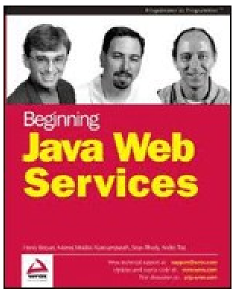


# Beginning Java Web Services

by Henry Bequet

# Chapter 6: Invoking Web Services

## Overview

By now it is apparent that web services are all about connecting applications over a network, allowing them to communicate. These applications do not necessarily know about each other in any depth; they don't have to be written in the same programming language, or even run on the same platform or operating system. The standards and specifications that we have read about in the previous chapters, and will read about in the following chapters, are all based on the fact that they should allow application-to-application communication regardless of platform, programming language, or even the network protocol that is used.

This implies that when talking about web services, we have to distinguish between the **requester,** or client, and the **provider,** or server, of a web service. The only contract between the two is the description of the web service, namely the WSDL document that contains the abstract definition of the offered interface and the protocol information that a client needs to bind to that service.

So far we have looked at some of the basic principles of web services, namely how XML makes it possible to exchange data in a programming language-neutral way, and how to describe functions in an abstract fashion using WSDL.

In this chapter, we shall see:

- Web services invocation models – static and dynamic
- Java API for XML-based RPC, or JAX-RPC
- Non-SOAP web services

An example of the last point above is the **Web Services Invocation Framework (WSIF),** which was initially developed by IBM and has been being donated to Apache as an open source project. We will take a look at some examples that show how to develop client code that takes a WSDL definition at run time and creates the appropriate invocation constructs based on that definition.

## Web Services Invocation Models

WSDL not only delivers information about the functional interface that is offered by a web service, it also describes details of the protocol that can be used to access that service. Most web services that are built today use the SOAP protocol as their communication protocol. A client that invokes a service will build an XML message, which has the `<Envelope>` element as its root element and carries a message that complies with the SOAP specification. For example, the Apache SOAP implementation used the namespace of the envelope's `<Body>` element as an identifier for the service that is invoked. This kind of information is needed by a client that wants to access the service, but is not really part of the service interface – it is protocol information, or, according to the WSDL specification, binding information.

We spend a great deal of time in this book discussing SOAP and its various implementations, most notably the Apache Axis package. However, if you think that web services can only be implemented and consumed using SOAP, you are mistaken. While SOAP is presently the dominant protocol used in the marketplace for web services, we expect that this will change over time to include many other protocols as well.

Web services may not exclusively be built to support SOAP as the communication protocol, which means that we want to build client code that is based on the service's interface definition, not on its run time environment. In other words, if a service happens to exist in the same address space that the client is running in, there is no reason to marshal and unmarshal all the data and send it out over an HTTP connection, if a local Java or RMI/IIOP call would do the job just fine.

In other cases, however, we may know that SOAP is the only way we will ever use to access a service. For example, the .NET platform currently only supports SOAP over HTTP as its communication protocol (unless you happen to be in a pure .NET environment; but we as Java programmers are not, are we?). Thus, if we want to take advantage of a service that has been implemented on either the Internet or on an intranet using .NET, SOAP may be the only choice. In these cases, a static client invocation model may be the right choice. Let's now look at the two invocation models in detail.

### Static Invocation Model

This model is the more common one. We have to learn about an existing web service, probably by accessing its WSDL description, potentially by finding it in a UDDI registry and generating code that somehow wraps the invocation of the service in a Java class. This is because, after all, we don't want everyone who uses this new service to write tons of code creating XML artifacts that encode the invocation of that service. This would mean converting all Java objects into XML structures of some sort.

Moreover, in the common case that this service is available via SOAP over HTTP, we would have to write code that generates the right type of HTTP request. We can make our lives easier by generating Java code, which converts a web service definition into a Java interface and a client side implementation of this interface that clients can use as if the service was available as a local Java class. To make this clear, let us reuse the `StockQuote` example from Chapter 3.

Here is the WSDL document again that describes the service:

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

```
    targetNamespace="http://localhost:8080/axis/services/StockQuote"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:impl="http://localhost:8080/axis/services/StockQuote-impl"
    xmlns:intf="http://localhost:8080/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:definitions>
```

This WSDL definition does not contain any information about the actual protocol that could be used to invoke this service. However, it describes the interface of this service in an abstract, or, a programming language-neutral way. We'll get to the protocol part in a little while.

To build a Java client with the static invocation model, which uses this web service, we have to create a representation of the service that is Java-based, and not XML-based. In other words, we have to generate a Java interface that matches the operations and messages defined in the WSDL document. Until recently, every provider of Java-based web services solutions chose their own way of doing this, providing proprietary tools within their environment. Luckily, this has now changed with the creation of a standard Java API that describes this, namely the **Java API for XML-based RPC,** or **JAX-RPC.**

For example, the `StockQuote` service that we showed above can be represented in Java by the following interface:

```
package com.wrox.jws.StockQuote;

public interface StockQuote extends java.rmi.Remote {

  public java.lang.String getQuote(java.lang.String in0) throws
    java.rmi.RemoteException, com.wrox.jws.StockQuote.Exception;
}
```

Looking at this example, we can easily see that the `<portType>` defined in the WSDL extract above, named `StockQuote`, was turned into a Java interface called `StockQuote`. Similarly, the operation in that port type, `<operation name="getQuote">`, was turned into a method called `getQuote()` on the Java interface.

So, the first step in the process is to generate a Java interface that represents the port type in the WSDL document.

We learned in the previous chapter the WSDL not only contains the definition of the offered web service interface, it also contains information about how to access that service. This information is called the protocol binding, and is represented by the `<binding>` element in the WSDL document. Most commonly, this protocol binding defines the additional information needed to access a web service via SOAP over HTTP.

For example, we could enhance the WSDL document for our `StockQuote` service with the following protocol binding:

```
<definitions ...>
  ...
  <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:8080/axis/services/StockQuote"
          use="encoded"/>
      </wsdl:input>
```

```
          <wsdl:output>
            <wsdlsoap:body
              encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              namespace="http://localhost:8080/axis/services/StockQuote"
              use="encoded"/>
            </wsdl:output>
          </wsdl:operation>
        </wsdl:binding>

      <wsdl:service name="StockQuoteService">
        <wsdl:port binding="intf:StockQuoteSoapBinding" name="StockQuote">
          <wsdlsoap:address
            location="http://localhost:8080/axis/services/StockQuote"/>
        </wsdl:port>
      </wsdl:service>

    </definitions>
```

The `<binding>` element describes everything we need to know to build a SOAP request message that the web service can understand, and it also defines what kind of return data we can expect from that service. We can generate code that knows how to build such a request and map it to the Java interface that we have generated in the first step. This allows client application developers to use this code to interact with the web service.

What exactly this code looks like depends on the SOAP package that we use. As mentioned above, the JAX-RPC standard only defines an API, namely how to map WSDL port types into a Java interface; it does not define how to implement the invocation of the actual request.

