Chapter 4: Transferring Data with Web Services

Overview

In the previous chapter, we created our own web service using the Remote Procedure Call (RPC) model. More specifically, we designed a Java class named `StockQuote` with a single method, `getQuote()`, that we called it over a network using the SOAP protocol. The fact that the data exchanged between the client and server was in the form of an XML document was incidental; in fact we could have used another protocol such as JRMP.

In this chapter, we will take another point of view–we will regard the XML documents that transit between the caller and the web service as essential. This type of SOAP development is called **document-style** SOAP programming.

In this chapter we shall discuss:

- An overview of document-style SOAP development
- The Document Object Model (DOM) and the Simple API for XML (SAX) to compare the pros and cons of both models
- A modified version of `StockQuote – Portfolio`, which will allow us to illustrate the concepts of document-style SOAP development

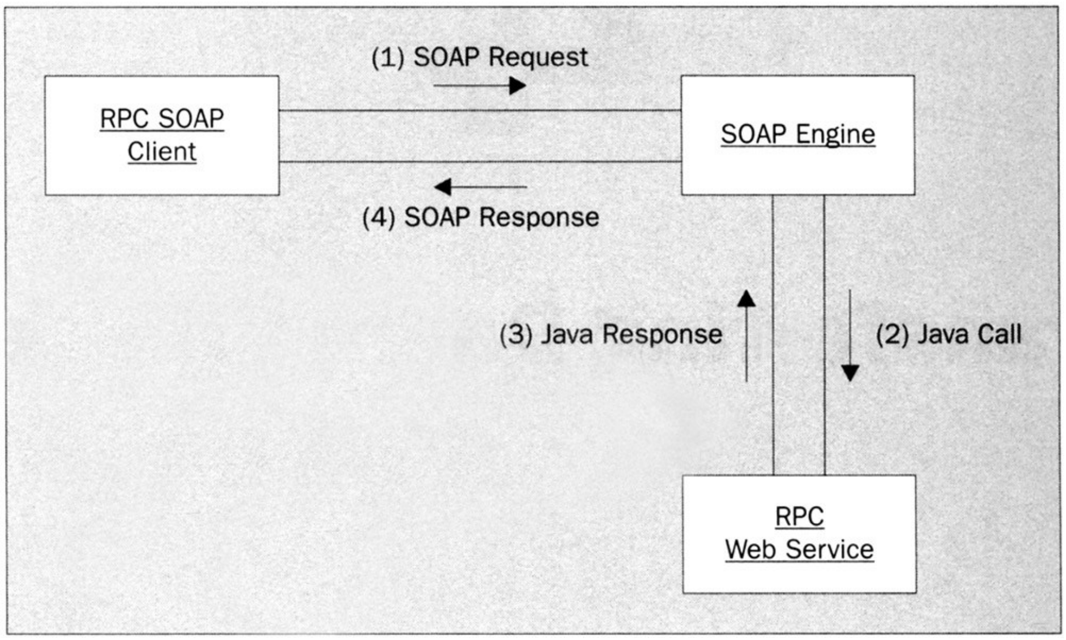
Let us begin by comparing RPC and document-style SOAP development.

RPC versus Document-Style

As shown by the diagrams overleaf, from an external perspective, RPC-style and document-style SOAP development are similar.

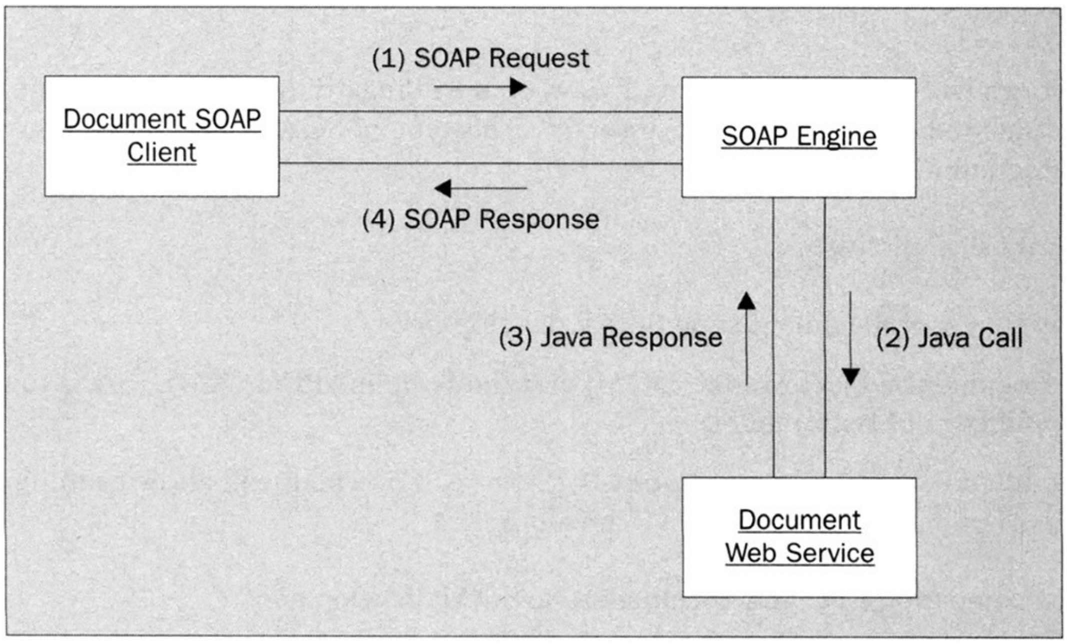
- **RPC-style SOAP development**

  In this, the client makes a remote method call to a web service instantiated on the server. At a macro level, the client submits a SOAP request and gets a SOAP response:



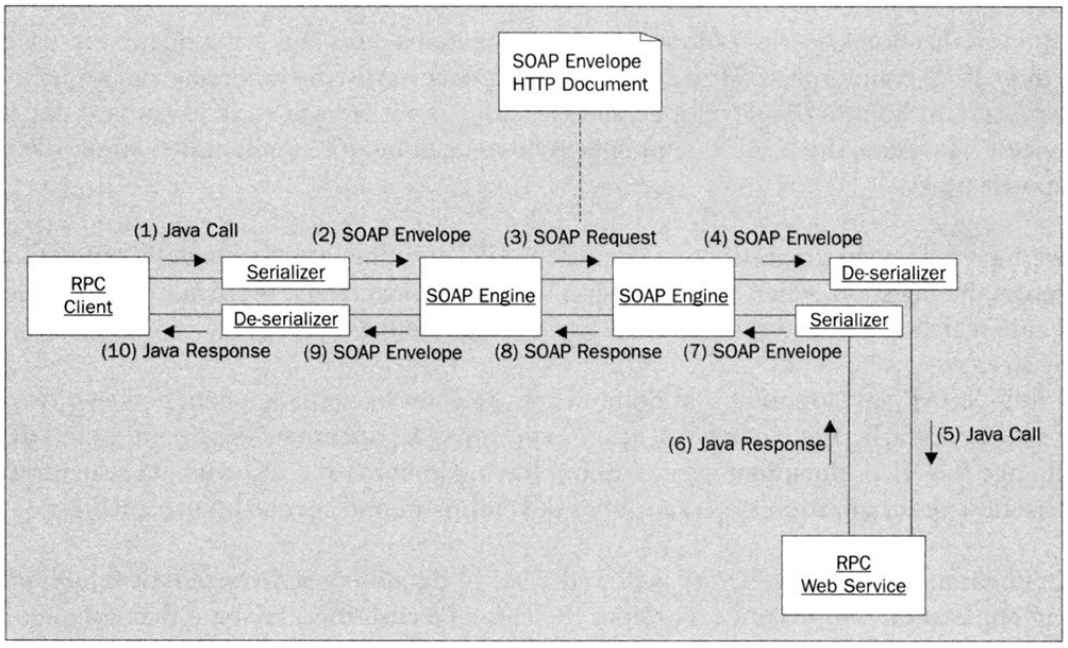
- **Document-style SOAP development**

  In this, the client submits an XML document to a web service instantiated on the server. Here also, at a macro level, the client submits a SOAP request and gets a SOAP response:

