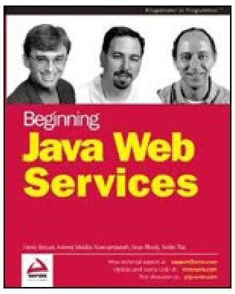


Beginning Java Web Services

by Henry Bequet

Apress. (c) 2002. Copying Prohibited.

Reprinted for Venkata Kiran Polineni, Verizon

kiran2chotu@gmail.com

Reprinted with permission as a subscription benefit of **Skillport**,
http://skillport.books24x7.com/

All rights reserved. Reproduction and/or distribution in whole or in part in electronic,paper or other forms without written permission is prohibited.

Chapter 5: Describing Web Services

Highlights

One of the major goals of web services is to provide a framework for electronic commerce (e-commerce) or electronic business (e-business). This framework is made up of official standards such as HTTP, HTML, and XML. In this context, 'official' implies international organizations such as the **World Wide Web Consortium** (W3C) with enough clout to entice developers to use them. The framework is also made of standards that have been proposed to organizations such as the W3C. SOAP, which we have spent some time describing in the first part of this book, is one such standard.

As we saw in the earlier chapters, SOAP has been designed to encode **Remote Procedure Calls** (RPC) or XML documents for their transit over a network. **Interoperability** or encoding data to be processed by remote computers is not the goal, but mandatory for successful e-commerce. The description and publication capabilities of a web service are as important as the actual processing of the data and its encoding.

To ensure a wide acceptance, the description of web services must conform to some standard. Without some agreement among buyers and sellers of web services, it would be close to impossible for a consumer to compare the services of two or more providers. The criteria that immediately come to mind, are name, description, and price.

Apart from this, the protocols used for calling a web service are equally important. Does the service support SOAP or is it simply accessible via HTTP GET? Does the service only support HTTP or does it support SMTP as well? These questions are fundamental when it comes to choosing one web service over another. It would do us no good to select a web service based on a better price if that service did not support the only protocol we are familiar with.

Usually, software vendors provide some textual description of what their products are capable of achieving. This human-readable description of a product is poorly suited for automated processing. A more formal description is needed for computer programs to make sense of it. It is also important to remember that we do not want to forgo the benefits of a human-readable description. The **Web Services Description Language (WSDL)** is an attempt at addressing this dual need.

Important In a nutshell, WSDL defines an XML dialect to describe the capabilities of a web service.

The use of XML is a reasonable balance between a machine-readable and a human-readable document.

We hinted earlier that a description is necessary but not sufficient. As a web service developer, we would be hard pressed to sell our components if we had no place to advertise them. The UDDI specification proposes an answer to this. We will discuss UDDI and other publishing technologies available to web services in Chapter 7.

In this chapter:

- We will start with a high-level view of the specification of a web service. Here, we will describe the WSDL top-level elements.
- We will then go from theory to practice by looking at a WSDL document that describes our simple `StockQuote` web service.
- We will then look into a complex version of this web service, which uses more complex data structures such as arrays and custom data types.
- We will also review a brief example that shows how client-side development is eased with the use of WSDL.

Let's begin with an overview of WSDL.

WSDL Overview

Note The specification for WSDL can be found at http://www.w3.org/TR/wsdl.

As we mentioned previously, WSDL provides a description of a web service in terms of **endpoints** and the **messages** traveling to and from the endpoints. A more intuitive way of looking at a WSDL document is to state that it answers four questions about a web service:

- **What do you do?**

  This question is answered both in machine-readable and human-readable forms. The human-readable answers can be found in the `<name/>` and `<documentation/>` elements. The answer in machine-readable terms comes from the `<message/>` and the `<portType/>` elements.

- **What language do you speak?**

  What data types do you use? This question is answered using the `<types/>` elements.

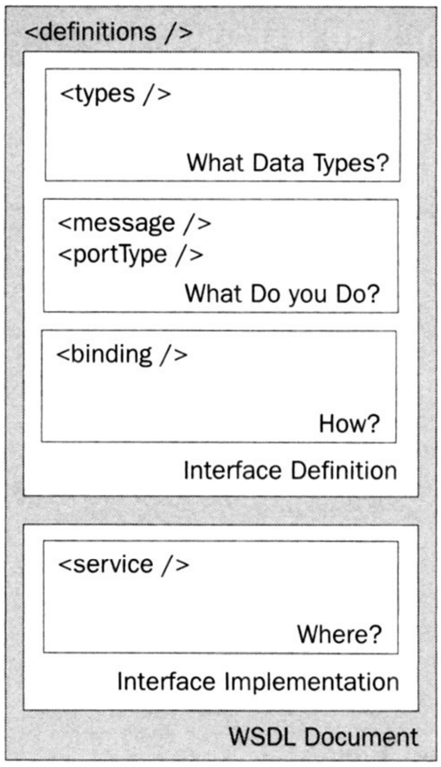
- **How do I talk to you?**

  How does a client talk to the service? HTTP? SMTP? This question is answered using the `<binding/>` elements.

- **Where do I find you?**

  Where can I find this web service? What is its URL? The answer is in the `<service/>` elements.

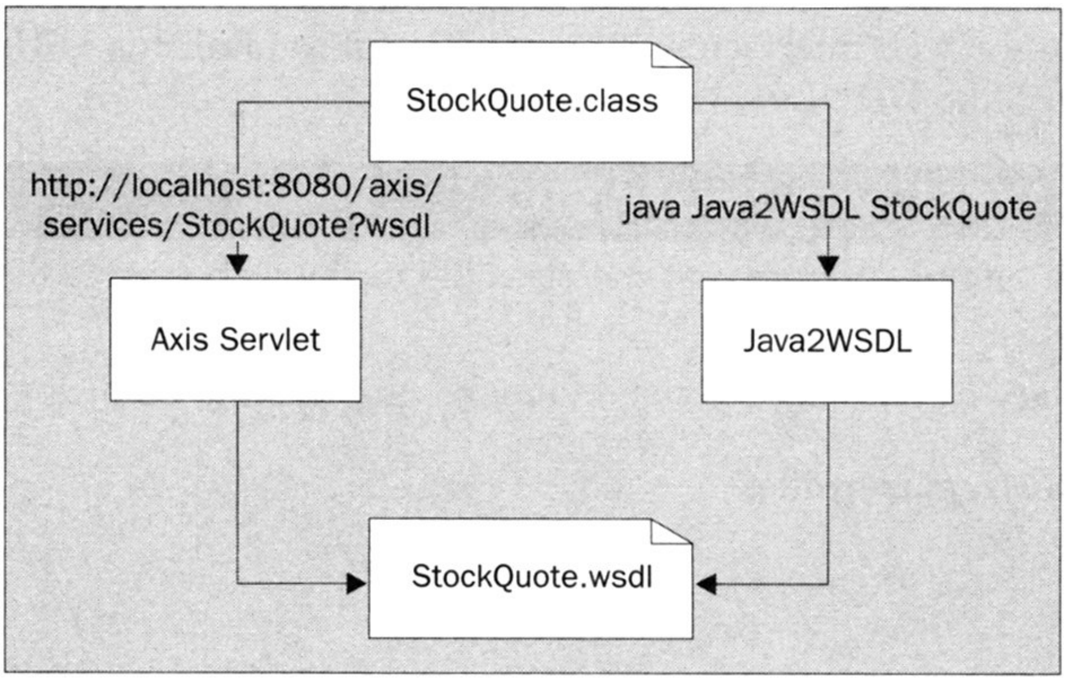
The following figure shows a high-level picture of the WSDL specification:



Instead of describing each WSDL element in abstract terms, we will look at the WSDL that describes the `StockQuote` web service developed in Chapter 3 (the instructions for building, deploying, and testing `StockQuote` can be found in Chapter 3 along with the code).