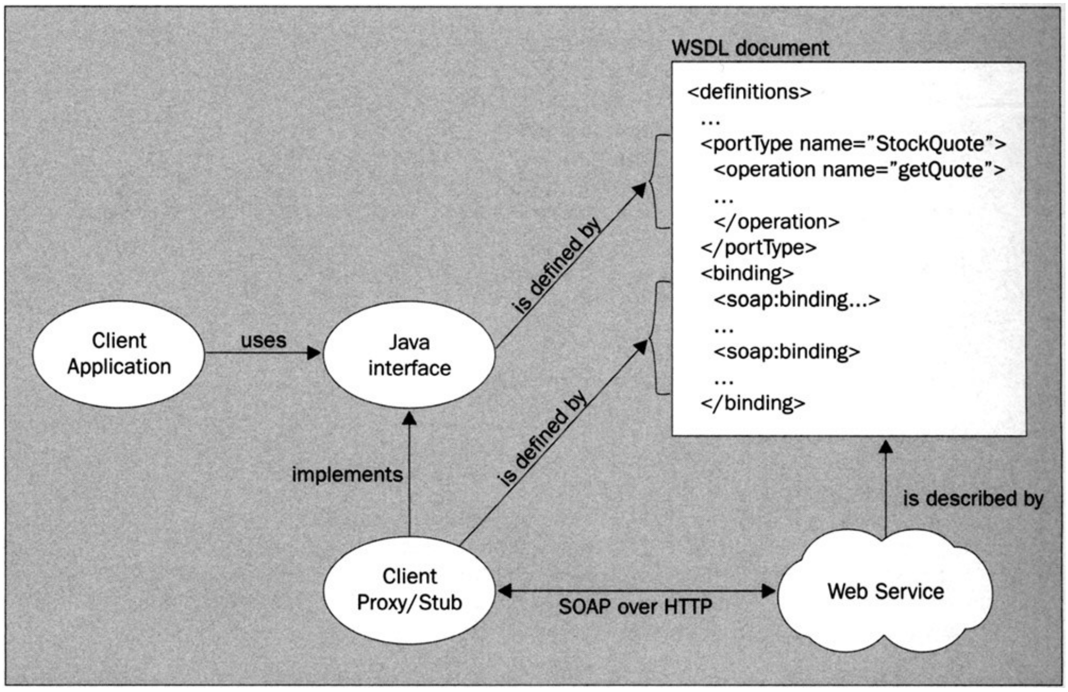
If we use the Apache Axis package, however, the `getQuote` operation from our WSDL is turned into the following implementation:

```
    public class StockQuoteSoapBindingStub extends org.apache.axis.client.Stub
      implements com.wrox.jws.stockquote.StockQuote {
    ...
      public java.lang.String getQuote(java.lang.String in0)
          throws java.rmi.RemoteException, com.wrox.jws.stockquote.Exception
      {
        if(super.cachedEndpoint == null) {
          throw new org.apache.axis.NoEndPointException();
        }
        org.apache.axis.client.Call call = createCall();
        javax.xml.rpc.namespace.QName p0QName =
          new javax.xml.rpc.namespace.QName("", "in0");
        call.addParameter(p0QName,
          new javax.xml.rpc.namespace.QName("http://www.w3.org/2001/XMLSchema",
            "string"),
          java.lang.String.class,
          javax.xml.rpc.ParameterMode.IN);

        call.setReturnType(
          new javax.xml.rpc.namespace.QName("http://www.w3.org/2001/XMLSchema",
            "string"));
        call.setUseSOAPAction(true);
        call.setSOAPActionURI("");
        call.setOperationStyle("rpc");

        call.setOperationName(new javax.xml.rpc.namespace.QName(
          "http://localhost:8080/axis/services/StockQuote", "getQuote"));

        Object resp = call.invoke(new Object[] {in0});

        if(resp instanceof java.rmi.RemoteException) {
          throw(java.rmi.RemoteException)resp;
        } else {
          try {
            return(java.lang.String) resp;
          } catch(java.lang.Exception e) {
          return(java.lang.String) org.apache.axis.utils.JavaUtils.
            convert(resp, java.lang.String.class);
          }
        }
      }
    }
```

Given this rather lengthy chunk of code, it is pretty obvious that we wouldn't want to write it ourselves (and it is only an extract of a class that is in reality even longer). Luckily, there are tools available as part of the Apache Axis package that generate this code for us. However, this snippet shows that the generated code implements the Java interface we saw earlier, representing the abstract interface of the service. This allows clients to use the `SOAPBindingStub` class as a local proxy, hiding the SOAP interaction from the client developer and offering a standard Java interface.

The second step in the static invocation model is the generation of a stub that implements the interface generated in first step, and contains the implementation of the protocol binding defined in the WSDL document.

The following diagram shows the structure of a static web services client:



This diagram shows that the WSDL document represents the contract between the web service and its client. The client need not know how the web service was implemented, which programming language was used, and on which platform it exists. Different parts of the WSDL document are used to create a Java interface and a Java class (which implements that interface) that hide the details of the service from the client. The model is called the static model because code that represents the service is generated at development time.

If the WSDL document changes, the code must be regenerated and the client code must be recompiled to take those changes into account. For example, if a new operation is offered in a newer version of the service, a new Java interface and stub must be generated, so that the client can take advantage of it. In this case, however, existing operations can still be used as before.

There is no automatic way to find out if a WSDL document has changed, or, in other words, whether a compiled Java stub still matches the interface described in the document. If a service provider updates the service, the client may or may not still work, depending on the changes. If an incompatible change was made, the client will receive a runtime error.

In general, it is fair to say that versioning of web services is currently not supported by any standard. Web services request messages do not carry a version number or anything like that. This means that any change to a web service will typically lead to disruption of the users of that service, so care should be taken to avoid this situation whenever possible.

This is not the case in the dynamic invocation model, which we will look at next.

## Dynamic Invocation Model

One of the promises of web services technology in general is to increase the level of automation in application-to-application communication. This means that applications can establish a communication path without knowing any details about each other (beyond the WSDL document describing their web services interfaces), or that changes to an existing system, or service, can be utilized without requiring manual changes to the user of that system.

Universal Discovery, Description, and Integration (UDDI) registries, as we shall see in Chapter 7, contain information about businesses and the services they offer, many of which may be web services represented by various WSDL documents. It should be possible for applications to use these registries to dynamically discover and use services with little, if any, manual intervention. Given this as a goal, we need a way for a client to use a service without the generation of code for interfaces and stub classes as described by the static invocation model.

Clients must be able to make decisions about which service to invoke and over which protocol, based on the definitions in the WSDL document for that service. If this document changes, the client must be able to adapt to that change without having to re-implement the client code.

In this case, the overall programming model that we use for web services remains the same as before: the service is represented by its WSDL interface definition and its protocol bindings. We use Java code to wrap the invocation of the service on the client, so that the client developer need not know or care about how to build requests for different protocols. This means that the client developer only needs to care about the functional interface of a service, and not about how it is invoked.

**Protocol Independence**

We have mentioned several times now that the description of a web service in a WSDL document is a key piece of information for developing clients, which consume that service. A WSDL document contains two layers of information, and we can handle both of them statically or dynamically. These layers are:

- The interface definition (represented by the `<portType>` element)
- The protocol bindings (represented by the `<bindings>` and `<service>` elements)

We now want to develop Java code that uses this service in a dynamic way, by using the WSDL definitions at runtime to create proper requests – in a format that the server can understand, as outlined in the WSDL definition.

Let us assume that a web service is available over a certain protocol, for example, SOAP over HTTP. In that case, the `<port>` element, which is contained in the `<service>` element, carries information about where the service is located, so that a client can send a request to that service. Our client should not have any hard-coded dependency on the target URL, but rather it should read this address from the WSDL document at run time. Then, if the address ever changes, perhaps because the service is moved to a new server, the client code does not have to change.

Another example of information that can change and should therefore not be hardwired into the client code is information about how exactly to build the request and response messages. For example, the service provider might choose to change a service to use the 'literal XML' encoding scheme rather than the default SOAP encoding scheme (discussed later in this chapter). Again, in the dynamic programming model, the client code can adjust to those changes because the encoding scheme is interpreted at run time and the right encoding is applied when building a request and receiving the response.

Also, a WSDL document could contain information allowing us to invoke a service over several protocols at the same time. Most web services today utilize SOAP over HTTP. In the future, however, the invocation of a web service could be done over RMI/IIOP (which is the protocol used for Enterprise JavaBeans invocations), or over a messaging layer using the Java Messaging Service (JMS), just to name two examples.

This is useful if we want to build a system that uses a common programming model for its components – namely accessing those components via WSDL-defined interfaces, regardless of whether they are local or remote.

**Interface Independence**

The interface description in WSDL, more specifically the `<portType>` element, contains information about the structure of the messages that a web service sends and receives. In other words, it defines the input parameters of the service as well as the return type. To achieve interface independence in the client code, this code must interpret this information at run time to build the right request and properly interpret the returned response.

For example there could be a situation where an existing web service may change its interface over time. A newer version of a web service could require additional parameters to be passed along, and we want to develop client code that can automatically adjust to those changes and pass along those additional parameters to the service when required. Again, these kinds of scenarios will be less common and will only occur in highly dynamic environments. Most usages of the dynamic invocation model will be based on protocol independence and not on itnerface invariance.

Now that we have described the two invocation programming models, we can go a level deeper and look at a standard Java API that supports both models, namely JAX-RPC.

## JAX-RPC

If we were to describe the main purpose of JAX-RPC in one sentence, we could say that this API defines the rules for turning WSDL port-type information into Java and vice versa. These rules apply on both the client and the server side. Consider the case where we have existing Java code that we want to offer to clients over a web services layer. One of the things we have to do is describe the interface of our Java code in WSDL, and JAX-RPC tells us how to do that. Similarly, if we have the WSDL description of a given web service and want to invoke that web service in our client Java application, we can use an implementation of the JAX-RPC API to turn this WSDL description into a Java interface.

Since this chapter is all about how to consume web services, the second scenario is what we will focus on here. However, many of the things we will discuss apply to both the client and the server.

As mentioned, JAX-RPC supports both the invocation models described earlier. We will look at the classes that are used for both of them in detail. For the code examples, we will use the Apache Axis beta 3 package, which implements the JAX-RPC interface. You can find detailed information about how to obtain Axis and how to install it in Chapter 3.