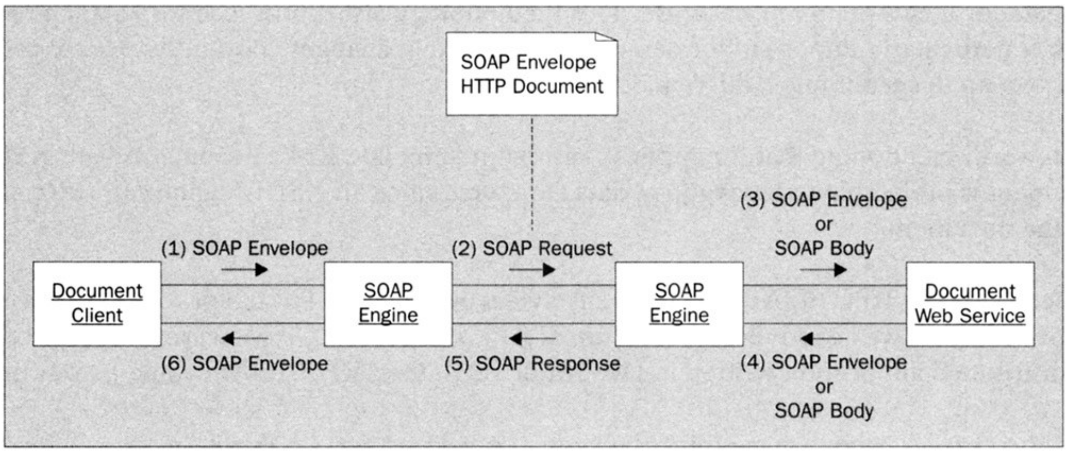


However, when we look at them from an internal perspective, things are different.

- **RPC-style SOAP development:**



- **Document-style SOAP development:**



The main differences between these two are:

- **Serialization and Deserialization**

  In document-style SOAP development, the client does not need to serialize a Java call and its arguments into an XML document. Conversely, the server does not need to deserialize the XML document into a Java call and data types.

- **Semantic**

  In RPC-style SOAP development, the client and the server are forced to adhere to a well-defined programming model – the client calls a method with possible arguments, and the server returns one value as a response. In document-style, an XML document is exchanged and the meaning of each element is left to the interpretation of the participants. In addition, the client and the web service can bundle multiple documents in the request and the response.

By looking at the last two diagrams, we can also infer that the reliance on the SOAP engine is minimized when using the document-style model. This simplification of the SOAP infrastructure is mostly due to the absence of a serialization engine.

As there is no serialization/deserialization process, applications designed using document-style ought to be faster than their RPC counterparts. However, that is not necessarily the case; one can argue that in document-style, the serialization is up to the developer, since at some point we will have to convert our Java data into XML and vice-versa. Also, the XML documents involved can be potentially much larger than the simple request-response types.

Now that we have reviewed the distinctions between RPC and document-style development, it is appropriate to ask ourselves the question, when should we use them? Unfortunately, there are no easy rules that we can apply to decide which style to adopt; but there are a few guidelines.

Document-style SOAP development is at home when we want to exchange data between two or more parties. This is particularly true when we already have an XML document representing the data as we can simply exchange the XML document as-is, without having to convert it to Java data structures. The removal of the serialization/deserialization steps simplifies development and speeds up processing.

Another application that lends itself very well to the use of document-style SOAP development is a data transforming application. For instance, consider the case of a customer database that contains names and addresses, but no demographic information like income, family size, neighborhood type, and so on. In order to improve our knowledge of the customers, we can submit their records to a demographic data enhancement service, which will return the customer data along with the desired demographics. We can then process the enhanced data for inclusion in our database. This methodology also works well for keeping records up-to-date, which is particularly important for demographic data that changes constantly (people get married, have children, move up the economic ladder, and so on).

Finally, it is worth mentioning that for applications that manipulate XML documents with XSLT, having an XML document is preferable to having Java data structures since an XSLT engine can automatically transform the document.

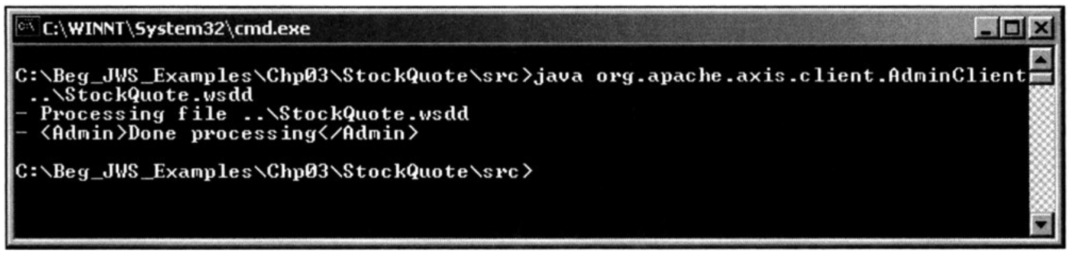
On the other hand, the RPC SOAP development style works very well if the problem can easily be modeled with method calls or if we already have a functional API. For instance, if we were adding a web service access layer to a distributed application written in DCOM or RMI, then RPC SOAP would be our first choice.

Later on in the chapter, when we examine the code of our `Portfolio` example, we will look at creating and parsing an XML document. But before bringing the discussion to a practical level, we must discuss the support provided by Axis for document-style SOAP development.

The easiest way to get more familiar with document-style SOAP development is to try it out with an example.

The StockQuote Deployment

In Chapter 3, we deployed the `StockQuote` web service using the `AdminClient` provided by Axis where we simply submitted an XML document to Axis for processing:



```
C:\Beg_JWS_Examples\Chp03\StockQuote\src>java org.apache.axis.client.AdminClient
 ..\StockQuote.wsdd
- Processing file ..\StockQuote.wsdd
- <Admin>Done processing</Admin>

C:\Beg_JWS_Examples\Chp03\StockQuote\src>
```

In other words, we are already familiar with document-style processing since we used it in the submission of a deployment descriptor. Note that only the deployment/un-deployment of the `StockQuote` web service is considered document-style SOAP programming; the `StockQuote` service uses the RPC model.

This simple example shows us that document-style SOAP development is more natural than we might think. If we were to re-design the deployment descriptors with an RPC API, we would have to define an interface that supports the dynamic nature of deployment descriptors. This interface would then involve arrays, variable number of arguments, and possibly method overloading, thus making it more complicated than its document-style counterpart.

If we have an XML background, we will be more likely to favor the simple XML schema over an informal description of Java methods (such as Javadoc) with several special cases. This last point touches on another characteristic of document-style SOAP development – XML documents are typically more expressive and therefore more succinct than their RPC counterparts. We will illustrate these points when we review the `Portfolio` example.