A WSDL Document for StockQuote

Instead of writing a WSDL document from scratch, we will use the tools provided by Axis to generate the WSDL document from our Java class as shown below:

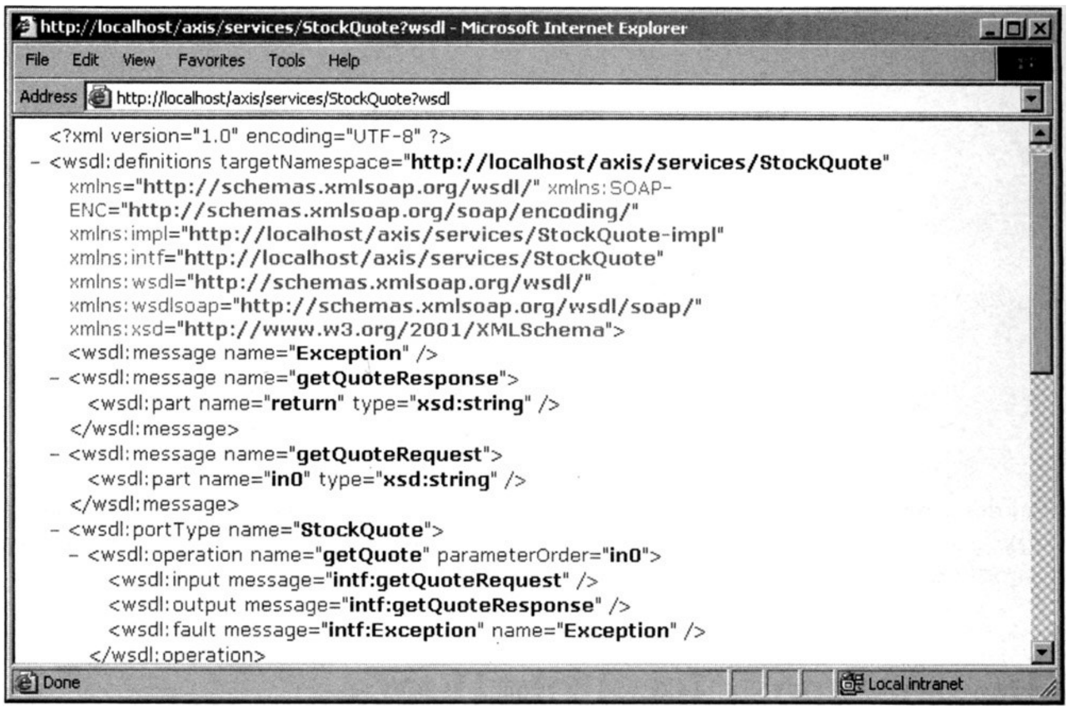

StockQuote.class
http://localhost:8080/axis/services/StockQuote?wsdl
java Java2WSDL StockQuote
Axis Servlet
Java2WSDL
StockQuote.wsdl

Reprinted for MC2WS/venkata.polineni@verizon.com, Verizon
Apress, Wrox Press (c) 2002, Copying Prohibited

As we can see from the diagram, there are two ways to generate a WSDL document from an existing web service. We will be looking at both methods as they each have their pros and cons. Let's start with the Axis servlet.

Try It Out: Generate WSDL Using the Axis Servlet

The easiest way to generate a WSDL document for a web service deployed with Axis is to navigate to the http://<server-name>/services/<service-name>?wsdl URL as shown in the screenshot:


http://localhost/axis/services/StockQuote?wsdl - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/axis/services/StockQuote?wsdl
Done
Local intranet

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <wsdl:definitions targetNamespace="http://localhost/axis/services/StockQuote"
    xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:SOAP-
    ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:impl="http://localhost/axis/services/StockQuote-impl"
    xmlns:intf="http://localhost/axis/services/StockQuote"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:message name="Exception" />
  - <wsdl:message name="getQuoteResponse">
      <wsdl:part name="return" type="xsd:string" />
    </wsdl:message>
  - <wsdl:message name="getQuoteRequest">
      <wsdl:part name="in0" type="xsd:string" />
    </wsdl:message>
  - <wsdl:portType name="StockQuote">
    - <wsdl:operation name="getQuote" parameterOrder="in0">
        <wsdl:input message="intf:getQuoteRequest" />
        <wsdl:output message="intf:getQuoteResponse" />
        <wsdl:fault message="intf:Exception" name="Exception" />
      </wsdl:operation>
```

How It Works

When the Axis servlet detects the `?wsdl` trailer on a valid web service URL, it generates the WSDL based on the class file of the service. The URL trailer is not case sensitive, so `?WsDl` works just as well. This built-in support in the web interface will be very useful to publish a URL for the description of a web service in a registry like UDDI.

Try It Out: Generate WSDL with Java2WSDL

In this second method, one needs access to the class file of the web service to use the `Java2Wsdl` tool.

1. Prior to running `Java2Wsdl` , we need to make sure that the following files are in our classpath:

   The Xerces (XML parser) files:

   ```
   xercesImpl.jar
   xmlParserAPIs.jar
   ```

   The JAR files distributed with Axis (see Chapter 3 for more details):

   ```
   axis.jar
   commons-logging.jar
   tt-bytecode.jar
   jaxrpc.jar
   ```

   The WSDL for Java package contains the WSDL to Java and Java to WSDL classes that we use in this chapter. It comes with Axis and can be found in the `lib` directory with the other JAR files distributed with Axis:

   ```
   wsd14j.jar
   ```

   The `.class` file (not the `.java` file) of the `StockQuote` example that we developed in Chapter 3.

   ```
   StockQuote
   ```

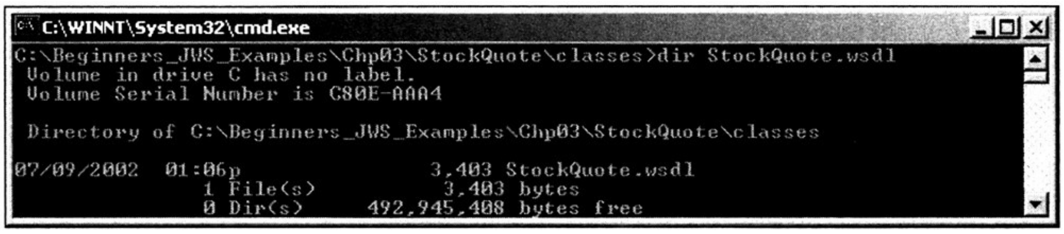
2. Here is the `Java2WSDL` command for creating the `StockQuote.wsdl` file:

   ```
   java org.apache.axis.wsdl.Java2WSDL com.wrox.jws.stockquote.StockQuote
   -1 http://localhost:8080/axis/services/StockQuote
   ```

3. Now we can check whether the `StockQuote.wsdl` file has been created:

Reprinted for MC2WS/venkata.polineni@verizon.com, Verizon

Apress, Wrox Press (c) 2002, Copying Prohibited


C:\WINNT\System32\cmd.exe

```
C:\Beginners_JWS_Examples\Chp03\StockQuote\classes>dir StockQuote.wsdl
 Volume in drive C has no label.
 Volume Serial Number is C80E-AAA4