### Java to WSDL Mapping

As we mentioned above, JAX-RPC defines how to map Java code into WSDL definitions. The specification can be found at http://java.sun.com/xml/downloads/jaxrpc.html#jaxrpcspec. The rules for this are pretty straightforward, so let us simply go through each of them here:

- **Basic Java types are mapped to basic XML Schema types.**

  This rule means that if we are dealing with the common data types like `String`, `Integer`, `Boolean`, and so on, we can map them directly to data types that are defined for XML Schemas.

- **Primitive types use Holder classes.**

  A primitive type is something that Java defines that does not inherit from `java.lang.Object`, such as `int` or `short`. For these to be handled properly, a so-called `Holder` class is used that carries the value of the primitive type within a real object.

- **JavaBean classes are mapped to an XML Schema structure.**

  If a class contains a value object, in other words, is an object of a class that holds data and is serializable, it gets converted into an XML Schema describing the structure of the data. For example, a Java class that looks like this:

  ```
  public class Person {
    public String firstName;
    public String lastName;
    public int age;
  }
  ```

  maps to this XML Schema definition:

  ```
  <complexType name="person">
    <all>
      <element name="firstName" nillable="true" type="string">
      <element name="lastName" nillable="true" type="String">
      <element name="age" nillable="true" type="int">
    </all>
  </complexType>
  ```

- **Java artifacts are mapped to the appropriate WSDL artifacts.**

  This is described in the following table:

  | Java | WSDL |
  |---|---|
  | Package | WSDL document |
  | Interface | `portType` |
  | Method | `operation` |
  | Exception | `fault` |

- **The Java interface must extend `java.rmi.Remote` and each method must throw a `java.rmi.RemoteException`.**

  This is in line with how remote invocations over RMI are done in Java. Each method that can be called over RMI (for example, methods exposed by an Enterprise JavaBean) must throw `java.rmi.RemoteException`. A web service is typically invoked over a network, hence the rule about adding the exception.

There are more rules and more details to this, but we have covered the most important parts. Normally, we would use a tool to generate the appropriate WSDL file for our Java code, and this tool will have all of the details implemented.

## WSDL to Java Mapping

The previous section described the mapping of existing Java code into WSDL. Here we are looking at the opposite scenario where WSDL information is turned into Java code. This set of rules is used on both the server and the client side. On the server, it defines what the concrete implementation of the described service in Java will look like. This can be used to create skeletons in Java for a given, possibly standardized, WSDL document.

In this chapter the client-side case is more relevant for us. An existing web service is represented by its WSDL document. This can now be used to generate clients, which follow the static programming model. It is also used by dynamic clients, which build the appropriate request and response structures at run time.

Let us look at the rules that are defined for this mapping:

- **Basic XML Schema types are mapped to basic Java types.**

  This is basically the same rule as for the opposite case described above.

- **XML structures and complex types are mapped to a JavaBean.**

  If the WSDL document uses message parts that are complex types, or XML structures, then these are mapped to plain JavaBeans that contain the elements of the structure as their attributes. An example for this is the simple `Person` class we listed above. Just in this case, the process is done in the opposite direction, starting with the WSDL document and creating Java code from that.

- **Enumerations are turned into public static final attributes.**

  Defining the attributes is the most common way of simulating enumerations in Java. For example, assume the following enumeration in XML (the Java package names for these classes are generated from the namespace of the WSDL document, but how exactly the mapping is done depends on the implementation):

```
    <simpleType name="MonthType">
      <restriction base="xsd:string">
        <enumeration value="January"/>
        <enumeration value="February"/>
        ...
      </restriction>
    </simpleType>
```

This maps to the following Java class:

```
public class MonthType implements java.io.Serializable {

  private java.lang.String_value_;
  private static java.util.HashMap _table_ = new java.util.HashMap();

  // Constructor
  protected MonthType(java.lang.String value) {
    _value_ = value;
    _table_.put(_value_, this);
  };

  public static final java.lang.String _January = "January";
  public static final java.lang.String _February = "February";
  public static final java.lang.String _March = "March";
  ...
  public static final MonthType January = new MonthType(_January);
  public static final MonthType February = new MonthType(_February);
  public static final MonthType March = new MonthType(_March);

  ...
  public java.lang.String getValue() { ...}

  public static MonthType fromValue(java.lang.String value)
    throws java.lang.IllegalStateException { ... }
  public static MonthType fromString(String value)
    throws java.lang.IllegalStateException { ... }
  public boolean equals(Object obj) { ... }
  public int hashCode() { ... }
  public String toString() { ... }
}
```

- **WSDL artifacts are mapped to the appropriate Java artifacts.**

  The same mappings between Java artifacts and WSDL artifacts apply as in the opposite case described previously. In other words, a document maps to a package, a port type maps to a class and each operation maps to a method in that class.

## Service Mapping

The mapping of the `<service>` element in WSDL is not quite as straightforward as those for the other elements that we have looked at so far, so we will take a little more time to have a look at it. As we shall see in Chapter 7, the `<service>` element defines where a web service can be found through its contained `<port>` elements.

JAX-RPC defines an interface called `javax.xml.rpc.Service`. A concrete class implementing the interface must exist at run time. The `Service` interface contains methods that a client can use to invoke the actual web service.

There are two different styles that clients can use in order to invoke a web service using the `Service` interface. One is the use of a proxy object, which is returned by one of the `getPort()` methods of the `Service` interface. This proxy object exposes the methods of the web service locally, by turning the WSDL port type into Java.

Previously we had generated an interface called `StockQuote` from the WSDL document. This interface had one method on it, namely `getQuote()`. Using the `getPort()` method returns a run-time object that implements the `StockQuote` interface and will route all invocations of the `getQuote()` method to the actual web service. The proxy is created at run time and no code is generated.

The other choice we have is the use of a `javax.xml.rpc.Call` object. A `Call` object represents one invocation of a web service. It allows us to set parameters and other invocation variables, and then execute the request. We will go through both invocation styles in more detail below.

First, let's have a look at the methods that are defined in the `Service` class. Here is the entire interface. You will find this interface in every JAX-RPC-compliant implementation:

```
package javax.xml.rpc;

import javax.xml.rpc.encoding.TypeMappingRegistry;
import javax.xml.rpc.handler.HandlerRegistry;
import javax.xml.rpc.namespace.QName;

public interface Service {

  public java.rmi.Remote getPort(QName portName, Class proxyInterface)
    throws ServiceException;

  public java.rmi.Remote getPort(Class serviceDefInterface)
    throws ServiceException;

  public Call createCall(QName portName)
    throws ServiceException;

  public Call createCall(QName portName, String operationName)
    throws ServiceException;

  public Call createCall(QName portName, QName operationName)
    throws ServiceException;
  public Call createCall() throws ServiceException;

  public Call[] getCalls() throws ServiceException;

  public HandlerRegistry getHandlerRegistry();

  public java.net.URL getWSDLDocumentLocation();

  public QName getServiceName();

  public java.util.Iterator getPorts();

  public TypeMappingRegistry getTypeMappingRegistry();
}
```

**The getPort() Method**

We will just pick the most important methods here and look at them in detail, so let's get started with the `getPort()` method:

```
public java.rmi.Remote getPort(QName portName, Class proxyInterface)
    throws ServiceException;
```

A port is the actual location of a web service. How we describe this location depends on the protocol that is used to access the service. For example, if the service is accessible via SOAP over HTTP, then the location is a URL, to which you can send the request message.

Each `<service>` element can have multiple `<port>` elements in it, and each one can be retrieved from the `Service` class using the `getPort()` method. We specify the port we are looking for by passing its `QName`. This is the fully qualified name of the `<port>` element in the WSDL document. We will have a look at this in more detail before explaining the `getPort()` method.