To take a closer look at what is happening here, we will have to talk in more detail about how deployment descriptors are handled in Axis.

In the SOAP request for the deployment of `StockQuote`, the portion shown below is passed to the service:

```
POST /axis/services/AdminService HTTP/1.0
Content-Length: 587
Host: localhost
Content-Type: text/xml; charset=utf-8
SOAPAction: "AdminService"

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <deployment
      xmlns="http://xml.apache.org/axis/wsdd/"
      xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
      <service name="StockQuote" provider="java:RPC">
        <parameter name="className"
          value="com.wrox.jws.stockquote.StockQuote"/>
        <parameter name="allowedMethods" value="getQuote"/>
      </service>
    </deployment>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The actual document that is typically processed by a web service is the document fragment residing inside the `<deployment>` element (more on that in a moment):

```
    <deployment
      xmlns="http://xml.apache.org/axis/wsdd/"
      xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
      <service name="StockQuote" provider="java:RPC">
        <parameter name="className"
          value="com.wrox.jws.stockquote.StockQuote"/>
        <parameter name="allowedMethods" value="getQuote"/>
      </service>
    </deployment>
```

If we look closely at the HTTP header of the previous SOAP request, we will see that the target web service is `AdminService()`, which is an Axis built-in web service that handles the deployment and un-deployment of web services.

Since our goal is to discuss document-style support in Axis, we don't need to know about all the intricacies of the `AdminService()` implementation. For our discussions, we are only concerned about the method that handles the incoming requests (deployment and un-deployment descriptors). As indicated by the `SOAPAction` in the HTTP header, this method bears the same name as its class (this is not always the case):

```
    public Element[] AdminService(Vector xml) throws Exception
```

The signature of the `AdminService()` method contains one argument, `java.util.Vector`, and returns one array of `org.w3c.dom.Element` objects. This might look surprising at first, as we might have expected to see something along the lines of:

```
    public Document AdminService (Document document) throws Exception
```

As a matter of fact, this second signature was a perfect fit; it was the signature of document-style web services in early versions of Axis. However, this signature had two shortcomings:

- **Only one document is passed in**

  If we go back to the SOAP specification, we will see that it is perfectly legal to pass more than one XML document as part of a SOAP packet. With the second `AdminService()` signature, only one document is passed to the web service.

- **Only one document is returned**

  Same problem as mentioned previously, but with the return value. If we pass more than one document as input, we might expect to return more than one document.

The remainder of `AdminService()` contains few surprises (the `MessageContext` is used by Axis to keep track of the state of a message):

```
          MessageContext msgContext = MessageContext.getCurrentContext();
        log.debug(JavaUtils.getMessage("enter00", "Admin:AdminService"));
        Document doc = process( msgContext, (Element) xml.get(0) );
```

The `process()` method handles the deployment descriptor. We can see above that the first element of the vector is passed to the `process()` method. In other words, if we were to send multiple deployment descriptors as part of the SOAP message, only the first one would be treated. The return value of the process is an `org.w3c.dom.Document` that we convert to an `org.w3c.dom.Element`, since it is the return type expected by Axis:

```
        Element[] result = new Element[1];
        result[0] = doc.getDocumentElement();
        log.debug(JavaUtils.getMessage("exit00", "Admin:AdminService") );
        return result;
      }
```

Note The class that contains the call to the AdminService() method is org.apache.axis.providers.java.MsgProvider.MsgProvider is the built-in provider for document-style web services in Axis. RPCProvider is the same package that we used in the previous chapter. It is the built-in provider for RPC-style web services.

Before looking at a more complete example, let's mention that Axis also supports a special form of document-style web services, called **full message service.** With full message service, not only the content of the SOAP body, but also the entire SOAP envelope is passed to the callback method. This description sheds some light on the difference between the client and the server that we saw in the earlier diagram – the client submits a SOAP envelope, full message services are handed a SOAP envelope, and other services deal with the SOAP body.

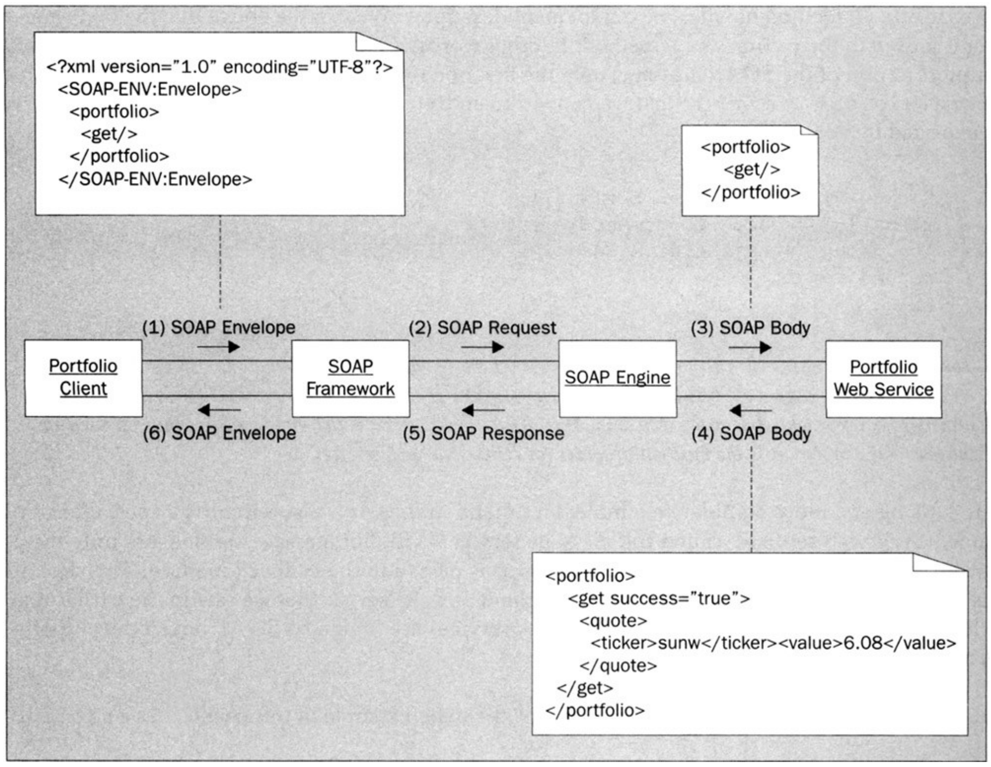
In the next section we will demonstrate and review the main example of this chapter, `Portfolio`.

Portfolio

To illustrate document-style SOAP development, we will develop a `Portfolio` example. This application is a refinement of the `StockQuote` example, which we wrote in the previous chapter. The `Portfolio` example supports three basic features:

- The user of the service can add stocks to their portfolio
- The user of the service can remove stocks from their portfolio
- The user of the service can get the status of their portfolio

The following figure illustrates the flow of data in the `Portfolio` application. Predictably, it follows the structure that we described earlier for document-style development. The `Portfolio` example is not a full message service; it only deals with the body that is inside its root element `<portfolio/>`:



Since it is an example, in the interest of keeping the discussion focused on document-style SOAP development, we have introduced a few limitations in it. The main restrictions are:

- The application does not provide a GUI; it only provides a command line interface to modify and retrieve the portfolio. The portfolio is displayed in raw XML.
- The portfolio is not persistent; in other words, if the serverside of the SOAP engine (for example, the SimpleAxisServer or Catalina) is restarted, we will lose our portfolio.
- The application supports only one portfolio.

Note that, these restrictions have little to do with SOAP development and will not interfere with our main objective of describing the issues faced by a SOAP developer in working with the document-style paradigm.

Try It Out: Portfolio Example

Trying out the Portfolio sample is similar to what we did for `StockQuote`: we need to compile, deploy, and test the web service.

1. Here is the entire implementation of Portfolio:

```
package com.wrox.jws.stockquote;

import java.util.Vector;
import java.util.set;
import java.util.HashSet;
import java.util.Collections;
import java.util.Iterator;

import java.net.URL;

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;

import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.message.SOAPEnvelope;
import org.apache.axis.message.SOAPBodyElement;
import org.apache.axis.utils.XMLUtils;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

import com.wrox.jws.stockcore.StockCore;

public class Portfolio {
  // The valid commands from the user. These are also the valid XML elements
  final static String addCommand    ="add";
  final static String removeCommand ="remove";
  final static String getCommand    ="get";

  // Attribute to be added to commands to give a status; might be true/false
  final static String successAttribute ="success";

  final static String nameSpaceURI ="Portfolio"; // portfolio ns URI
  final static String portfolioTag ="portfolio"; //Root of the document
  final static String tickerTag ="ticker"; // A ticker inside add or remove
  final static String quoteTag ="quote";     // A quote (response to a get)
  final static String valueTag ="value";     // A value (inside a quote)

  // Our (simplistic) portfolio
  static Set portfolioSet = Collections.synchronizedSet (new HashSet());

  /**
   * Creates a <quote/> element with the values passed in. The result should
   * look like the following XML:
   *     <quote>
   *       <ticker>ticker</ticker>
   *       <value>value</value>
   *     </quote>
   *
   * This method is used internally to build the document fragment returned
   * by the <get/> command.
   *
   * @param document the document that will contain the element
   * @param tickerName the ticker symbol for the stock (e.g. sunw)
   * @param tickerValue the value of the stock (e.g. 100.00)
   * @return the newly created <quote/> element
   */
  private static Element createQuote (Document document,
      String tickerName, String tickerValue) {

    // Create the elements and text nodes
    Element  quote      = document.createElement (quoteTag);
    Element  ticker     = document.createElement (tickerTag);
    Element  value      = document.createElement (valueTag);
    Node     nameText   = document.createTextNode (tickerName);
    Node     valueText  = document.createTextNode (tickerValue);

    // Create the hierarchy
    ticker.appendChild(nameText);
    value.appendChild(valueText);
    quote.appendChild(ticker);
    quote.appendChild(value);

    return quote;
  }

  /**
   * Process a set of commands. The valid commands are:
   *   <get/>
   *   <add><ticker>ticker1</ticker>...<ticker>tickern</ticker></add>
   *   <remove><ticker>ticker1</ticker>...<ticker>tickern</ticker></remove>
   *
   * The output document depends on the input commands. For add and remove,
   * we simply echo the command with the attribute success="true". For the
   * get command, we list the portfolio using the following format:
   *   <get success="true">
   *     <quote>
   *       <ticker>ticker1</ticker>
   *       <value>value1</value>
   *     </quote>
   *     ...
   *     <quote>
   *       <ticker>tickern</ticker>
   *       <value>valuen</value>
   *     </quote>
   *   </get>
   *
   * @param command the command in the form of an xml document
   * @return the resulting XML document
   */
  public static Element [] portfolio (Vector xmlDocument) throws Exception {

    Document requestMessage=
      ((Element)xmlDocument.get(0)).getOwnerDocument();
    Document responseMessage = (Document)requestMessage.cloneNode (true);

    Element[] result = new Element [1];
    result [0] = portfolio(requestMessage.getDocumentElement(),
      responseMessage.getDocumentElement());

    return result;
  }

  /**
   * Called by the public version of portfolio. Useful for testing as
   * inproc.
   *
   * @param requestElement the <portfolio/> element of the request
   * @param requestElement the <portfolio/> element of the response
   * @return the modified requestElement
   */
  private static Element portfolio (Element requestElement,
      Element responseElement) throws Exception {

    // We traverse the document and process each command as we go
    NodeList nodes = responseElement.getChildNodes();
    for (int nodeIndex = 0; nodeIndex < nodes.getLength(); nodeIndex++) {
      Node currentNode = nodes.item(nodeIndex);

      if (currentNode instanceof Element) { // Ignore anything else
        Element commandElement = (Element)currentNode;
        String commandName     = commandElement.getTagName();
        boolean success        = true;

        if (commandName.equals(addCommand) ||
            commandName.equals(removeCommand) ) {
            // We get the list of ticker symbols to add/remove
            NodeList tickerNodes = commandElement.getChildNodes();
            for(int tickerIndex = 0; tickerIndex < tickerNodes.getLength();
                tickerIndex++) {
              Node currentTickerNode = tickerNodes.item(tickerIndex);
              if(currentTickerNode instanceof Element) {
                Element tickerTag = (Element)currentTickerNode;
                String tickerName = tickerTag.getFirstChild().getNodeValue();

                // Add/Remove the ticker to/from the list. We do not return an
                // error for non-exisitent ticker symbols.
                if(commandName.equals(addCommand)) {
                  portfolioSet.add(tickerName);
              } else { // remove
                portfolioSet.remove(tickerName);
              }
          }
        }
      } else if (commandName.equals(getCommand)) {
        Iterator tickers = portfolioSet.iterator();
          StockCore stockcore = new StockCore();
```

```
              // We loop on the ticker symbols passed in and add the value
              while (tickers.hasNext()) {
                String tickerName = (String)tickers.next();
                String tickerValue = stockcore.getQuote(tickerName);
                Element quote =
                          createQuote(responseElement.getOwnerDocument(),
                                        tickerName, tickerValue);

                // Be sure to add the quote element as a child of the command
                commandElement.appendChild(quote);
              }
            } else {
              // Unknown command, we mark it as failure
              success = false;
            }

            commandElement.setAttribute(successAttribute,
              new Boolean(success).toString());
          }
        }

        return responseElement;
      }

      /**
       * Display usage information and leave.
       *
       * @arg error Error message
       */
      private static void usage(String error) {
        System.err.println("Portfolio Client: " + error);
        System.err.println(
          " Usage          :java com.wrox.jws.stockquote.Portfolio"
          + "<command> [<ticker-symbols>]");
        System.err.println(
          "  Valid Commands: add, remove, get; get takes no ticker symbols.");
         System.err.println(
          "  Examples: java com.wrox.jws.stockquote.Portfolio get");
         System.err.println(
          "          : java com.wrox.jws.stockquote.Portfolio add sunw msft );
        System.exit(1);
      }

      /**
       * The main is a client test for the Portfolio service. It modifies the
       * portfolio with the stocks passed in as arguments using the add an
       * remove commands in the XML document submitted to server. It also gets
       * the status of the portfolio with the get command, which takes no
       * argument.
       *
       * @param args The command and the ticker symbols (e.g. sunw msft ibm)
       */
      public static void main(String [] args) {
        final String methodName ="Portfolio.main";

        try {
          // Validate the arguments
          if(args.length < 1) {
            usage("Please specify one command: add, remove, or get.");
          } else if(args[0].equals(addCommand) ||
                    args[0].equals(removeCommand)) {
            if(args.length == 1) {
              usage(
                "You need to specify at least one ticker for add and remove.");
            }
          } else if(args[0].equals(getCommand)) {
            if(args.length > 1) {
              usage("You may not specify any ticker for get.");
            }
```