 Directory of C:\Beginners_JWS_Examples\Chp03\StockQuote\classes

07/09/2002  01:06p              3,403 StockQuote.wsdl
               1 File(s)          3,403 bytes
               0 Dir(s)     492,945,408 bytes free
```

Be sure to indicate the actual location (`-1 http://...`) of the service since it is not available in the class file. The `Java2Wsdl` supports more options, which we can see by running the class without any command-line arguments.

Regardless of the methodology that we use to generate the WSDL document, the result will be the same as shown below. It would be a good idea to take a few moments to go through that document and compare it to the high-level block diagram shown earlier in this chapter. Do not focus on the details for now; have a look at the elements and their high-level meanings as stated in the block diagram:

```
    <?xml version="1.0" encoding="UTF-8"?>

    <wsdl:definitions
      targetNamespace="http://localhost/axis/services/StockQuote"
      xmlns="http://schemas.xmlsoap.org/wsdl/"
      xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:impl="http://localhost/axis/services/StockQuote-impl"
      xmlns:intf="http://localhost/axis/services/StockQuote"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <wsdl:message name="Exception"/>

      <wsdl:message name="getQuoteRequest">
        <wsdl:part name="in0" type="xsd:string"/>
      </wsdl:message>
      <wsdl:message name="getQuoteResponse">
        <wsdl:part name="return" type="xsd:string"/>
      </wsdl:message>

      <wsdl:portType name="StockQuote">
        <wsdl:operation name="getQuote" parameterOrder="in0">
          <wsdl:input message="intf:getQuoteRequest"/>
          <wsdl:output message="intf:getQuoteResponse"/>
          <wsdl:fault message="intf:Exception" name="Exception"/>
        </wsdl:operation>
      </wsdl:portType>

      <wsdl:binding name="StockQuoteSoapBinding" type="intf:StockQuote">
        <wsdlsoap:binding style="rpc"
          transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="getQuote">
          <wsdlsoap:operation soapAction=""/>
          <wsdl:input>
            <wsdlsoap:body
              encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              namespace="http://localhost/axis/services/StockQuote"
              use="encoded"/>
          </wsdl:input>

          <wsdl:output>
            <wsdlsoap:body
              encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              namespace="http://localhost/axis/services/StockQuote"
              use="encoded"/>
          </wsdl:output>
        </wsdl:operation>
      </wsdl:binding>

      <wsdl:service name="StockQuoteService">
        <wsdl:port binding="intf:StockQuoteSoapBinding" name="StockQuote">
          <wsdlsoap:address
```

Reprinted for MC2WS/venkata.polineni@verizon.com, Verizon Apress, Wrox Press (c) 2002, Copying Prohibited

```
            location="http://localhost/axis/services/StockQuote"/>
        </wsdl:port>
      </wsdl:service>
    </wsdl:definitions>
```

We shall now explain the different elements in this WSDL document section by section.

The definitions Element

The root of every WSDL document is the `<definitions/>` element. In other words, a WSDL document is simply a list of definitions. As we can see from the document, the `<definitions/>` element defines several namespaces and each namespace corresponds to:

```
targetNamespace="http://localhost/axis/services/StockQuote"
```

The `targetNamespace` is the namespace of our web service. When we look at the `<types/>` element, we will see that it allows the **XML Schema** contained in a WSDL document to refer to itself. As is always the case with namespaces, a document does not necessarily exist at the specified URL.

Note XML Namespaces and Schema were introduced in Chapter 2.

The default namespace of our document is http://schemas.xmlsoap.org/wsdl/:

```
xmlns="http://schemas.xmlsoap.org/wsdl/"
```

It is the namespace used to reference WSDL elements, like `<definitions/>` or `<portType/>`. Note that in our example, the default namespace is redefined with the prefix `wsdl`. In other words, `<portType/>` is equivalent to `<wsdl:portType/>` in this document. The namespace used to describe SOAP encoding is http://schemas.xmlsoap.org/soap/encoding:

```
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
```

For example, if we wanted to specify that an array should be encoded using SOAP encoding, we would qualify the encoding with (`SOAP-ENC:Array`). We will see an example of array encoding later in the chapter.

The next attributes define the implementation and definition namespaces for `StockQuote`:

```
xmlns:impl="http://localhost/axis/services/StockQuote-impl"
xmlns:intf="http://localhost/axis/services/StockQuote"
```

Conceptually, a web service can be separated into two WSDL files:

- Document for the definition of the interface of the web service (elements `<types/>`, `<message/>`, `<portType/>`, and `<bindings/>`)
- Document for the implementation of the web service (element `<service/>`)

For simplicity, we will use only one file here, and consequently we will only work with the http://localhost/axis/services/StockQuote namespace, prefixed with `intf`:

```
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

The remaining two namespaces are for SOAP elements (`<body/>`) and XML Schema definition elements (`<xsd:string/>`).

Note that the `<definitions/>` element can also define the name of the service via the `name` attribute:

```
...
<wsdl:definitions
  name="StockQuote"
  targetNamespace="http://localhost/axis/services/StockQuote"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
...
```

However, the `Java2WSDL` class does not generate that attribute.

Let's conclude this review of the `<definitions/>` root element by mentioning that WSDL also supports a `<document/>` element for the definition of a human-readable description of the web service. For instance, the following modification of our WSDL document is valid:

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions
...
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:document>
    The StockQuote service provides near real-time stock quote
    at a very affordable price.
  </wsdl:document>
```

Reprinted for MC2WS/venkata.polineni@verizon.com, Verizon

Apress, Wrox Press (c) 2002, Copying Prohibited

```
...
```

The message Element

The `<message/>` element gives information about the data that travels from one endpoint to another. More specifically, the `<message/>` element depicts a **one-way message** (more on this later). To describe an RPC method call, one needs an **input message** and an **output message.** This becomes obvious if we look at the WSDL generated by Axis:

```
...
<wsdl:message name="getQuoteRequest">
  <wsdl:part name="in0" type="xsd:string"/>
</wsdl:message>

<wsdl:message name="getQuoteResponse">
  <wsdl:part name="return" type="xsd:string"/>
</wsdl:message>
...
```

The `getQuote()` method is described in terms of a request (the method call) and a response (the return value). Messages are grouped into operations that can be compared to Java methods, and these operations are grouped into port types that can be compared to Java classes.

The name of the `<message/>` element simply provides a unique identifier (within the document) for the one-way message. Each argument of the call is described with the `<part/>` element. Like all WSDL elements, it can be qualified by a unique name. To have `Java2WSDL` generate part names with meaningful names rather than the `in0, in1` that we have in our document, we can use a class file compiled with debug information (that is by using `javac -g` rather than `javac`). For instance, `in0` would be replaced by `ticker`:

```
...
<wsdl:message name="getQuoteRequest">
  <wsdl:part name="ticker" type="xsd:string"/>
</wsdl:message>
...
```

The most important attribute here is the `type` attribute; it specifies an XML Schema data type. More precisely, the value of the `type` attribute must be a namespace-qualified XML Schema element, also known as **QName**. In this particular case, we use the `xsd` (XML Schema Definition) prefix, which refers to http://www.w3.org/2001/XMLSchema, the XSD namespace. We can also use our own data type definitions.

Note Beware of tools that are still using older XML schema namespaces like http://www.w3.org/1999/XMLSchema, because they might be the source of interoperability problems. Typically the solution is to manually edit the WSDL file and replace the older namespace with the 2001 version.

The `getQuote()` method defines only one argument. Consequently, the `<message/>` element in our WSDL document contains only one `<part/>` element. If the method signature contains multiple elements, we can simply specify multiple `<part/>` elements. For instance, consider the following variation of `getQuote()`:

```
public void getQuotes(String ticker1, String ticker2);
```

It would be described by the following WSDL `<message/>` element:

```
  <wsdl:message name="getQuoteRequest">
    <wsdl:part name="ticker1" type="xsd:string"/>
    <wsdl:part name="ticker2" type="xsd:string"/>
  </wsdl:message>
```

Let's see how messages can be grouped to form **port types** and **operations.**

The portType Element

The `<portType/>` element describes and defines the operations (or methods) supported by the web service. Operations can have `input messages`, `output messages`, and `fault messages`. In our example, we have an RPC call that has one `input message`, one `output message`, and one `fault message`:

```
  ...
  <wsdl:portType name="StockQuote">
    <wsdl:operation name="getQuote" parameterOrder="in0">
      <wsdl:input message="intf:getQuoteRequest"/>
      <wsdl:output message="intf:getQuoteResponse"/>
      <wsdl:fault message="intf:Exception" name="Exception"/>
    </wsdl:operation>
  </wsdl:portType>
  ...
```

Note that the `input`, `output`, and `fault` messages must specify a message that is defined elsewhere in the WSDL document (for example `getQuoteRequest()`). As mentioned before, the `<portType/>` element can roughly be compared to the concept of a Java class. It would

Reprinted for MC2WS/venkata.polineni@verizon.com, Verizon

Apress, Wrox Press (c) 2002, Copying Prohibited

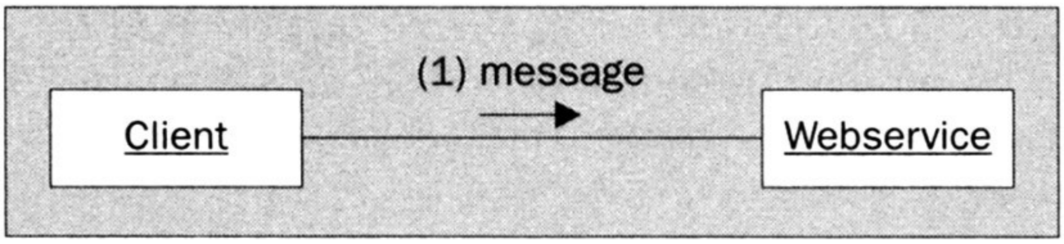
be more correct to compare a port type to an interface since a port type only defines operations (methods), and not instance data.

> **Note** The order of definitions is not relevant in a WSDL document. For instance, we can define an operation that refers to a message defined later in the document.

WSDL supports four modes of operations:

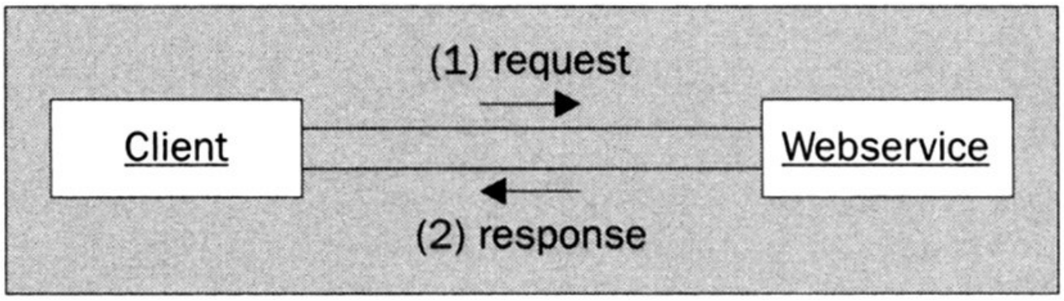
- **One-way**

  A message is sent to an endpoint of the service. For instance, a client sends a message to a web service via SMTP or calls a method with a `void` return type over HTTP. In this case, only an input message appears in the operation:



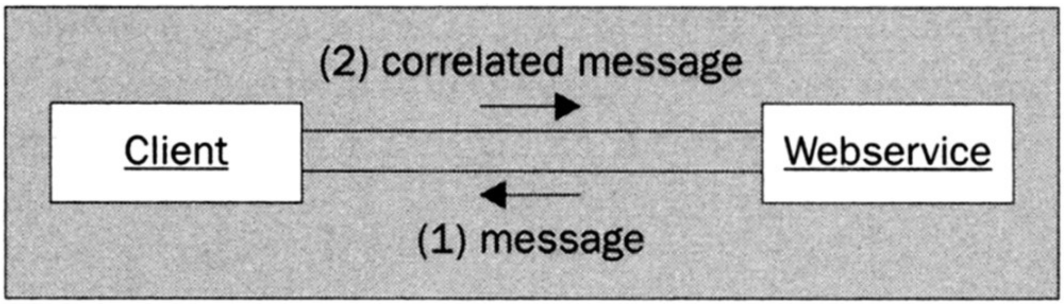
- **Request-Response**

  This mode of operation is the most common. In this case, an `input`, an `output`, and an optional `fault` element appear in the operation. A non-void method call like `getQuote()` is an example of a request-response operation:



- **Solicit-Response**

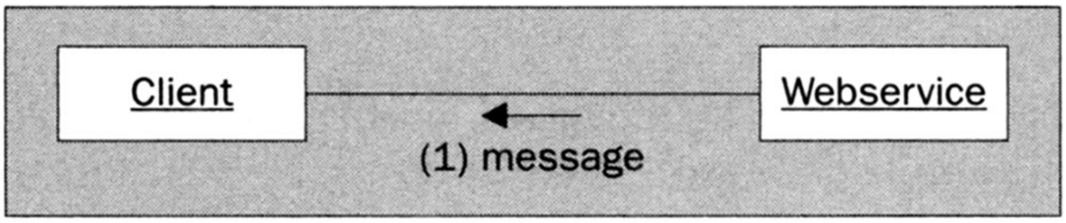
  In this transmission mode, the endpoint is the client of another endpoint. The format of the operation is similar to the request-response mode, but the output is listed before the input. A possible use of this mode is an advertisement to which different clients can send different responses:



- **Notification**

  This mode is another version of the one-way transmission primitive where the endpoint sends the message rather than receives it. In this case, the operation only contains an output message. This is similar to event notification in GUI development:

Reprinted for MC2WS/venkata.polineni@verizon.com, Verizon Apress, Wrox Press (c) 2002, Copying Prohibited



So far we have only answered one of the four questions that we identified earlier – What do we do? We know what a web service does, the messages that can be exchanged between the clients and the endpoints, to form operations that can be further aggregated into port types.

At this point, we still do not know how the operations are going to travel across the wire. How will RPC calls be encoded? Will they use SOAP? Will they be simple HTTP GETs? Answering the 'how' of a web service is the essence of the `<binding/>` element that we introduce in the next section.

The binding Element

The `<binding/>` element brings the discussion to a more practical level. It describes how the operations defined in a port type are transmitted over the network. Since it applies to a port type, a `<binding/>` element must specify a port type defined somewhere else in the document. In our example, the binding refers to the only port type that we have defined – `StockQuote`:

```
      <wsdl:binding name="StockQuoteSoapBinding" type="intf:StockQuote">
```

The value of the `name` attribute specifies a unique identifier for this binding. The next element, `<wsdlsoap:binding/>` belongs to the http://schemas.xmlsoap.org/wsdl/soap namespace, and is part of the WSDL specification. Elements in that namespace define a **SOAP extension** to WSDL, which allows one to define the details of SOAP elements like the body and the envelope right in the WSDL file.

The `<wsdlsoap:binding/>` element indicates that the binding describes how the `StockQuote` port type will be accessed via SOAP:

```
    <wsdlsoap:binding style="rpc">
      transport="http://schemas.xmlsoap.org/soap/http"/>
```

The value of the style attribute indicates that the SOAP message will follow the Remote Procedure Call (RPC) format – a request followed by a response or a fault if things go badly. More specifically, a call to `getQuote()` will be encoded with a SOAP envelope and a SOAP body as in the following example:

```
    <?xml version="1.0" encoding="UTF-8"?>

    <SOAP-ENV:Envelope
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      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"
      xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

      <SOAP-ENV:Body>
        <ns1:getQuote xmlns:ns1="StockQuote">
          <ticker xsi:type="xsd:string">sunw</ticker>
        </ns1:getQuote>
    </SOAP-ENV:Body>

     </SOAP-ENV:Envelope>
```

In our WSDL document, the `transport` attribute indicates the SOAP transport that will be used. In this case, we use HTTP. Another valid value for indicating the use of the SMTP transport that we mentioned earlier would be http://schemas.xmlsoap.org/soap/SMTP.

In the WSDL document snippet below, we further qualify how a call to the `getQuote()` method gets translated into a valid SOAP document:

```
        <wsdl:operation name="getQuote">
          <wsdlsoap:operation soapAction=""/>
```

The `Java2WSDL` does not specify a `soapAction` attribute for the SOAP action header, but it is usually a good idea to do so since some servers use this header as a hint to allow or deny a request.

The next element is the `<input/>` element that defines the format of the SOAP body. The `<body/>` element is part of the SOAP extensions for WSDL that we mentioned earlier. It is used to describe the input and output messages of our SOAP calls. In the case of our example, the encoding style is standard SOAP:

```
      <wsdl:input>
        <wsdlsoap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://localhost/axis/services/StockQuote"
```

Reprinted for MC2WS/venkata.polineni@verizon.com, Verizon
Apress, Wrox Press (c) 2002, Copying Prohibited

```
            use="encoded"/>
        </wsdl:input>
```

Another possible value for the `use` attribute is literal, as in `use="literal"`. In this case, the argument is an XML document fragment. This type of encoding saves space because the XML special characters like '<' or '>' do not have to be escaped with '`&lt;`' or '`&gt;`'. However, this saving comes at the expense of coding – the client and server must manually serialize and de-serialize the data. For this reason, literal XML works best when exchanging data structures that are already represented as XML documents in the code.

As seen below, the description of the output packet is identical to the input, except for the `<wsdl:output/>` element:

```
        <wsdl:output>
          <wsdlsoap:body
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="http://localhost/axis/services/StockQuote"
            use="encoded"/>
          </wsdl:output>
        </wsdl:operation>
      </wsdl:binding>
```

Now, we have the answer to the *what* and the *how* of a web service. We still need to define the implementation of the web service. In Java terms, we have the Javadoc for our web service, but we still need a class file or a JAR file. In web service terms, we need to answer the 'where' question that we posed earlier, by giving a URL to the server hosting the service. This question is answered in the next section.

The service Element

The `<service/>` element specifies where to find the web service. In our example, the web service can be found on `localhost`:

```
      <wsdl:service name="StockQuoteService">
        <wsdl:port binding="intf:StockQuoteSoapBinding" name="StockQuote">
          <wsdlsoap:address
            location="http://localhost/axis/services/StockQuote"/>
        </wsdl:port>
      </wsdl:service>
```

As for the other WSDL elements that we have seen so far, the `name` attribute uniquely identifies the service in the WSDL document.

The `<service/>` element is a collection of `<port/>` elements. If we continue the analogy to the Java language, we could say that the port types are classes and the ports are objects. A `port` must refer to an existing binding (defined somewhere else in the document). In this example, it refers to the `StockQuoteSoapBinding`, which itself refers to the `StockQuote port` type.

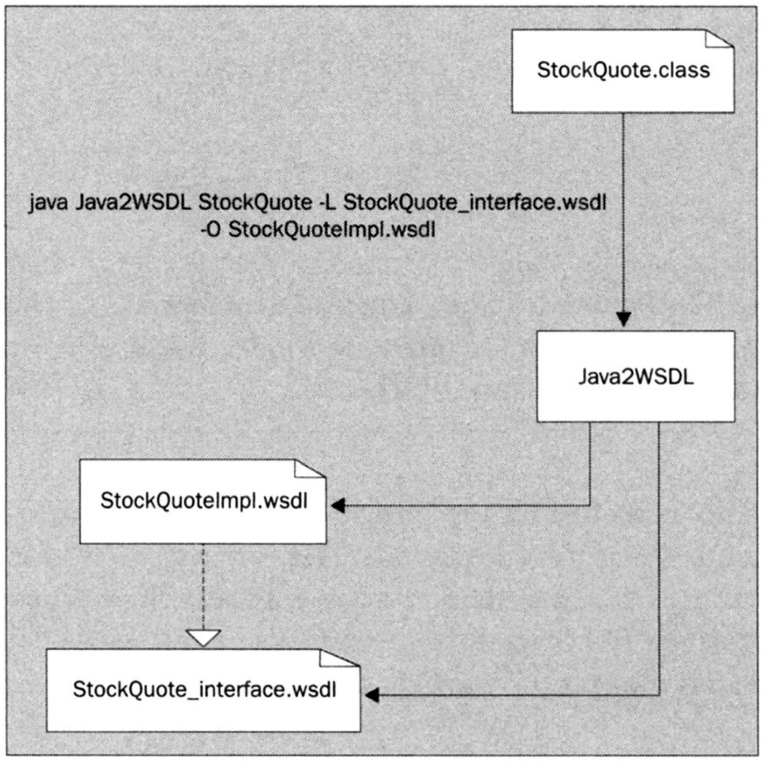
Once we have specified the binding for our service, we need to specify protocol-specific data for the actual location of the web service. In our case, we use the SOAP extensions for WSDL to specify the location of the service: `http://localhost/axis/services/StockQuote`.

We will see shortly how a WSDL document can be used to automatically generate client-side proxies to ease the burden of SOAP development. First, we will have a look at two more WSDL elements: `<import/>` and `<types/>`. Let's start with the `<import/>` element that allows us to organize our WSDL documents better.

The import Element

Very often, the `<service/>` element will be segregated into its own WSDL document for practical reasons. Among other things, it allows clients to bind to one well-defined interface and switch implementations at will. To satisfy that requirement, the `Java2WSDL` Axis tool that we used earlier can be invoked to create two files: one for the **interface definition** and one for the **interface implementation** as shown in the following diagram:

Apress, Wrox Press (c) 2002, Copying Prohibited


StockQuote.class
java Java2WSDL StockQuote -L StockQuote_interface.wsdl
-O StockQuoteImpl.wsdl
Java2WSDL
StockQuoteImpl.wsdl
StockQuote_interface.wsdl

Try It Out: Generate a WSDL Interface Definition Document

The following command creates two files, `StockQuote_interface.wsdl` and `StockQuoteImpl.wsdl` (make sure that the JAR files and `StockQuote.class` are in our classpath prior to running the command):