Here is the complete WSDL document again that we use for this example. This doucment will serve as the basis for most examples that will follow below. We can find the namespace of this WSDL document in the `<definitions>` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  targetNamespace="http://localhost:8080/axis/services/StockQuote"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:impl="http://localhost:8080/axis/service/StockQuote-impl"
  xmlns:intf="http://localhost:8080/axis/service/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:8080/axis/services/StockQuote"
             use="encoded"/>
          </wsdl:input>
          <wsdl:output>
            <wsdlsoap:body
             encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
             namespace="http://localhost:8080/axis/services/StockQuote"
             use="encoded"/>
          </wsdl:output>
        </wsdl:operation>
      </wsdl:binding>

      <wsdl:service name="StockQuoteService">
```

We will take the `<port>` element here as our example to illustrate the concept of fully qualified names in XML. The `<port>` element is defined like this:

```
        <wsdl:port binding="intf:StockQuoteSoapBinding" name="StockQuote">
          <wsdlsoap:address
           location="http://localhost:8080/axis/services/StockQuote"/>
        </wsdl:port>
      </wsdl:service>
    </wsdl:definitions>
```

So, the local name `StockQuote` and the namespace `http://localhost:8080/axis/services/Stockquote` can be used to build a `QName` instance like this:

```
    QName portQN = new QName("http://localhost:8080/axis/services/StockQuote",
      "StockQuote");
```

Many of the APIs in JAX-RPC require fully qualified names for elements, so you will notice this kind of code throughout all the samples.

### Try It Out: Using the getPort() Method

Now that we know how to build a `QName` instance, we can build a client application that invokes our stock quote service.

1. Save the above WSDL document in a file called `StockQuote.wsdl`, saved in a `\Chp06\ServiceTest` directory

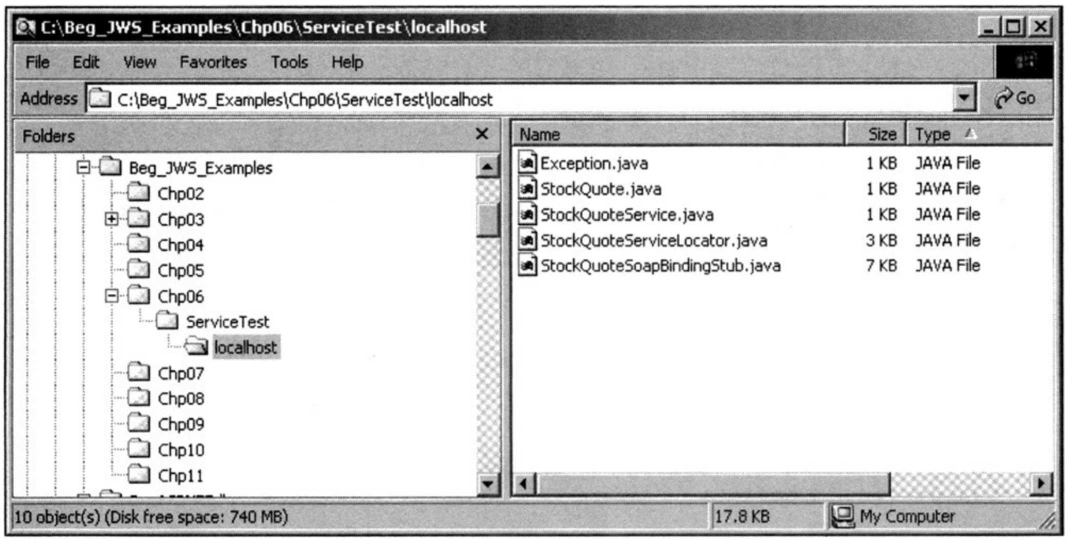
2. We now want to generate the `StockQuote` service interfaces and classes, similar to those we saw earlier in the chapter. To do this we need to run the WSDL2Java tool. First let's make sure we have the `wsdl4j.jar` file that ships with Axis in our classpath:

   ```
   set classpath=%classpath%;%axisDirectory%\lib\wsdl4j.jar
   ```

   Now run the following command to generate the necessary classes (assuming we're in the `\Chp06\ServiceTest` directory):

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

This will create five `.java` files in a new `\localhost` subdirectory:



C:\Beg_JWS_Examples\Chp06\ServiceTest\localhost

| Name | Size | Type |
|---|---|---|
| Exception.java | 1 KB | JAVA File |
| StockQuote.java | 1 KB | JAVA File |
| StockQuoteService.java | 1 KB | JAVA File |
| StockQuoteServiceLocator.java | 3 KB | JAVA File |
| StockQuoteSoapBindingStub.java | 7 KB | JAVA File |

10 object(s) (Disk free space: 740 MB) 17.8 KB My Computer

3. Now compile these generated classes:

```
javac localhost\*.java
```

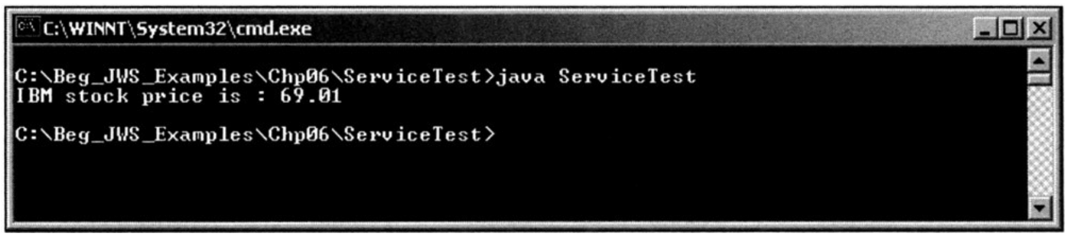
4. Write the following example code in `ServiceTest.java` in the `\Chp06\ServiceTest` directory:

```
import javax.xml.rpc.*;
import javax.xml.namespace.*;

import localhost.*;

public class ServiceTest {

  public static void main(String args[]) throws java.lang.Exception {
    QName portQN = new
      QName("http://localhost:8080/axis/services/StockQuote",
      "StockQuote");
    QName serviceQN = new
      QName("http://localhost:8080/axis/services/StockQuote",
      "StockQuoteService");
    Service service = ServiceFactory.newInstance().createService(
      new java.net.URL("file:stockquote.wsdl"), serviceQN);

    StockQuote port = (StockQuote) service.getPort(portQN, StockQuote.class);
    System.out.println("IBM stock price is : " + port.getQuote("IBM"));
  }
}
```

5. Finally execute our ServiceTest class:

```
java ServiceTest
```

We'll get our now familiar stock price result:



C:\WINNT\System32\cmd.exe

```
C:\Beg_JWS_Examples\Chp06\ServiceTest>java ServiceTest
IBM stock price is : 69.01

C:\Beg_JWS_Examples\Chp06\ServiceTest>
```

## How It Works

Getting back to the `getPort()` method, we can see that the returned object is of type `java.rmi.Remote`:

```
public java.rmi.Remote getPort(QName portName, Class proxyInterface)
    throws ServiceException;
```

Here is where the service definition interface comes into play. In the `StockQuote` example that we used earlier, this service definition interface is the `StockQuote` interface. A client can now downcast the returned object from the `getPort()` method to an instance of `StockQuote`. Moreover, the class of the returned interface is passed to the method in the second parameter. Here is how a client would use this method:

```
...
QName portQN = new
   QName("http://localhost:8080/axis/services/StockQuote",
  "StockQuote");
QName serviceQN = new
  QName("http://localhost:8080/axis/services/StockQuote",
  "StockQuoteService");

Service service = ServiceFactory.newInstance().createService(
  new java.net.URL("file:stockquote.wsdl", serviceQN);
StockQuote port = (StockQuote) service.getPort(portQN, StockQuote.class);
System.out.println("The current stock price is " + port.getQuote("IBM"));
...
```

In this example, the `Service` object is instantiated using the `ServiceFactory` class, passing the URL of the WSDL document to it, plus the fully qualified name of the service within that document. There are other ways of creating a `Service` object, and we will revisit this later in an example. Then, the `getPort()` method is used to return a proxy that can be used by the client to invoke the service.

This client code reads the WSDL document for the web service at run time and can therefore react to changes made in that document. However, it takes advantage of the generated `StockQuote` class. Thus, we see a mix of the static and the dynamic invocation models here. Note that the location of the service, its URL, is not hard-coded in the client but it is read from the WSDL file at run time. This is done by using the `getPort()` method. The method will read the endpoint definition, or location, from the WSDL document.

### Other Variations

Let us look at some other variations. The following method is a variation of the method described above:

```
public java.rmi.Remote getPort(Class serviceDefInterface)
    throws ServiceException;
```

It does not require a port name to be passed and is used typically when there is only one port defined in the WSDL document.

The `getPorts()` method returns a collection of all ports defined in a WSDL document:

```
public java.util.Iterator getPorts();
```

In fact, it returns an `Iterator` allowing us to retrieve the fully qualified names of the ports as instances of the `QName` class. These names can then be used to retrieve individual proxy objects for each port using one of the `getPort()` methods described above.

### The createCall() Method

Moving on, we create a `Call` object from a given port name as shown in the code snippet below. A `Call` object can be used to make an invocation of a service, if no generated service definition interface (`StockQuote` in our example) is available. You will note that the programming style we use in this case is very different from the example above, where the returned dynamic proxy was used. If you are familiar with the use of reflection in the Java language, you may find that this is a similar style:

```
    public Call createCall(QName portName) throws ServiceException;
```

The `Call` interface defines methods for setting parameters for a request. For example, coming back to the `StockQuote` example, we can set one parameter on the request for the ticker symbol of the requested stock. We can also set things like the name of the operation, the endpoint address, the encoding style, and other attributes on a `Call` object. Let us see this method now in practice.

#### Try It Out: Using the createCall() Method

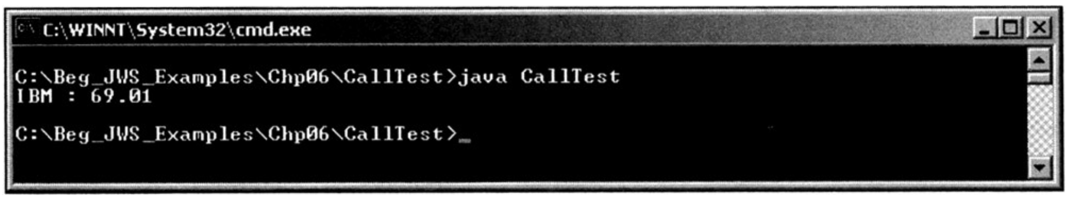
1. Save the following code example in the `CallTest.java file`:

```
import javax.xml.rpc.*;
import javax.xml.namespace.*;

public class CallTest {

  public static void main(String args[]) throws java.lang.Exception {
    Service service = ServiceFactory.newInstance().createService(null);
    Call call = service.createCall();
    call.setTargetEndpointAddress(
      "http://localhost:8080/axis/services/StockQuote");
    call.setOperationName(new QName(
      "http://localhost:8080/axis/services/StockQuote", "getQuote"));

    QName stringQN = new QName("http://www.w3.org/2001/XMLSchema",
      "string");

    call.addParameter("in0", stringQN, ParameterMode.IN);
    call.setReturnType(stringQN);

    String res = (String) call.invoke(new Object[] {"IBM"});

    System.out.println("IBM : " + res);
  }
}
```

2. Run the following commands for compiling and executing the code:

```
javac CallTest.java
java CallTest
```

The output will be as follows:



```
C:\Beg_JWS_Examples\Chp06\CallTest>java CallTest
IBM : 69.01

C:\Beg_JWS_Examples\Chp06\CallTest>_
```

### How It Works

Using this approach, we have much finer control over how an individual request is built – and how the response is handled. First, we have to build a `Service` object and retrieve a `Call` object from it:

```
...
     Service service = ServiceFactory.newInstance().createService(null);
     Call call = service.createCall();
...
```

Now we can set attributes on the `Call` object. Note that these attributes can be retrieved from the WSDL document.

For example, let us look at the `in0` parameter, which we will add to the `Call` object next. This parameter maps to the content of the request message defined in WSDL:

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

Here is the definition of the operation and request message again, for our reference:

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

The `<wsdl:input>` element refers to a message named `intf:getQuoteRequest`. This message is defined above, with one part in it, named `in0`. The name of this part is used as the parameter name when adding the parameter to the `Call` object:

```
QName stringQN = new QName("http://www.w3.org/2001/XMLSchema",
  "string");
call.addParameter("in0", stringQN, ParameterMode.IN);
```

The JAX-RPC specification dictates that those names have to match. Moreover, if we had more than one part listed in the `<wsdl:message>` element, then each one would be mapped to a parameter on the `Call` object, and their sequence as defined in the `<wsdl:message>` element must be preserved.

Finally, we indicate what response type we expect and invoke the service:

```
    call.setReturnType(stringQN);
    String res =(String) call.invoke(new Object[] {"IBM"});
    System.out.println("IBM : " + res);
```

Note that we as client developers are responsible for making sure that the attributes we set on the `Call` object are correct according to the WSDL definition. If we set one of these values incorrectly (for example, by using the wrong order of parameters), a run-time error may occur later when we call the service. This sequence can also be defined explicitly by using the `parameterOrder` attribute on the `<operation>` element.

Earlier, we described the case where a proxy object is created at run time, which can then be used to invoke a service. The right request message is built based on the definitions in the WSDL document. Here, we have more fine-grained control over how the message is built – and also how the response is handled, but we have to write more code manually, and we have to make sure that this code properly maps the WSDL definition.

Let us look at some other variations. Above, we have looked at one method that we can use to obtain a `Call` object. The `Service` interface defines a number of methods that allow us to create and retrieve `Call` objects, which can then be used similarly to the example above. These methods differ in how we define which operation we would like to be represented by the `Call` object. There is also a method to obtain an array of `Call` objects, one object per operation defined in the WSDL document, namely the `getCalls()` method.

## Type Mapping

One method on the `Service` interface that we have not talked about yet is the `getTypeMappingRegistry` method. Here is its signature:

```
    public TypeMappingRegistry getTypeMappingRegistry();
```

This is a good time to talk in more detail about type mapping. The goal is pretty simple: web services technology is based on the exchange of XML messages. We want to build applications in Java, so we need to find a way to convert XML constructs into Java objects. That is where the type mapping registry comes in handy. This registry contains an entry for each data type that a web service deals with, namely its XML type definition, Java type definition, and how to convert one into the other. This last part is defined by a couple of interfaces called `Serializer` and `Deserializer`. A `Serializer` turns a Java object into an XML string, and a `Deserializer` does the opposite.

Most implementations of JAX-RPC will come with a set of predefined serializers and deserializers, which can convert the most common data types. So, if we use basic types like `String`, `Integer`, or `Boolean` on our web services interface, we won't have to do anything. The mapping between XML Schema types and Java types for those is well defined.

On top of the `Serializer` and `Deserializer` classes for the basic types, the Apache Axis package contains a `BeanSerializer` class and a `BeanDeserializer` class. These two classes can handle the conversion of a JavaBean into an XML document and vice versa.

Effectively, this means that we should not have to worry about type mapping to much, at least not in the beginning stage. It may become more of an issue if complex data types are exchanged via a web service link. Still, in most cases, we will use tooling to create the right serializer and deserializer code.

## JAX-RPC and SOAP

The JAX-RPC API was defined in a way that lets us use it independently of the protocol that is used to invoke a web service. However, most if not all web services today use SOAP as the invocation protocol. So there is a special section in the specification that explains how to use JAX-RPC with respect to SOAP only.

Interaction with a web service over SOAP can happen in one of two ways, or styles. One style is called `rpc` style and the other one is called `document` style (for more information on this refer to Chapter 4). In short, `rpc` style means that the invocation of a web service is viewed as a function invocation, where parameters are passed into the function and a result value is received back. The `document` style means that we are sending an XML document to a web service, and may or may not get another XML document back as a response. We will try below to explain common rules of thumb to define which one to use, and you will find other references to this throughout the book.

On top of the invocation style, there are two main ways of encoding data into a SOAP message: one is to use the default SOAP encoding as defined in the SOAP specification, the other one is called **literal XML.** With literal encoding (or rather, no encoding at all) you don't encode any of the data, but add a chunk of XML to the SOAP body.

In almost all cases, only two combinations of these are used: web services either support RPC-style invocation with default SOAP encoding, or they support document style with literal XML encoding. The JAX-RPC specification requires that any implimentation of the API supports the two combinations mentioned above – the other possible combinations are optional. In fact, the specification requires that the client-side API does not differ between the two cases, so that we can use web services for both styles in the same way. There is an exception to that rule in a case where there is no easy mapping of a type defined in WSDL to a Java type. We will get to that in a second.

So when do we choose one way over the other? In many cases, the choice will be determined by the implementation of the web service.

Let's assume we have an application that can process XML documents as part of its external interface. It may store XML documents in a database, or process them in some other way. Now we want to add a webservices layer to that application, to make it accessible via SOAP, just to name one option. In that case, document style with literal XML encoding will probably be the easier choice, because we can define the XML constructs that our application already deals with and define them as messages in our WSDL definition for the new service. All of the parsing and interpreting of these XML messages is already done within our application, so there is no need to add another layer to it where this is done again.

On the other hand, we may have an application that has a pure Java-based external interface. Clients using this application pass Java objects to it and get Java objects returned. In those cases, if we add a web services wrapper around the application, we will want to support RPC style with default SOAP encoding as this gives us a more object-based view on a web service, where we send parameters to the service and get a return value. This is the reason that the style attribute in the WSDL bindings for SOAP deserves its name. The question is whether our service lets clients make a remote procedure call, or whether we exchange documents with our clients. Obviously, there is gray area in between those, and some tools don't even give a choice to begin with and simply force the developer into one style over the other.

The JAX-RPC specification states that interfaces do not change between the two styles. In other words, you cannot tell from the generated interfaces which style has been defined in the WSDL document for a web service. The differences between the styles are handled strictly in the underlying runtime. The specification defines how a given WSDL type definition is turned into a remote service interface in Java. In the case of basic data types, or simple structures, this is a straightforward thing, and the specification defines how to map those types into Java.

A problem, however, exists if some of the defined message parts cannot be mapped to Java. One example for this is if an XML Schema defines attributes to be part of an XML document. Attributes cannot be mapped to Java types. So what do we do? In these cases, the interface will contain an object that simply wraps the XML construct. We will have a look at an example for this in a second, but first let us see where this ‘XML object wrapper’ comes from.

There is a standard Java API, called **JAXM,** or **Java API for XML Messaging**, which describes how to send and receive XML-based messages in Java. One aspect of this specification deals with the notion of sending and receiving SOAP messages. The JAX-RPC API takes advantage of some classes that have been defined for JAXM and describe XML documents containing SOAP messages. One of these classes is called `javax.xml.soap.SOAPElement`. It provides basic methods for building and/or parsing an XML element.

JAX-RPC takes advantage of the `SOAPElement` class by saying that whenever a type cannot be mapped into Java according to the JAX-RPC rules it shows up as a `SOAPElement` in the interface. It is then up to the requester to build the right object, and up to the service provider to parse it, because no automatic way is defined to turn it into a regular object. In most cases, this will occur when the web service uses document style and literal XML encoding.

## Try It Out: Complex Type Mapping

Let's enhance and modify the WSDL document for the stock quote web service that we have been using before, to see how this works.

1. Modify the WSDL document:

```
<?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-imp1"
  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"
  xmlns:types="http://localhost/StockQuote">
  <types>
    <xsd:schema elementFormDefault="qualified"
      targetNamespace="http://localhost/StockQuote">
      <xsd:element name="GetQuote">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0'
              maxOccurs="1"
              name="StockSymbol"
              type="xsd:string"/>
            <xsd:element minOccurs="0"
              maxOccurs="1"
              name="AdditionalInfo"
              type="xsd:anyType"/>
          </xsd:sequence>
          <attribute name="theAttribute"
            type="xsd:string"
            use="optional"/>
        </xsd:complexType>
      </xsd:element>
```

```
        </xsd:schema>
      </types>

      <wsdl:message name="Exception"/>

      <wsdl:message name="getQuoteRequest">
        <wsdl:part name="in0" element="types:GetQuote"/>
      </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="inft:getQuoteRequest"/>
          <wsdl:output message="intf:getQuoteResponse"/>
          <wsdl:fault message="inft:Exception" name="Exception"/>
        </wsdl:operation>
      </wsdl:portType>

      <wsdl:binding name="StockQuoteSoapBinding" type="inft:StockQuote">
        <wsdlsoap:binding style="document"
          transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:opearation name="getQuote">
          <wsdlsoap:operation soapAction=""/>
          <wsdl:input>
            <wsdlsoap:body use="literal"
              namespace="http://localhost/axis/services/StockQuote"/>
          </wsdl:input>
          <wsdl:output>
            <wsdlsoap:body
              namespace="http://localhost/axis/services/StockQuote"
              use="literal"/>
          </wsdl:output>
        </wsdl:operation>
      </wsdl:binding>

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

          <wsdlsoap:address
            location="http://localhost:8080/axis/services/StockQuote"/>
        </wsdl:port>
      </wsdl:service>
    </wsdl:definitions>
```

2. Invoke the `WSDL2Java` tool in Axis:

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

This creates a number of Java classes, just as before. One of them is called `StockQuoteSoapBinding.java`:

```
package localhost;

public interface StockQuoteSoapBinding extends java.rmi.Remote {

public java.lang.String getQuote(localhost.GetQuote in0) throws
  java.rmi.RemoteException, localhost.Exception;
}
```

The `WSDL2Java` tool also creates a class named `GetQuote` for us – we will explain it below:

```
package localhost;

public class GetQuote implements java.io.Serializable {
  private java.lang.String stockSymbol;
  private java.lang.Object additionalInfo;
  private java.lang.String theAttribute; // attribute

  public GetQuote() {}
```

```
    public java.lang.String getStockSymbol() {
      return stockSymbol;
    }

   ...// additional getters and setters left out here
    // Type metadata
    private static org.apache.axis.description.TypeDesc typeDesc =
      new org.apache.axis.description.TypeDesc(GetQuote.class);

    static {
      org.apache.axis.description.FieldDesc field = new
        org.apache.axis.description.AttributeDesc();
      field.setFieldName("theAttribute");
      typeDesc.addFieldDesc(field);
      field = new org.apache.axis.description.ElementDesc();
      field.setFieldName("stockSymbol");
      field.setXmlName(new
        javax.xml.rpc.namespace.QName("http://localhost/StockQuote",
        "StockSymbol"));
      typeDesc.addFieldDesc(field);
      field = new org.apache.axis.description.ElementDesc();
      field.setFieldName("additionalInfo");
      field.setXmlName(new
        javax.xml.rpc.namespace.QName("http://localhost/StockQuote",
        "AdditionalInfo"));
      typeDesc.addFieldDesc(field);
    };

    /**
     * Return type metadata object
     */
    public static org.apache.axis.description.TypeDesc getTypeDesc() {
      return typeDesc;
    }

    /**
     * Get Custom Serializer
     */
    public static org.apache.axis.encoding.Serializer getSerializer(
      String mechType, Class _javaType, javax.xml.rpc.namespace.QName _xmlType)
    {
      return
      new org.apache.axis.encoding.ser.BeanSerializer(_javaType,
        _xmlType, typeDesc);
    };

    /**
     * Get Custom Deserializer
     */
    public static org.apache.axis.encoding.Deserializer getDeserializer(
      String mechType, Class _javaType,
      javax.xml.rpc.namespace.QName _xmlType) {
      return
        new org.apache.axis.encoding.ser.BeanDeserializer(
        _javaType, _xmlType, typeDesc);
    };
  }
```

**How it Works**

Let's walk through what is going on here. First, we changed the `getQuoteRequest` message in the WSDL definition to contain a complex type instead of a string:

```
<wsdl:message name="getQuoteRequest">
  <wsdl:part name="in0" element="types:GetQuote"/>
</wsdl:message>
```

This complex type, called `GetQuote`, is described in an XML Schema, embedded in the `<types>` element. Note that the complex type contains two elements and one attribute. One element, called `StockSymbol`, is a string. The other one, called `AdditionalInfo`, is an `anyType`. You will notice later that our JAX-RPC implementation, the Apache Axis package, maps this type to a Java object. The optional attribute is called `theAttribute` and is a string:

```
<xsd:schema elementFormDefault="qualified"
  targetNamespace="http://localhost/StockQuote">
  <xsd:element name="GetQuote">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element minOccurs="0"
          maxOccurs="1"
          name="StockSymbol"
          type="xsd:string" />
        <xsd:element minOccurs="0"
          maxOccurs="1"
          name="AdditionalInfo"
          type="xsd:anyType" />
      </xsd:sequence>
      <attribute name="theAttribute"
        type="xsd:string"
        use="optional" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Besides adding a complex type, we have changed the SOAP operation style from `rpc` to `document`, and the encoding from the SOAP default `encoding` to `literal`:

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

The `StockQuoteSoapBinding` interface looks similar to the service interface that we have seen before. The only difference is that the `getQuote()` method now does not take a string as its only parameter, it takes an object of type `localhost.GetQuote` instead:

```
...
public java.lang.String getQuote(localhost.GetQuote in0) throws
    java.rmi.RemoteException, localhost.Exception;
...
```

This parameter maps to the `GetQuote` type definition in the WSDL document. The `WSDL2Java` tool generated the `GetQuote` class for us to represent this data type. There are a few interesting things to note about this class. First of all, it contains properties that map to the elements of the `GetQuote` type. There is also the attribute, `theAttribute`, mapped to a property of this class. The JAX-RPC specification defines support for attributes as optional, so other implementations may not be able to handle it, or may even deal with it in a different way (for example, by using the `SOAPElement` class out of JAXM).

```
public class GetQuote implements java.io.Serializable {
  private java.lang.String stockSymbol;
  private java.lang.Object additionalInfo;
  private java.lang.String theAttribute; // attribute
...
}
```

The `GetQuote` class also has a property called `typeDesc`, which contains metadata about the structure of the mapped XML document. Here you can see that the attribute is handled differently from the elements:

```
// Type metadata
private static org.apache.axis.description.TypeDesc typeDesc =
  new org.apache.axis.description.TypeDesc(GetQuote.class);
```

```
static {
  org.apache.axis.description.FieldDesc field = new
    org.apache.axis.description.AttributeDesc();
  field.setFieldName("theAttribute");
  typeDesc.addFieldDesc(field);
  field = new org.apache.axis.description.ElementDesc();
  field.setFieldName("stockSymbol");
  field.setXmlName(new
    javax.xml.rpc.namespace.QName("http://localhost/StockQuote",
    "StockSymbol"));
  typeDesc.addFieldDesc(field);
  field = new org.apache.axis.description.ElementDesc();
  field.setFieldName("additionalInfo");
  field.setXmlName(new
    javax.xml.rpc.namespace.QName("http://localhost/StockQuote",
    "AdditionalInfo"));
  typeDesc.addFieldDesc(field);
};
```

This type description object is also used by two other methods, namely the `getSerializer` and the `getDeserializer` methods. The `Serializer` and `Deserializer` classes are in charge of turning the appropriate XML pieces in the SOAP message into Java and vice versa:

```
public static org.apache.axis.encoding.Serializer getSerializer(
  String mechType, Class _javaType,
  javax.xml.rpc.namespace.QName _xmlType) {
  return
  new org.apache.axis.encoding.ser.BeanSerializer(
    _javaType, _xmlType, typeDesc);
};

/**
 * Get Custom Deserializer
 */
public static org.apache.axis.encoding.Deserializer getDeserializer(
  String mechType, Class _javaType,
  javax.xml.rpc.namespace.QName _xmlType) {
  return
    new org.apache.axis.encoding.ser.BeanDeserializer(
    _javaType, _xmlType, typeDesc);
};
```

Let's now look at the same WSDL document but with a few modifications added to it.