```
    } else {
      usage("Unknown command: " + args[0] + ".");
    }

    // We create an XML document for the request.
    // We start by creating a parser
    DocumentBuilderFactory documentBuilderFactory = null;
    DocumentBuilder        parser                 = null;

    // We need a namespace-aware parser
    documentBuilderFactory = DocumentBuilderFactory.newInstance();
    documentBuilderFactory.setNamespaceAware(true);
    parser = documentBuilderFactory.newDocumentBuilder();

    // We create the document for the request (the XML document that will
    // be embedded inside the SOAP body).
    Document document = parser.newDocument();
    Element element   = document.getDocumentElement();

    // The <portfolio/> element is the root of our document;
    Element requestElement = document.createElementNS(nameSpaceURI,
                                    portfolioTag);
    Element command        = document.createElement(args[0]);

    requestElement.appendChild(command);

    // We add the list of ticker symbols
    for(int index = 1; index < args.length; index++) {
      Element ticker = document.createElement(tickerTag);

      ticker.appendChild(document.createTextNode(args[index]));
      command.appendChild(ticker);
    }

    // Unless the command is a get, we add the get command
      if(!args[0].equals(getCommand)) {
        requestElement.appendChild(document.createElement(getCommand));
      }

      Element result = null;
      boolean local  = false; // set to true for local test (inproc)

      if(local) {
        // Local test, no SOAP
        result = portfolio(requestElement,
           (Element)requestElement.cloneNode(true));
      } else {
        // We create a SOAP request to submit the XML document to the
        // server. You might need to replace the following URL with what is
        // suitable for your environment
        String  endpointURL =
                "http://localhost:8080/axis/servlet/AxisServlet";
        Service service     = new Service();
        Call    call        = (Call) service.createCall();

        // Create a SOAP envelope to wrap the newly formed document
        SOAPEnvelope    requestEnvelope = new SOAPEnvelope();
        SOAPBodyElement requestBody     =
                            new SOAPBodyElement(requestElement);
        requestEnvelope.addBodyElement(requestBody);

        // Set the endpoint URL (address we are talking to) and method name
        call.setTargetEndpointAddress(new URL(endpointURL));
        call.setSOAPActionURI(portfolioTag); // method name = tag name

        // Submit the document to the remote server
        SOAPEnvelope responseEnvelope = call.invoke(requestEnvelope);

        // Get the <portfolio/> element from the response
```

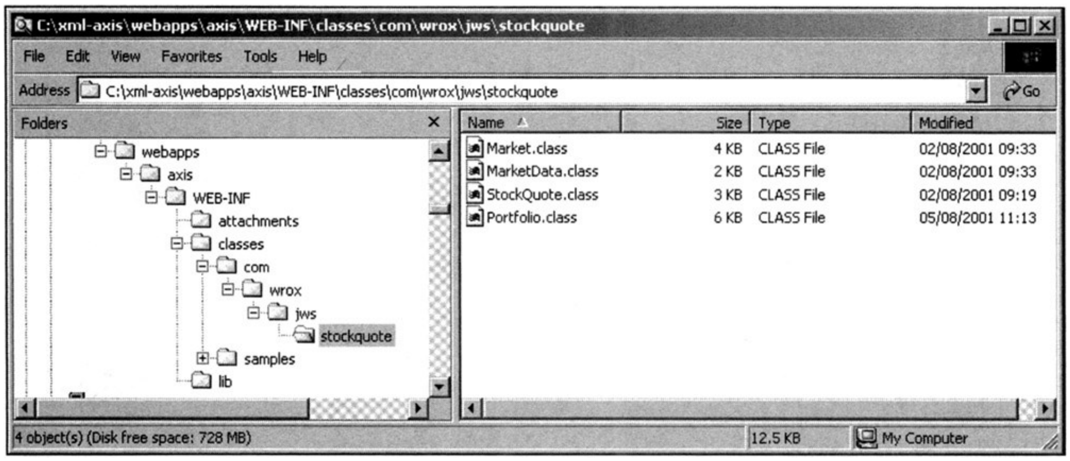
```
              SOAPBodyElement responseBody =
              (SOAPBodyElement)responseEnvelope.getBodyElements().get(0);
              result = responseBody.getAsDOM();
            }

            // Display the output
            System.out.println("Document from server: ");
            XMLUtils.PrettyElementToStream(result, System.out);
          } catch (Exception exception) {
              System.err.println(methodName + ": " + exception.toString());
              exception.printStackTrace();
          }
        }
      }
```

2. Compiling `Portfolio.java` is also similar (you can refer to Chapter 3 for a discussion about the classpath). Assuming that `src` is the current directory, the following command will compile `Portfolio.java`:

   ```
   javac -d ..\classes Portfolio.java
   ```

3. The `Portfolio.class` file needs to find its way into the `classes` directory of our Axis installation as shown in the following screenshot:



C:\xml-axis\webapps\axis\WEB-INF\classes\com\wrox\jws\stockquote

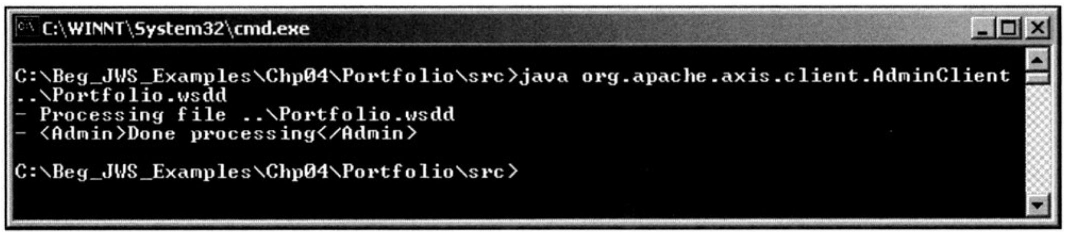
Name	Size	Type	Modified
Market.class	4 KB	CLASS File	02/08/2001 09:33
MarketData.class	2 KB	CLASS File	02/08/2001 09:33
StockQuote.class	3 KB	CLASS File	02/08/2001 09:19
Portfolio.class	6 KB	CLASS File	05/08/2001 11:13

4 object(s) (Disk free space: 728 MB) 12.5 KB My Computer

   Also, make sure the `stockcore.jar` is in the `lib` directory.

4. Now, we need to deploy the Portfolio web service:

   ```
   java org.apache.axis.client.AdminClient ..\Portfolio.wsdd
   ```



```
C:\Beg_JWS_Examples\Chp04\Portfolio\src>java org.apache.axis.client.AdminClient
..\Portfolio.wsdd
- Processing file ..\Portfolio.wsdd
- <Admin>Done processing</Admin>