```
java org.apache.axis.wsdl.Java2WSDL com.wrox.jws.stockquote.StockQuote
-1 http://localhost:8080/axis/services/StockQuote -0 StockQuoteImpl.wsdl
-L StockQuote_interface.wsdl
```

How It Works

The `StockQuote_interface.wsdl` document is the `StockQuote.wsdl` document reviewed earlier, minus the `<service/>` element. The `StockQuoteImpl.wsdl` document (listed below) is made of the `<definitions/>` and `<service/>` elements that we are now familiar with, plus the `<import/>` element:

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions
  targetNamespace="http://stockquote.iws.wrox.com-impl"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:impl="http://stockquote.iws.wrox.com-impl"
  xmlns:intf="http://stockquote.iws.wrox.com"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <wsdl:import
    location="StockQuote_interface.wsdl"
    namespace="http://stockquote.iws.wrox.com"/>
  <wsdl:service name="StockQuoteService">
    <wsdl:port binding="intf:StockQuoteSoapBinding" name="StockQuote">
      <wsdlsoap:address
        location="http://localhost/axis/services/StockQuote"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Important Be sure to specify the '`-L StockQuote_interface.wsdl`' option. Otherwise `Java2WSDL` will simply generate an import statement without a location, which is going to be useless unless we hand-modify it.

The `<import/>` element allows us to include one WSDL document into another. In this particular case, we import

Reprinted for MC2WS/venkata.polineni@verizon.com, Verizon Apress, Wrox Press (c) 2002, Copying Prohibited

StockQuote_interface.wsdl. The <import/> element also specifies the namespace http://stockquote.iws.wrox.com, which is the namespace of our interface definition. Note that the interface file is useful on its own since it gives us a definition of the web service. This definition could be shared among developers that would each provide their own implementation of the web service.

We will now wrap up our review of WSDL with the <types/> elements that allows us to extend the data types provided by XML schemas. We will use the Market service that we developed back in Chapter 3 as an example of a web service that uses custom types.

The types Element

We can use Java2WSDL to generate the WSDL document for the Market example (make sure that the Axis JAR files we mentioned previously and Market.class are in the classpath prior to running that command):

```
java org.apache.axis.wsdl.Java2WSDL com.wrox.jws.stockquote.Market
-1 http://localhost/axis/services/Market
```

Please note that, we can also use the Axis servlet to generate the WSDL file, for this simply navigate to http://localhost/axis\services/Market?wsdl.

The generated WSDL document, Market.wsdl looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://stockquote.iws.wrox.com"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:impl="http://stockquote.iws.wrox.com-impl"
  xmlns:intf="http://stockquote.iws.wrox.com"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

The <definitions/> element is identical to the WSDL file we generated for StockQuote. However, the <types/> element that we see below is new. It contains the XML Schema for the MarketData class. Note that there is no schema for the Market class since we only use its methods and we never return or pass it as data. When required, Java2WSDL automatically adds the schema:

```
<types>
  <schema
    targetNamespace="http://stockquote.iws.wrox.com"
    xmlns="http://www.w3.org/2001/XMLSchema">
    <complexType name="MarketData">
      <sequence>
        <element name="DOW" nillable="true" type="xsd:string"/>
        <element name="NASDAQ" nillable="true" type="xsd:string"/>
        <element name="SP500" nillable="true" type="xsd:string"/>
```

The MarketData class starts with three strings. This might seem wasteful, but the nillable="true" attribute allows these strings to be omitted.

Keep in mind that most automated tools, including the bean serializer, which we discussed in Chapter 3, will include these values. When we generate a Java client based on the WSDL file, we will also see that these strings' values are fairly useless unless we manually modify the code and initialize the strings to meaningful values ("^DJI", "^IXIC"). Note that the static instance also suffers from the same ailment.

The rest of the sequence element contains the elements generated from the MarketData instance data, that is the ticker symbol, the double value, and the array of doubles for the market indices:

```
            <element name="ticker" nillable="true" type="xsd:string"/>
            <element name="doubleValue" type="xsd:double"/>
            <element name="indices" nillable="true"
              type="intf:ArrayOf_xsd_double"/>
          </sequence>
        </complexType>
```

The indices are defined as an array of doubles encoded according to the SOAP encoding rules:

```
    <complexType name="ArrayOf_xsd_double">
      <complexContent>
        <restriction base="SOAP-ENC:Array">
          <attribute ref="SOAP-ENC:arrayType"
            wsdl:arrayType="xsd:double[]"/>
        </restriction>
      </complexContent>
    </complexType>
    <element name="MarketData" nillable="true" type="intf:MarketData"/>
  </schema>
```

Reprinted for MC2WS/venkata.polineni@verizon.com, Verizon

Apress, Wrox Press (c) 2002, Copying Prohibited

```
    </types>
```

Apart from the addition of the types and minor changes like the one shown below for the return value of `getQuote()`, the remainder of `Market.wsdl` is identical to `StockQuote.wsdl`:

```
    ...
      <wsdl:message name="getQuoteResponse">
         <wsdl:part name="return" type="intf:MarketData"/>
      </wsdl:message>
    ...
```

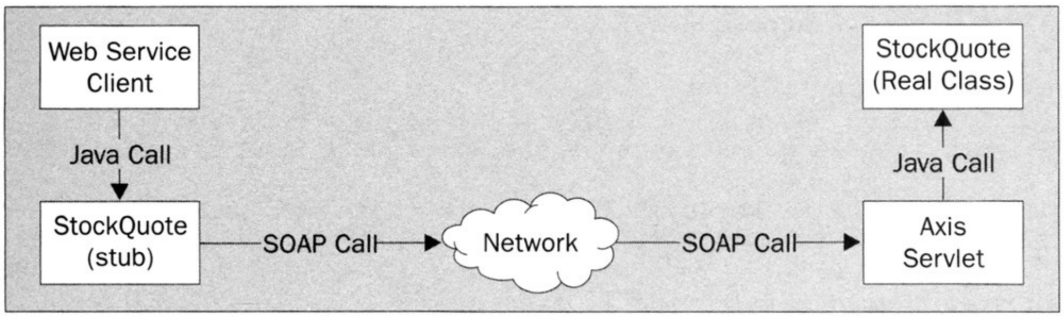
This presentation of the `<types/>` element concludes our overview of WSDL. In the next section we will use WSDL documents to develop web service clients.

WSDL-Based Clients

Using the WSDL documents we have generated, we will be able to automatically create client classes that provide an interface to the web service.

Java Clients

In this section, we will create a Java **stub class** and the Java classes necessary to support it. The following diagram shows the relationship between the client and the server when using a stub:



When calling the `getQuote()` method, our application calls the stub class, which takes care of **serializing** (from Java to XML) our inputs and sending them over the network via a SOAP call. When the call completes, the stub class also takes care of **deserializing** (from XML to Java) the return value. From the point of view of our application, everything looks as if we are making a local Java call.

Well, almost, because the stub will throw a `java.rmi.RemoteException` when something goes wrong. Keep in mind that we could be calling a service written in C++, C#, or Perl and running on an operating system that may be different from the one hosting our client application. This level of integration across platforms and programming languages is a major component in the success of web services.

It's apparent from this discussion that the WSDL file is the glue that binds web services and their clients together. Without the WSDL file, .NET and Java would not have a common language.

The StockQuoteClient

The `StockQuoteClient` is a WSDL-based client that uses the `StockQuote` example that we developed in Chapter 3.

Try It Out: StockQuoteClient

The first step is to generate the stub and its supporting classes. For this example, that step is optional as these classes are provided with the code.

Note If we want to regenerate the stub and its supporting class, we should make sure that `Exception.java` and all its references are deleted. This topic is discussed in the *How It Works* section below.

1. To generate the stub class that is used by `StockQuoteClient`, Axis provides the `WSDL2Java` class:

   ```
   java org.apache.axis.wsdl.WSDL2Java StockQuote.wsdl
   ```

   Note At the time of this writing, the version of `WSDL2Java` that comes with Axis (Beta 3) contains some minor problems. We will make special note of these problems as we go through the example. Hopefully, those caveats will have been addressed by the time you read these lines.

2. Build the stub and the client. The client code is contained in the `StockQuoteClient` class:

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

Reprinted for MC2WS/venkata.polineni@verizon.com, Verizon
Apress, Wrox Press (c) 2002, Copying Prohibited