### Try It Out: Complex Type Mapping with SOAP Encoding

We can now take the same WSDL document as a basis and see what happens when we change the invocation style to `rpc` and the encoding style to `literal`.

1. Modify the code example accordingly:

```
<?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"
  xmlns:types="http://localhost/StockQuote">
  <types>
    <xsd:schema elementFormDefault="qualified"
      targetNamespace="http://localhost/StockQuote">
      <xsd:element name="GetQuote">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0"
              maxOccurs="1"
              name="StockSymbol"
              type="xsd:string" />
```

```
                  <xsd:element minOccurs="0"
                    maxOccurs="1"
                    name="AdditionalInfo"
                    type="xsd:anyType" />
                </xsd:sequence>
                <attribute name="theAttribute"
                  type="xsd:string"
                  use="optional"/>
              </xsd:complexType>
            </xsd:element>
          </xsd:schema>

        </types>
        <wsdl:message name="Exception"/>

        <wsdl:message name="getQuoteRequest">
          <wsdl:part name="in0" element="types:GetQuote"/>
        </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
                use="encoded"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://localhost/axis/services/StockQuote"/>
            </wsdl:input>
            <wsdl:output>
              <wsdlsoap:body
                namespace="http://localhost/axis/services/StockQuote"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                use="encoded"/>
            </wsdl:output>
          </wsdl:operation>
        </wsdl:binding>

        <wsdl:service name="StockQuoteService">
          <wsdl:port binding="intf:StockQuoteSoapBinding" name="StockQuote">
            <wsdlsoap:address
              location="http://localhost:8080/axis/services/StockQuote"/>
          </wsdl:port>
        </wsdl:service>
      </wsdl:definitions>
```

2. Use the `WSDL2Java` tool in Axis again to generate the client code for this new service definition:

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

### How It Works

The only difference between this WSDL document and the one in the previous example is the invocation style and encoding. Both of these values are defined in the `<wsdlsoap:binding>` element:

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

Most of the generated code is exactly the same. Clients using this code will not be different. In other words, the interface hides the details of these things from the client developer.

## Non-SOAP Web Services

Most web services today are invoked using SOAP over HTTP. This makes sense, since web services technology started out as something that would allow applications to communicate with each other over the Internet in a standard way.

What we see today, however, is a trend towards building service-oriented architecture. While the concept has been around for some time, we now have a widely used mechanism to implement it, namely, web services. This expands the concept of application-to-application communication beyond the Internet and SOAP over HTTP. Applications are connected internally, in an intranet, or even on the same computer. For an application developer, the type of protocol used to invoke a service should be transparent. The run time system should pick the best protocol depending on how and where a service is deployed.

For example, assume that a JavaBean provides some business logic. To make this business logic available as a web service, we typically add or generate a servlet that can receive SOAP over HTTP requests, for example, by using the Apache Axis package. This servlet will then map request messages for the web service into Java calls to the JavaBean. But what if the requester of the service happens to be running in the same process, like the JavaBean with the business logic? Obviously, we can spare the overhead of creating an XML message and sending it over HTTP. We might as well make a local Java call.

### Non-SOAP Protocol Bindings in WSDL

To allow this kind of optimization, we need to find a way for a service to describe how it can be accessed over different protocols. The WSDL `<binding>` element is the perfect place for this. In fact, it is meant for exactly this purpose. The WSDL specification defines protocol bindings for SOAP and HTTP, but it also states that new bindings can be defined.

So, in our example above, we could define bindings that allow a local Java call instead of going over SOAP. Note, however, that no mechanism exists today to pick the best protocol to be used at run time. The type of binding to be used is still a decision made while developing the client code. Future web services implementations may offer such functionality.

**Using Java Bindings is WSDL**

Here is an example of what the Java bindings for our `StockQuote` example could look like:

```
    <?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:java="http://schemas.xmlsoap.org/wsdl/java/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      ...
      <wsdl:binding name="StockQuoteJavaBinding" type="intf:StockQuote">
        <java:binding/>
      </wsdl:binding>

      <wsdl:service name="StockQuoteService">
        <wsdl:port binding="intf:StockQuoteJavaBinding" name="StockQuote">
          <java:address class="com.wrox.jws.StockQuote"/>
        </wsdl:port>
      <wsdl:service>
```

```
</wsdl:definitions>
```

The WSDL listing above contains the regular definitions for the messages and operations of the service. What is interesting here, and why this example is different from all the others that we have looked at so far, is the `<binding>` and `<port>` elements.

Let us look at the `<binding>` element first:

```
<wsdl:binding name="StockQuoteJavaBinding" type="intf:StockQuote">
  <java:binding/>
</wsdl:binding>
```

As usual, the `<wsdl:binding>` element contains a child element that indicates the specific protocol that is supported. In this case, it is the `<java:binding>` element. Here, this element is empty, because there is no protocol-specific information needed.

The `<port>` element contains the location of the service implementation; in this case it is just a Java class:

```
<wsdl:port binding="intf:StockQuoteJavaBinding" name="StockQuote">
  <java:address class="com.wrox.jws.StockQuote"/>
</wsdl:port>
```

At runtime, we can interpret this binding by simply invoking a method on the Java class as defined in the `<java:address>` element.

**Note** Note that WSDL documents can contain multiple binding and port elements, so that we can define several ways of accessing a web service. Clients can then choose the method that is best for them.

### Other Examples

Another example for a useful WSDL protocol binding is the **Java 2 Connector Architecture (JCA).** This architecture, which is part of the J2EE standard, defines how certain back-end non-J2EE environments called **Enterprise Information Systems (EIS)** can be connected to a J2EE application server in a transactional and secure way. Explaining the connector architecture goes well beyond the scope of this book, but what we can note here is that it is the recommended way in Java to connect to existing legacy applications and databases. This allows for integration of existing environments into new J2EE-based solutions. The J2EE application server can include the legacy backend into its transactional control and connections to backends can be pooled and shared among clients.

We mentioned earlier that web services technology plays a big role in Enterprise Application Integration. AJCA connector provides the ‘plumbing’ to make this possible. We can invoke requests to an EIS via the connector, which will then pass it on to the backend. Thus, if we can define a WSDL binding for JCA as the protocol, we can view the backend that we want to integrate as a web service and leave the hard work of getting the invocation done to the connector.

Here is what the stock quote binding for a sample JCA connector, going over CICS in this case, would look like:

```
<?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:cics="http://schemas.xmlsoap.org/wsdl/cics/"
  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="StockQuoteJCABinding" type="intf:StockQuote">
    <format:typemapping style="COBOL" encoding="COBOL">
      <format:typemap typename="Symbol"
        formattype="StockSymbol.ccp:SYMBOL"/>
```

```
         </format:typemapping>
         <operation name='getQuote'>
           <cics:operation functionName="GETQUOTE"/>
        <input>
          ...
        </input>
        <output>
          ...
        </output>
      </wsdl:binding>

      <wsdl:service name="StockQuoteService">
        <wsdl:port binding="intf:StockQuoteJCABinding" name="StockQuote">
          <cics:address connectionURL="..." serverName="CICS_A" />
        </wsdl:port>
      </wsdl:service>
    </wsdl:definitions>
```

Note that these bindings are not standardized. We won't go into any detail here about how to build and a JCA connector, or how exactly the bindings above are built and used. This was just to show the idea of non-SOAP binding protocols and how they could be mapped to a WSDL document. Obviously, if these bindings were part of a standard, that would improve portability between implementations by different vendors, making it desirable that such standardization takes place in the near future.