C:\Beg_JWS_Examples\Chp04\Portfolio\src>
```

   The `Portfolio.main()`, our test client, takes the following commands as arguments:

   - `add`

     This command adds a stock to the portfolio. It takes one argument: the symbol of the stock to add.

   - `remove`

     This command removes a stock from the portfolio. It takes one argument: the symbol of the stock to remove.

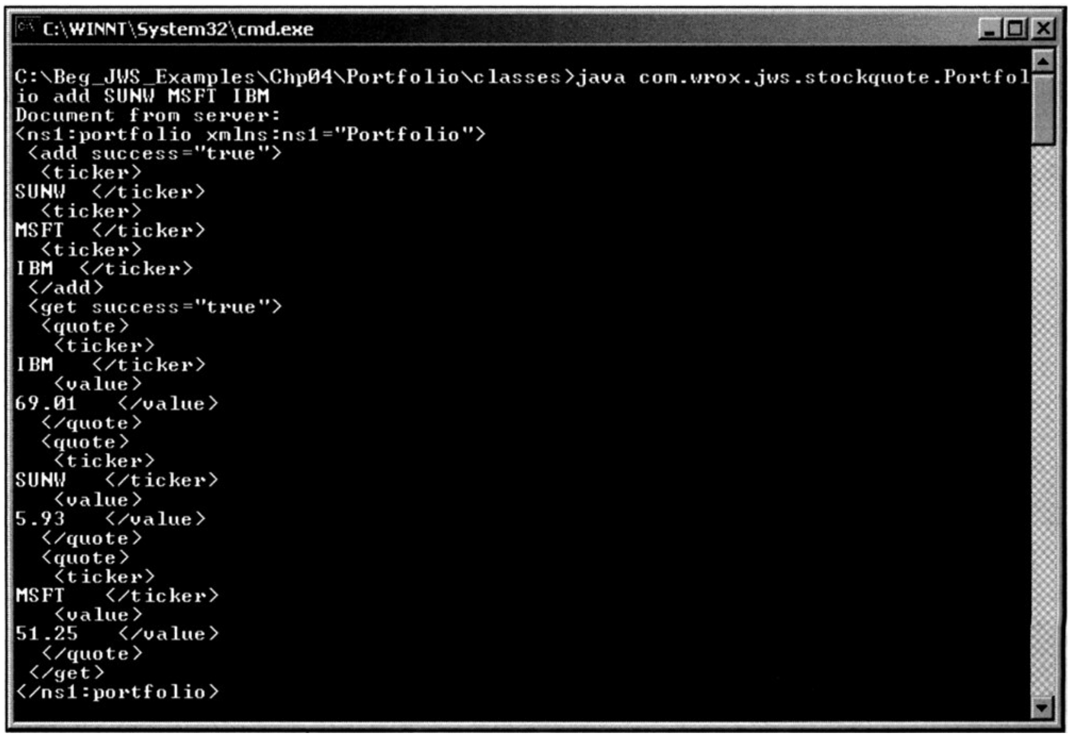
   - `get`

     This command returns the portfolio with the stock values. It takes no parameter.

5. Here is the command for adding `SUNW, MSFT`, and `IBM` to the portfolio:

```
java com.wrox.jws.stockquote.Portfolio add SUNW MSFT IBM
```

6. The following screenshot shows the results of the execution on a Windows machine:



```
C:\Beg_JWS_Examples\Chp04\Portfolio\classes>java com.wrox.jws.stockquote.Portfol
io add SUNW MSFT IBM
Document from server:
<ns1:portfolio xmlns:ns1="Portfolio">
 <add success="true">
  <ticker>
SUNW  </ticker>
  <ticker>
MSFT  </ticker>
  <ticker>
IBM  </ticker>
 </add>
 <get success="true">
  <quote>
   <ticker>
IBM   </ticker>
   <value>
69.01   </value>
  </quote>
  <quote>
   <ticker>
SUNW   </ticker>
   <value>
5.93   </value>
  </quote>
  <quote>
   <ticker>
MSFT   </ticker>
   <value>
51.25   </value>
  </quote>
 </get>
</ns1:portfolio>
```

How It Works

`Portfolio.main()` sends the following SOAP request to the server:

```
POST /axis/servlet/AxisServlet HTTP/1.0
Content-Length: 330
Host: localhost
Content-Type: text/xml; charset=utf-8
SOAPAction: "portfolio"

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<SOAP-ENV:Body>
  <nsl:portfolio xmlns:nsl="Portfolio">
    <add>
      <ticker>SUNW</ticker><ticker>MSFT</ticker><ticker>IBM</ticker>
    </add>
    <get/>
  </nsl:portfolio>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

As you can see, the request is a simple **XML** document, which is based on the specified command-line parameters. The `<get/>` element is always added by the client when the `add` or `remove` commands are given as input. This allows the user to see the effect of the command without having to resubmit an additional **XML** document to the server.

The response from the server contains the response document wrapped in a SOAP envelope and an **HTTP** response (the document has been slightly modified for presentation):

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 517
Date: Thu, 13 Jun 2002 13:45:40 GMT
```

```
Server: Apache Tomcat/4.0.3 (HTTP/1.1 Connector)

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <nsl:portfolio xmlns:nsl="Portfolio">
      <add success="true">
        <ticker>SUNW</ticker><ticker>MSFT</ticker><ticker>IBM</ticker>
      </add>
      <get success="true">
        <quote><ticker>SUNW</ticker><value>5.93</value></quote>
        <quote><ticker>MSFT</ticker><value>51.25</value></quote>
        <quote><ticker>IBM</ticker><value>69.01</value></quote>
      </get>
    </nsl:portfolio>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

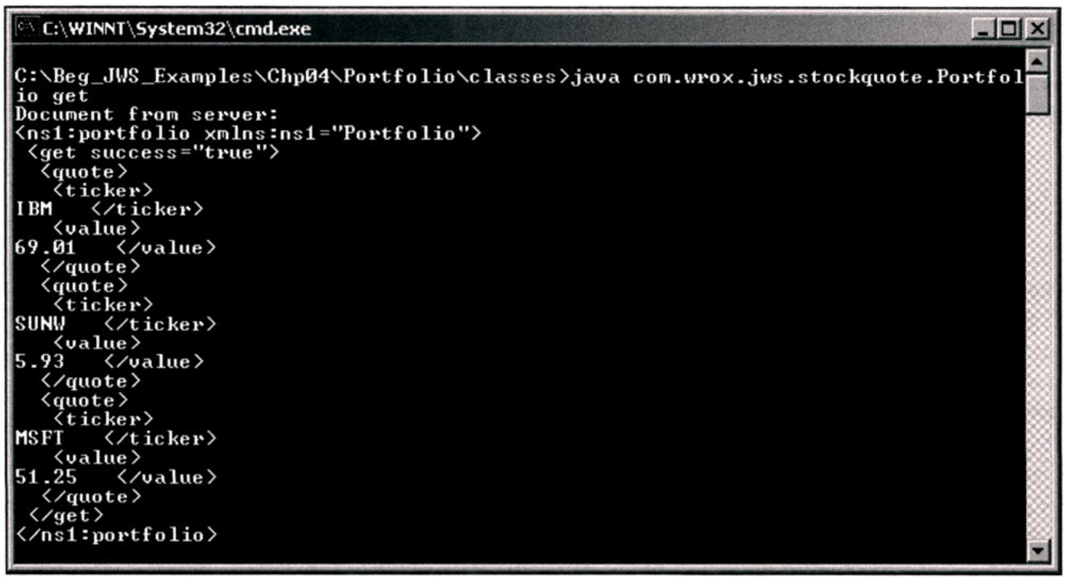
As you can see, the response document is similar to the request document. If we look at the `<add/>` and `<get/>` elements, we can see that the `Portfolio` web service added the attribute `success="true"` to inform the requester that the transaction was a success. The inner data of the `<add/>` element is identical to the request, but the `<get/>` element contains the value of the stocks we are interested in.

Note Note that in the interest of brevity, the `Portfolio` web service could have removed the inner data of the `<add/>` element. Another fact worth noting about the documents that are exchanged between the client and the server is the absence of type information; this is because of the implicit contract between the participants. The type information is optional.