```
    public class StockQuoteClient {
      /**
       * The main is used as a client of the StockQuote SOAP service.
       *
       * @param args The ticker symbols to get a quote for (e.g. sunw)
       */

      public static void main (String [] args) {
        final String methodName = "StockQuoteClient.main";

        try {
          if (args.length != 1) {
             System.err.println ("StockQuote Client");
             System.err.println (" Usage:
               java com.wrox.jws.stockquote.StockQuoteClient <ticker-symbol>");
             System.err.println (" Example:
               java com.wrox.jws.stockquote.StockQuoteClient sunw");
             System.exit (1);
          }

          // We create a service (StockQuoteService is the interface)
          StockQuoteService stockQuoteService = new StockQuoteServiceLocator();

          // We get a stub that implements the Service Description
          //Interface (SDI)
          StockQuote stockQuote = stockQuoteService.getStockQuote();

          // We call getQuote
          String quote = stockQuote.getQuote (args [0]);

          System.out.println ("The (delayed) value of " + args [0]
            + " is: " + quote);
        } catch (Exception exception) {
          System.err.println (methodName + ": " + exception.toString());
          exception.printStackTrace();
        }
      }
    }
```

3. Building the example is similar to what we did in Chapter 3 for `StockQuote` and `Market`. Keep in mind that we need to compile the client, the stub, and the supporting classes. Also, make sure that our classpath matches the earlier one (`src` is the current directory):

```
javac -d ..\classes StockQuote.java
javac -d ..\classes StockQuoteService.java
javac -d ..\classes StockQuoteServiceLocator.java
javac -d ..\classes StockQuoteSoapBindingStub.java
javac -d ..\classes StockQuoteClient.java
```

4. To run the `StockQuoteClient`, simply use the following command (make sure that our class path contains the files that we compiled in the previous step):

```
java com.wrox.jws.stockquote.StockQuoteClient IBM
```

5. The output is:

   The value of IBM is: 69.01

6. Unsurprisingly, the net result of the command is identical to the client that we developed in Chapter 3. The differences reside in the methodology, which involves the creation of a stub class as we describe in the next section.

How It Works

The first command that we executed in the previous section was `WSDL2Java`. The execution of that command creates the following files:

- `Exception.java`

  We have seen earlier, in the `StockQuote.wsdl` file, that the `Exception` class is thrown when a fault is received. Since the `WSDL2Java` class does not know (solely from the WSDL file) that the exception is really a `java.lang.Exception`, it generates a `com.wrox.jws.stockquote.Exception`. One way to avoid this caveat is to rely only on `java.rmi.RemoteException`, since it is thrown by every service method. In other words, we can simply ignore the generated `Exception` class.

- `StockQuote.java`

Reprinted for MC2WS/venkata.polineni@verizon.com, Verizon

Apress, Wrox Press (c) 2002, Copying Prohibited

The `StockQuote` interface corresponds to the `StockQuote portType` in the WSDL file. It extends `java.rmi.Remote`. This interface is called the **Service Definition Interface** (SDI).

- `StockQuoteSoapBindingStub.java`

  This class is the stub class that we mentioned earlier. It implements the SDI (`StockQuote` in our case).

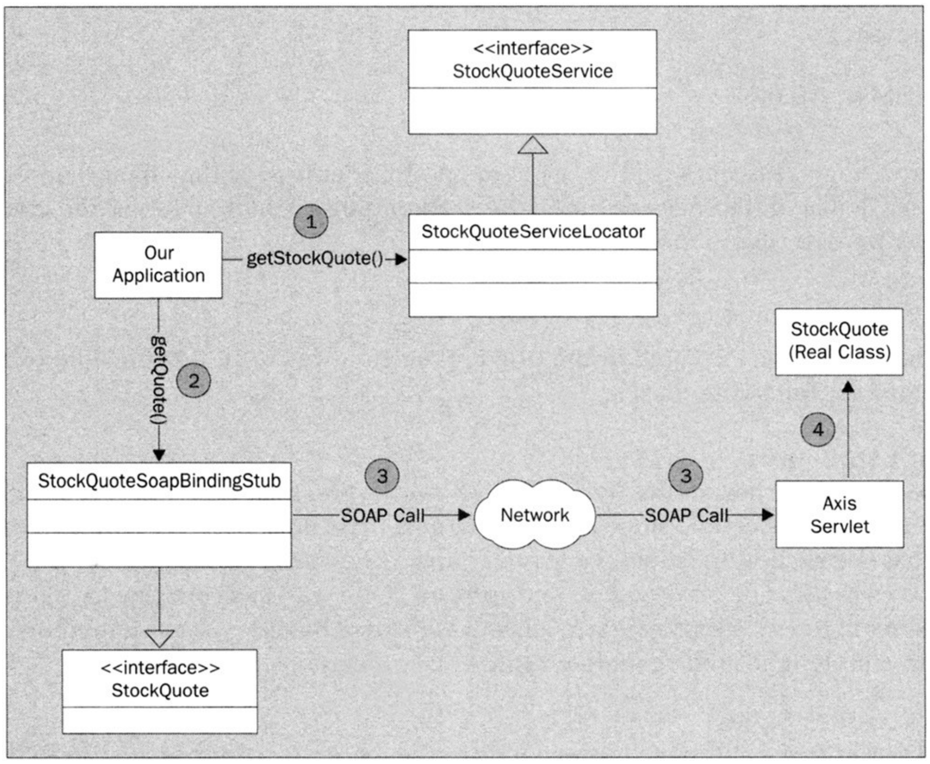
- `StockQuoteService.java`

  The `StockQuoteService` interface is derived from the `<service/>` element in WSDL. It defines three methods – one method to get the URL of the web service, one method per port to get a stub, and one to get a stub class for an arbitrary URL. This interface is implemented by the **service locator.**

- `StockQuoteServiceLocator.java`

  This class implements the `StockQuoteService` interface. It is a class factory for the stub (see below for details).

Thus, we can refine the previous diagram to include the supporting classes of the stub (numbers in circles indicate the order of execution):



The following table summarizes the rules of class/interface creation when using `WSDL2Java`:

WSDL Element	Java Class or Interface
For each element in `<types/>`	A Java class
`<portType/>`	A Java interface
`<binding/>`	A stub class
`<service/>`	A service interface
	A service implementation (the locator)
`<fault/>`	A Java class

Note that `WSDL2Java` follows the rules of JAX-RPC that we mentioned in Chapter 3.

Now we have a basic understanding of the `WSDL2Java` tool. Let's see how we can use the classes generated from WSDL when developing `StockQuoteClient`.

If we look at the listing above, we will notice that since all generated classes are in the `com.wrox.jws.stockquote` package along with this client, we don't need to import anything.

Reprinted for MC2WS/venkata.polineni@verizon.com, Verizon Apress, Wrox Press (c) 2002, Copying Prohibited

After dealing with the only possible argument, a stock ticker, we reach the essence of the client code. First, we get a reference to the `StockQuoteService` interface by creating an instance of the `StockQuoteServiceLocator` class:

```
// We create a service (StockQuoteService is the interface)
StockQuoteService stockQuoteService = new StockQuoteServiceLocator();
```

Next, we get a reference to the `StockQuote` interface (the one that implements `getQuote()`) through the `getStockQuote()` method:

```
// Get a stub that implements the Service Description Interface (SDI)
StockQuote stockQuote = stockQuoteService.getStockQuote();
```

Third, and last, we call the `getQuote` method as we would call any other local interface:

```
// We call getQuote
String quote = stockQuote.getQuote(args[0]);

System.out.println("The (delayed) value of " + args[0]
  + " is: " + quote);
```

To compare the simplicity of this approach to the relative complexity of an Axis client, we can go back to the `main()` method of `StockQuote` or `Market` that we developed in Chapter 3.

The MarketClient

The `StockQuoteClient` class was a fairly trivial example since it only used strings as input and output. To see for ourselves what happens when we use complex data types, we will use `WSDL2Java` to generate a stub for the `Market` web service.

Try It Out: MarketClient

1. If we want to generate the stub and its supporting classes (they are included with the code of this chapter), the following command will do the trick (use the same classpath as before):

   ```
   java org.apache.axis.wsdl.WSDL2Java Market.wsdl
   ```

2. Now we build the stub and client. The `MarketClient` class is listed below. As you can see, it is very similar to the `StockQuoteClient`:

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

   public class MarketClient {
     /**
      * The main is used as a client of the Market SOAP service.
      *
      * @param args The ticker symbols to get a quote for (e.g. sunw)
      */
     public static void main(String [] args) {
       final String methodName = "MarketClient.main";

       try {
         if(args.length != 1) {
           System.err.println("Market Client");
           System.err.println(" Usage:
             java com.wrox.jws.stockquote.MarketClient <ticker-symbol>");
           System.err.println(" Example:
             java com.wrox.jws.stockquote.MarketClient sunw");
           System.exit(1);
         }

         // We create a service (StockQuoteService is the interface)
         MarketService marketService = new MarketServiceLocator();

         // We get a stub that implements the Service Description
         // Interface (SDI)
         Market market = marketService.getMarket();

         // We call getQuote
         MarketData marketData = market.getQuote(args[0]);

         System.out.println("The (delayed) value of " + args[0]
           + " is: " + marketData.getDoubleValue() + ", and the NASDAQ is: "
           + marketData.getIndices() [1]);
       } catch (Exception exception) {
         System.err.println(methodName + ": " + exception.toString());
   ```

Reprinted for MC2WS/venkata.polineni@verizon.com, Verizon Apress, Wrox Press (c) 2002, Copying Prohibited

```
        }
      }
    }
```

3. Building the code is analogous to what we did for `StockQuoteClient` (use the same class path and the current directory is `src`):

```
    javac -d ..\classes Market.java
    javac -d ..\classes MarketService.java
    javac -d ..\classes MarketServiceLocator.java
    javac -d ..\classes MarketSoapBindingStub.java
    javac -d ..\classes MarketData.java
    javac -d ..\classes MarketClient.java
```

4. Now, to test the stub and the client, we run the command:

```
    java com.wrox.jws.stockquote.MarketClient SUNW
```

5. Here is the output of an execution of `MarketClient` for `SUNW`:

   The value of SUNW is: 5.93, and the NASDAQ is: 1360.62

   Here also, we have the same output as what we saw in Chapter 3.

How It Works

The notable difference between the `StockQuoteClient` and the `MarketClient` is the `MarketData` class. The appearance of the `MarketData` class conforms to the rules that we mentioned earlier – one Java class is generated per entry in the `<types/>` element.

Functionally, the client-side `MarketData` class is equivalent to the server-side version. The main differences are the initialization of final values, the removal of a constructor, and the addition of the `equal()` method. However, there are a few caveats that are worth pointing out, so let's quickly review the code for the `MarketData` class generated by `WSDL2Java`:

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

public class MarketData implements java.io.Serializable {
  private java.lang.String DOW = "^DJI";
  private java.lang.String NASDAQ = "^IXIC";
  private java.lang.String SP500 = "^GSPC";
  private java.lang.String ticker; // attribute
  private double doubleValue; // attribute
  privat double[] indices; // attribute

public MarketData() {
```

We modified the DOW, NASDAQ, and SP500 variables with the initializations shown above. By default, these three strings are initialized to `null`. As we can see below, without these changes the `getDow()` method would be useless:

```
    public java.lang.String getDow() {
      return DOW;
    }
```

The `getNASDAQ()` and `getSP500()` methods follow the same pattern.

The next method is `getIndices()`, which returns the values of the three major US stock indexes:

```
    public double[] getIndices() {
      return indices;
    }
```

To be able to use the `getIndices()`, one needs to know that `index=0` corresponds to DOW, `index=1` to NASDAQ, and `index=2` to SP500. This could be indicated in the documentation of the web service. The remainder of this listing contains no surprises.

The other classes generated from the `Market.wsdl` file are analogous to the ones generated for `StockQuote`.

Using the stub for `Market` contains no new ideas as one can see in the code below. The `MarketClient.java` file, which comes with the code for this chapter, contains the entire code:

```
        // We create a service (StockQuoteService is the interface)
        MarketService marketService = new MarketServiceLocator();

        // Get a stub that implements the Service Description Interface (SDI)
        Market market = marketService.getMarket();
        // We call getQuote
        MarketData marketData = market.getQuote(args[0]);
```

Reprinted for MC2WS/venkata.polineni@verizon.com, Verizon

Apress, Wrox Press (c) 2002, Copying Prohibited

```
            System.out.println("The (delayed) value of " + args[0]
              + " is: " + marketData.getDoubleValue() + ", and the NASDAQ is: "
              + marketData.getIndices()[1]);
```

During testing, if we happen to get an error like ‘could not find deserializer for type http://localhost/axis/services/StockQuote:MarketData’, be sure to double-check our deployment descriptor for the Market service for the QName and its name space. The client and server must use the same names (QName and namespace) for the deserializer to be registered properly.

Summary

In this chapter, we have covered the important topic of web service description. It is an important topic for two reasons. First, we want people to be able to read descriptions of the services that we provide. Without that knowledge, nobody would be able to use the software that we develop. Second, we want tools like `WSDL2Java` to automatically read a description of our web services and provides, their users with a quick interface to our web services.

A technology that has the potential of transforming that dream into reality is the Web Service Description Language (WSDL), and XML dialect. We reviewed the elements that make up a WSDL document and saw that WSDL answers three fundamental questions about web service by describing them in terms of endpoints and the messages that transit between these endpoints.

The four questions about web services that are addressed by WSDL are:

- **What?**

  The machine-readable answer to that question is in the `<message/>` and `<portType/>` elements. The human-readable answer is in the `<name/>` and `<document/>` elements.

- **Which?**

  This tells us about the language used and is found in the `<types/>` element.

- **How?**

  The answer can be found in the `<binding/>` elements.

- **Where?**

  The answer is specified in the `<service/>` elements.

We concluded the chapter by generating stub classes for `StockQuote` and `Market` using their respective WSDL documents.

Reprinted for MC2WS/venkata.polineni@verizon.com, Verizon Apress, Wrox Press (c) 2002, Copying Prohibited