OUT");
}

```

```

    public static final ParameterMode PARAM_MODE_INOUT =
        new ParameterMode("PARAM_MODE_INOUT");
}

```

The Call implementation is required to validate the parameters and the return type. It can do so by relying on the client code to call the addParameter() and setReturnType() methods.

Notice that addParameter() doesn't take a value. It is not building a parameter list; it is simply building the method signature. In JAX-RPC, the values for the parameters are passed as an array of Objects to invoke() or invokeOneway(). The invoke() implementation is responsible for validating the parameters with the signature that is built. An implementation that requires you to build the parameter list must explicitly return true from the isParameterAndReturnSpecRequired() method.

Alternatively, an implementation may support the definition of a method signature through a type mapping registry or through another means that matches a signature with its corresponding WSDL definition. In this case, the client code is not required to call addParameter() and setReturnType(). If the implementation doesn't require or support the addParameter()/setReturnType() approach, it must throw a JAXRPCException if the client code attempts to call these methods. Furthermore, the isParameterAndReturnSpecRequired() method must return false.

Setting the properties

Like the Stub class, the Call class has several predefined properties that are listed in Table 7-3. These properties are expected to be set prior to making the actual method call, based on the assumption that the underlying transport or infrastructure might need this kind of information to reach the service successfully.

Table 7-3. Call properties

| Property name | Property type | Description |
|---|-------------------|--|
| javax.xml.rpc.security.auth.username | java.lang.String | Username for Authentication. |
| javax.xml.rpc.security.auth.password | java.lang.String | Password for Authentication. |
| javax.xml.rpc.soap.operation.style | java.lang.String | "rpc" if the operation style is rpc; "document" if the operation style is document. Note that a Call implementation may choose not to allow the setting of this property. In this case, the setProperty method throws JAXRPCException. |
| javax.xml.rpc.soap.http.soapaction.use | java.lang.Boolean | Indicates whether SOAPAction will be used. |
| javax.xml.rpc.soap.http.soapaction.uri | java.lang.String | Indicates the SOAPAction URI if the javax.xml.rpc.soap.http.soapaction.use property is set to true. |
| javax.xml.rpc.encodingstyle.namespace.uri | java.lang.String | Encoding style specified as a namespace URI. The default value is the SOAP 1.1 encoding. |

Making the call and retrieving the results

Regardless of how the method signature is created, the `invoke()` and `invokeOneWay()` methods are responsible for matching up the parameters and return types with the supplied signature. They are required to generate a `JAXRPCException` if an error occurs until the client has delivered its payload onto the wire. Examples of errors that can occur include signature/parameter mismatch, or specifying an out or inout parameter and calling `invokeOneWay()`. The `invoke()` method must continue to block until the remote service receives the method call and returns either a response or a remote exception. The `invokeOneWay()` method is not allowed to propagate a remote exception—another subtle difference between JAX-RPC and Apache SOAP.

Once the invocation has taken place, you can obtain the out and inout parameters by calling the `getOutputParams()` method:

```
// create the call object.
javax.xml.rpc.Call call = service.createCall(portName, "<operationName>");

// build the method signature.
call.addParameter("param1", <xsd:string>, ParameterMode.PARAM_MODE_IN);
call.addParameter("param2", <xsd:string>, ParameterMode.PARAM_MODE_OUT);
call.setReturnType(<xsd:int>);

// build the parameter list itself.
Object[] inParams = new Object[] {"<SomeString>"};

// invoke the remote method
Integer ret = (Integer) call.invoke(inParams);

// get the output parameters
Map outParams = call.getOutputParams();
String outValue = (String)outParams.get("param2");
```

Note that the number of calls to `addParameter()` does not necessarily match the number of parameters placed on the parameter list. This mismatch is not obvious at first. The reason for the discrepancy is that some parameters are purely output parameters, as specified by the `ParameterMode.PARAM_MODE_OUT`.

Service Context Propagation and SOAP Message Handlers

Remember the discussion we had about SOAP headers in Chapter 3? A typical use for a header element would be to pass around a transaction ID; that is one example of propagating conversational or contextual information between service clients and service implementations. In JAX-RPC, this contextual information is referred to as a *Service Context*. A Service Context is implementation-dependent and may include such things as a transaction ID, a security token, or some sort of conversation ID.

A Service Context may be handled explicitly or implicitly by the JAX-RPC runtime environment. Those of you familiar with CORBA OTS or JTA should be familiar with the notions of implicit and explicit propagation. For implicit propagation, the generated client stub code may transparently propagate a security token by marshaling it into the SOAP header when it generates the envelope. In the explicit case, the generated stub code has additional parameters appended to the end of each method signature. For example:

```
public interface StockQuoteProvider extends java.rmi.Remote {  
    // Method returns last trade price  
    float getStockQuote(String tickerSymbol, StringHolder context)  
        throws java.rmi.RemoteException;  
}
```

The SOAP Message Handler APIs provide a way to expose implicit context propagation to the application developer. The Handler interface is basically a way to plug your own interceptors into the runtime infrastructure for both the client-side stub implementation or the server-side invocation. Simply write a class that extends the `javax.xml.rpc.handler.Handler` interface and plug it into a Handler chain. At runtime, your `Handler.handle()` method is invoked and is passed a `SOAPMessageContext` object. The `SOAPMessageContext` gives you full access to the SOAP envelope, to do with what you wish.