The existence of a more formal contract than a simple **RPC** paradigm is what usually allows document-style **SOAP** to be more expressive and therefore reduces the amount of information that has to be exchanged for the dialog to be successful.

Note that after running the previous command, we will get the same output from the get command since the stocks have been stored in the portfolio (the value of stocks might change):

```
java com.wrox.jws.stockquote.Portfolio get
```



```
C:\Beg_JWS_Examples\Chp04\Portfolio\classes>java com.wrox.jws.stockquote.Portfol
io get
Document from server:
<ns1:portfolio xmlns:ns1="Portfolio">
 <get success="true">
  <quote>
   <ticker>
IBM   </ticker>
   <value>
69.01   </value>
  </quote>
  <quote>
   <ticker>
SUNW   </ticker>
   <value>
5.93   </value>
  </quote>
  <quote>
   <ticker>
MSFT   </ticker>
   <value>
51.25   </value>
  </quote>
 </get>
</ns1:portfolio>
```

The **XML** document for the `get` command contains only one element inside `<portfolio/>` (the **SOAP** envelope and body are not shown):

```
...
<nsl:portfolio xmlns:nsl="Portfolio"><get/></nsl:portfolio>
...
```

Let's now review the `Portfolio` code.

We use **DOM** and **JAXP** to parse the request and build the response (please refer to Chapter 2 for an introduction to **JAXP** and **DOM**):

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
```

We will rely on the Axis **API** on the client side and on the server side. We will get into the specifics shortly. The actual stock quote values will be provided by `StockCore`.

As you can see in the implementation, the `Portfolio` class contains several constants that will come in handy when we build and parse the **XML** documents. Their meaning should be self-explanatory.

The next line of code contains the declaration of the back-end data store for our portfolio. As we said previously, no mention is made of persistence. Note that since it is our only (non final) data structure that will survive from call to call, it is worth synchronizing it. This saves us from having to manually protect the statements that modify the back-end store. Also we are using a set since in our sample it does not make much sense to ask for the same stock quote more than once:

```
// Our (simplistic) portfolio
static Set portfolioSet = Collections.synchronizedSet(new HashSet());
```

The next method requires a short preamble. We have seen that in response to a `get` command, `Portfolio` returns a list of `<quote/>` elements, one for each stock in the portfolio. The `createQuote()` method creates a `<quote/>` element for a specific document. We have to pass the target document as an input argument since it is not legal to create an element with one document and insert it in another. The `createQuote()` method returns a reference to the newly created `<quote/>` element, which is now ready for insertion inside a `<get/>` element:

```
private static Element createQuote(Document document,
String tickerName, String tickerValue) {

// Create the elements and text nodes
Element quote     = document.createElement(quoteTag);
Element ticker    = document.createElement(tickerTag);
Element value     = document.createElement(valueTag);
Node nameText  = document.createTextNode(tickerName);
Node valueText = document.createTextNode(tickerValue);

// Create the hierarchy
ticker.appendChild(nameText);
value.appendChild(valueText);
quote.appendChild (ticker);
quote.appendChild (value);

return quote;
```

The next method, which is simply called `portfolio()`, is at the core of the interface between Axis and a document-style web service. You will notice that its structure is similar to what we saw in the `Admin` web service. Here are the steps performed by `portfolio()`:

1. Extract the **XML** document passed by the user from the first element of the vector. Once again, only one document is processed per request.

2. Create the response document. In this case, we simply clone the request since the schema of the request is (very) similar to the schema of the response.

3. Call an overloaded version of the `portfolio` method, which returns an `Element`.

4. Wrap the `Element` inside an array and return it to Axis for inclusion inside a **SOAP** envelope.

The code structure is as follows:

```
public static Element [] portfolio (Vector xmlDocument) throws Exception {

  Document requestMessage =
    ((Element) xmlDocument.get (0)).getOwnerDocument();
  Document responseMessage = (Document) requestMessage.cloneNode (true);

  Element [] result = new Element [1];
  result [0] = portfolio (requestMessage.getDocumentElement(),
  responseMessage.getDocumentElement());

  return result;
}
```

The next method is a private version of `portfolio()` that takes two arguments – the element of the request document and the element of the response document. Its return value is superfluous since it returns the (hopefully modified) request element:

```
private static Element portfolio (Element requestElement,
  Element responseElement) throws Exception {
```

The algorithm implemented by `portfolio` is straightforward:

- Traverse the child elements of `<portfolio/>` in the response document, which at the beginning, is identical to the request document.
- Process commands as they are encountered and add the necessary child elements (`<quote/>`) and create a `success` (Boolean) attribute.

The following loop traverses each command:

```
      // We traverse the document and process each command as we go
      NodeList nodes = responseElement.getChildNodes();
      for (int nodeIndex = 0; nodeIndex < nodes.getLength(); nodeIndex++) {
        Node currentNode = nodes.item (nodeIndex);
```

For each command, we test if it is an `Element`. In other words, we ignore any text inside `<portfolio/>` and outside the children elements.

```
        if (currentNode instanceof Element) { // Ignore anything else

          Element commandElement = (Element) currentNode;
          String commandName = commandElement.getTagName();
          boolean success = true;
```

The `add` and `remove` commands are handled in the same block of code, as their processing is similar. For this, we need to extract the ticker symbol from the `<ticker/>` element and modify the portfolio data structure with a `Set.add()` and `Set.remove()` respectively:

```
            if(commandName.equals (addCommand) ||
                 commandName.equals (removeCommand)) {
              // We get the list of ticker symbols to add/remove
              NodeList tickerNodes = commandElement.getChildNodes();
              for (int tickerIndex = 0; tickerIndex < tickerNodes.getLength();
                   tickerIndex++) {
                Node currentTickerNode = tickerNodes.item (tickerIndex);
                if (currentTickerNode instanceof Element) {
                  Element tickerTag = (Element) currentTickerNode;
                  String tickerName = tickerTag.getFirstChild() .getNodeValue();

                  // Add/Remove the ticker to/from the list. We do not return an
                  // error for non-exisitent ticker symbols.
                  if (commandName.equals (addCommand)) {
                    portfolioSet.add(tickerName);
                  } else { // remove
                    portfolioSet.remove(tickerName);
                  }
                }
              }
```

To process the `get` command, we simply loop over the ticker symbols defined in the portfolio, get the quote using `StockCore`, and add a `<quote/>` element with the help of the `createQuote()` method that we reviewed earlier:

```
            } else {
              if(commandName.equals(getCommand)) {
                  Iterator tickers = portfolioSet.iterator();

                  // We loop on the ticker symbols passed in and add the
                  //value
                  while(tickers.hasNext()) {
                    String tickerName = (String)tickers.next();
                    String tickerValue = StockCore.getQuote(tickerName);
                    Element quote =
                    createQuote(responseElement.getOwnerDocument(),
                    tickerName, tickerValue);

                    // Be sure to add the quote element as child of the
                    // command
                    commandElement.appendChild(quote);
                  }
                } else { // Unknown command, we mark it as failure
                  success = false;
                }
```

Any other command is unknown so we mark it with `success"false"`. The statement below sets the success attribute to the value held by the variable `success` (`true` or `false`):

```
            commandElement.setAttribute(successAttribute,
              new Boolean(success).toString());
```

```
        }
      }

      return responseElement;
    }
```

This completes the definition of the `Portfolio` web service. The remainder of the quote contains the implementation of the test client.

The `usage()` method that displays help about the command-line arguments is self-explanatory:

```
    private static void usage (String error) {
      System.err.println("Portfolio Client: " + error);
      System.err.println(
        " Usage          : java com.wrox.jws.stockquote.Portfolio "
        + "<command> [<ticker-symbols>]");
      ...
```

The main task of the client is to build the **XML** document that constitutes the request. We will, once again, use **DOM** and its Xerces implementation to achieve this objective.

The beginning part of the `main()` method simply collects the arguments from the user and exits with an error message, in case it does not like what was passed to it.

Things start getting more appealing when we begin to build the document. As we can see below, we need to have a namespace-aware parser (this is not the default). This requirement comes from our `<portfolio/>` that is qualified with the `Portfolio` namespace (the full request was listed earlier) - `<nsl:portfolio xmlns:nsl="Portfolio">`.

```
      // We create an XML document for the request. We start by creating
      // a parser
      DocumentBuilderFactory documentBuilderFactory = null;
      DocumentBuilder        parser                 = null;

      // We need a namespace-aware parser
      documentBuilderFactory = DocumentBuilderFactory.newInstance();
      documentBuilderFactory.setNamespaceAware(true);
      parser = documentBuilderFactory.newDocumentBuilder();
```

Armed with a parser, we can now build a document. Things are not as easy as they were in the server case, as we do not have a document that we can morph into the request. Note that it would be a valid alternative design to start with two literal documents for add and remove and modify them to add the ticker symbols passed as arguments. In that implementation, we would read the documents into a **DOM** and transform them using the `org.w3c.dom` **API** as we did for the web service.

After creating a document, we get a hold of its element and use it to append the command element (`<add/>or<remove/>`) that we create using the document:

```
      // We create the document for the request (the XML document that
      // will be embedded inside the SOAP body).
      Document document = parser.newDocument();
      Element element   = document.getDocumentElement();

      // The <portfolio/> element is the root of our document;
      Element requestElement =
        document.createElementNS(nameSpaceURI, portfolioTag);
      Element command        = document.createElement(args[0]);
      requestElement.appendChild(command);
```

Our next task is to create the list of `ticker` symbols for the command. This code should look familiar since it is similar to what we did on the server side, when we created the list of tickers based on the portfolio data structure:

```
      // We add the list of ticker symbols
      for (int index = 1; index < args.length; index++) {
        Element ticker = document.createElement(tickerTag);
        ticker.appendChild(document.createTextNode(args[index]));
        command.appendChild(ticker);
      }

      // Unless the command is a get, we add the get command
      if (!args[0].equals(getCommand)) {
        requestElement.appendChild(document.createElement(getCommand));
      }
```

As we mentioned earlier and as seen in the statement above, we also add a `<get/>` element (command), unless the user gave us a `get` command to begin with.

If we look below, we will see that we have a `boolean` variable to differentiate between a local test and a remote test. This is useful to save time during development and easy to achieve thanks to our private version of portfolio:

```
Element result = null;
boolean local = false; // set to true for local test (inproc)

if(local) {
  // Local test, no SOAP
  result = portfolio(requestElement,
  (Element)requestElement.cloneNode(true));
} else {
```

The case that requires our attention is the remote case. This code is not very different from the `StockQuote` client that we developed in Chapter 3. The main difference is that we add the request element, `<portfolio/>`, to the SOAP body via the Axis `SOAPBodyElement`:

```
// We create a SOAP request to submit the XML document to the
// server. You might need to replace the following URL with
// what is suitable for our environment
String endpointURL =
  "http://localhost:8080/axis/servlet/AxisServlet";
Service service = new Service();
Call call = (Call)service.createCall();

// Create a SOAP envelope to wrap the newly formed document
SOAPEnvelope    requestEnvelope = new SOAPEnvelope();
SOAPBodyElement requestBody     =
  new SOAPBodyElement(requestElement);
requestEnvelope.addBodyElement(requestBody);

// Set the endpoint URL (address we are talking to) and
// method name
call.setTargetEndpointAddress(new URL(endpointURL));
call.setSOAPActionURI(portfolioTag); // method name = tag name
```

In the document-style version of a SOAP call, we pass the envelope to the `Call` object and we get an envelope back as a result, or an exception when things go awry:

```
// Submit the document to the remote server
SOAPEnvelope responseEnvelope = call.invoke(requestEnvelope);
```

We extract the `<portfolio/>` element from the response. Once again, it would be legal to have a response with more than one root element. We then assign the **DOM** of the response to the result variable. The remainder of the code displays the DOM and handles any exception thrown our way:

```
// Get the <portfolio/> element from the response
SOAPBodyElement responseBody =
  (SOAPBodyElement)responseEnvelope.getBodyElements().get(0);
result = responseBody.getAsDOM();
...
```

The deployment descriptor for `Portfolio, Portfolio.wsdd` looks a lot like `StockQuote.wsdd` that we introduced in the previous chapter:

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
<service name="Portfolio" provider="java:MSG">
  <parameter name="className"
    value="com.wrox.jws.stockquote.Portfolio"/>
  <parameter name="allowedMethods" value="portfolio"/>
</service>
</deployment>
```

Note To deploy a document-style web service as a full message web service, add the `FullMessageService` parameter to our deployment descriptor `<parameter name="FullMessageService" value="true"`.

Aside from the class name and the allowed method, the provider is different – `provider="java:MSG"`. In the case of RPC, the provider was `provider="java:RPC"`. The `java:MSG` attribute value tells Axis to instantiate an `org.apache.axis.providers.java.MsgProvider` object. The main task of the `MsgProvider.processMessage()` method, the only method defined in `MsgProvider`, is to instantiate our class, extract the document from the SOAP envelope, and pass it to our method. In the case of a full message service, the SOAP envelope is passed as-is.

This concludes our review of the `Portfolio` class. Before closing this chapter on document-style SOAP development, it is worth spending a few lines discussing our XML parsing strategy.

SAX versus DOM

In the `Portfolio` example, both on the client and server side, we manipulated the XML document using DOM, which provided us with an easy-to-use interface. However, this facility comes with one noteworthy disadvantage – the entire document must be loaded in memory. If the document is very large, this requirement might significantly hamper the performance of our application.

The preferred way to handle large documents is to use the Simple API for XML (SAX) that we covered in Chapter 2. The methodology used by SAX is radically different from that of DOM – we register a callback class that implements the `org.xml.sax.DocumentHandler` interface, and, for document-related events, methods of this interface get called by the parser (for example, the beginning or the end of an element).

A similar methodology could be implemented as part of Axis, where we would get a callback function with a stream as input argument and a stream as output argument. Alas, at the time of this writing, such support does not exist in Axis. If our application needs to handle large XML documents, our best bet is to write a servlet that uses SAX to parse the requests and document fragments.

Summary

Document-style SOAP development differs from RPC-style in a significant way. In the RPC model, the client is bound to the procedure call paradigm, whereas in the document-style model, the client is bound to what can be expressed through an XML document. This freedom requires a contract that defines the semantics of the requests and responses.

Practically, the implementation of a document-style web service requires the implementation of a callback method that uses the DOM for its input arguments and return value. We first looked at the structure of the `Admin` web service to see how a document-style web service interfaces with Axis. We then wrote a modified version of the `StockQuote` example, `Portfolio`, which used the document-style model as an interface to a simplistic portfolio management application. We also made extensive use of the DOM to create and manipulate the request and the response.

We concluded the chapter by briefly discussing the drawbacks of using the DOM and mentioned a SAX-based servlet as an alternative.