In order to use web services that have been defined this way, we need a Java client implementation that can interpret these protocol bindings at run time and generate the right kind of invocation.

## The Web Services Invocation Framework (WSIF)

So far in this chapter, we have looked at static and dynamic ways to invoke web services. In many cases, this involved the generation of code, based on the WSDL definition for a service. Thus, the client code introduced so far never showed a truly dynamic invocation model. In JAX-RPC, we either generated a service interface and used a proxy via the `javax.xml.rpc.Service.getPort` method or we created a `Call` object and set all of the parameters of the invocation manually.

In this section, we want to focus on an alternative approach to web service invocation, which is more dynamic and, most of all, completely independent from any protocol. Thus, the same client code can support not only invocations via SOAP over HTTP, but also invocations that are simple local Java calls or calls over a JCA connector, just to name two examples.

The WSIF provides an API and runtime that does exactly what we are looking for – invocation of web services independent from the supported protocol. Originally developed by IBM, it is in the process of being turned into an open source project at the Apache foundation. At the time of this writing, there are nightly builds of WSIF at Apache, but no external driver has been released yet. As part of this transition of WSIF into an open source project, the package names will have changed from being based on `com.ibm.wsif` to `org.apache.wsif`. We will stick with the `com.ibm.wsif` package names here, but you should have no problem converting the code to work with latest release from Apache once there are external releases of it.

WSIF depends on a package called WSDL4J, which is an open source Java API for reading and creating WSDL documents (WSDL4J is also the base for a proposed standard Java API to process WSDL. The proposal is captured under JSR 110: http://jcp.org/jsr/detail/110.jsp). In other words, it provides us with an interface that lets us interpret an existing WSDL document, or build a brand new one. It contains Java wrapper classes for all of the elements defined in WSDL. For example, there is a class called `javax.wsdl.Operation`, and also one called `javax.wsdl.PortType`. At the top of the WSDL4J API class hierarchy is a class called `javax.wsdl.Definition`, which captures most of the WSDL document.

Invoking a web service through WSIF is based on the WSDL document that describes that service. No code is generated upfront; all handling is done at run time.

### Try It Out: Invoking a Web Service with WSIF

Let us step through an example that shows how to invoke a web service with WSIF. We will invoke the stock quote web service that we have used before. Assume here that a file named `stockquote.wsdl` exists.

1. We first need to get hold of the WSIF package. At the time of writing this was easier said than done because it was still in transition from IBM to Apache. Eventually you should be able to download it from the Apache site, but at the time of writing we downloaded it from Alphaworks (http://alphaworks.ibm.com/tech/wsif).

2. Save the following example code in `WSIFSample.java`:

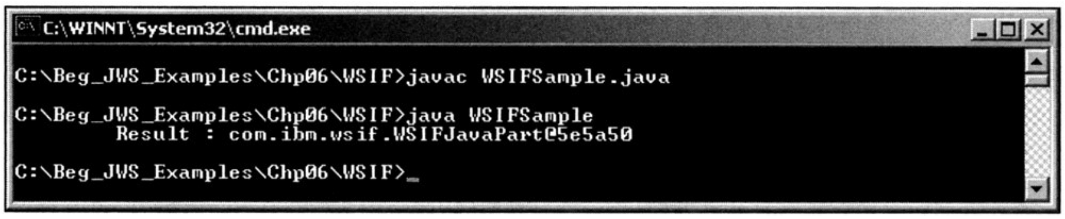
```
    import javax.wsdl.*;
    import org.xml.sax.*;
    import java.io.*;
    import com.ibm.wsdl.xml.*;
    import com.ibm.wsif.*;

    public class WSIFSample {
```

```
    public static void main(String[] args) throws Exception {
      Definition def = WSDLReader.readWSDL(null,
        new InputSource("http://localhost:8080/axis/stockquote.wsdl"));
      WSIFDynamicPortFactory dpf = new WSIFDynamicPortFactory(def);
      WSIFPort port = dpf.getPort();
      WSIFMessage input = port.createInputmessage();
      WSIFMessage output = port.createOutputMessage();
      WSIFMessage fault = port.createFaultMessage();

      WSIFPart inputPart =
        new WSIFJavaPart(java.lang.String.class, "IBM");
      input.setPart("in0", inputPart);
      port.executeRequestResponseOperation("getQuote",
        input, output, fault);
      WSIFPart outputPart = output.getPart("return");
      System.out.println("\tResult : " + outputPart);
    }
  }
```

3. Run the following commands to compile and execute it:

```
javac WSIFSample.java
java WSIFSample
```

**Note** The IBM version of WSIF also requires an Apache SOAP installation but presumably when the project is fully moved to Apache this will be replaced by Axis.



```
C:\Beg_JWS_Examples\Chp06\WSIF>javac WSIFSample.java

C:\Beg_JWS_Examples\Chp06\WSIF>java WSIFSample
        Result : com.ibm.wsif.WSIFJavaPart@5e5a50

C:\Beg_JWS_Examples\Chp06\WSIF>_
```

### How it Works

Notice that this approach focuses a lot on the definitions made in the WSDL document. That is everything we need to do. We will now walk through a code sample that shows how these steps look in Java code.

Step 1 was to read in the WSDL document and store it in a `javax.wsdl.Definition` object (note that the package declarations are omitted to make the code more readable):

```
Definition def = WSDLReader.readWSDL(null,
  new InputSource("http://localhost:8080/axis/stockquote.wsdl"));
```

Next, the dynamic port factory is created. All type mapping between the XML types declared in the WSDL and the Java classes that are used in the client is defined on this factory:

```
WSIFDynamicPortFactory dpf = new WSIFDynamicPortFactory(def);
```

Note that to make a dynamic call, Java classes must already exist for the types that are declared in the `<types>` element of the WSDL document (or in the schema that is included via an `<import>` element). In our case, this is not needed since all the types the stock quote web service handles are basic data types. In case of a WSDL definition that defines complex types on its interface, a mapping is needed, and the port factory is the place to add such mappings.

**Important** The next release of WSIF is scheduled to take advantage of yet another technology that will define a way to handle this dynamically. This dynamic handling is done through the notion of generic Java objects that map an XML tree. You can find out more about this technology at http://www.alphaworks.ibm.com/tech/jrom.

Now we can retrieve the `com.ibm.wsif.WSIFPort` object and wrap the messages and parts:

```
WSIFPort port = dpf.getPort();

WSIFMessage input = port.createInputMessage();
WSIFMessage output = port.createOutputMessage();
WSIFMessage fault = port.createFaultMessage();

WSIFPart inputPart = new WSIFJavaPart(java.lang.String.class, "IBM");
input.setPart("in0", inputPart);
```

If we wanted to write a completely generic client, that is, a client that can read and interpret any **`WSDL`** document, we would have to add code here that reads the part definitions and fills in appropriate values. For example, in our stock quote case, the input message contains one part of type string. We can map this into the code above. To make this dynamic, we would have to read the part definition from the WSDL document. Then we would detect, for this case, that the input message contains one part that is of type `String`. Given this information, we could build the `com.ibm.wsif.WSIFPart` object. Note, however, that this can turn into rather lengthy code, especially in cases where we have to deal with complex data types.

We are now ready to invoke the service with `executeRequestResponseOperation()` on the `com.ibm.wsif.WSIFPort` object:

```
port.executeRequestResponseOperation("getQuote", input, output, fault);
```

This is the only place where the `<operation>` is referenced by its name. The result of the call can now be retrieved from the output message:

```
WSIFPart outputPart = output.getPart("return");
```

An important thing that this code shows is that the service is invoked independently from the actual protocol that is used. In other words, this code will work for a SOAP binding as well as for an HTTP binding of the service, and no additional client package is required. The WSIF uses something called a **dynamic provider** for this. Based on the specification in the WSDL document, a provider that implements the right protocol is used. For example, if `<soap:binding>` is specified, a class called `WSIFDynamicProvider_ApacheSOAP` is used to make the call. However, this never shows up in the client code. This means that we could write our own extension to WSIF to support our own protocol bindings, by supplying a specific dynamic provider.

This example shows how we can deal with a web service on a WSDL level, without requiring any knowledge about the protocol it supports and how the service was implemented. The use of an abstract API like WSDL4J together with additional support for dynamic invocation as provided by the WSIF makes this possible.

## Summary

In this chapter, we have seen that there are different ways in which a client can invoke web services. In the static model, tools are used to generate client-side classes from the WSDL definition of a service. These classes can then be used by client developers who, consequently, don't have to worry about the network protocol that is used to invoke a service, or how to build responses for it, and interpret responses from it.

In the dynamic model, client-side objects are generated at run time, from the WSDL definition of the service. This still allows client code to be developed without any knowledge about the actual invocation mechanism. In other words, client programmers don't have to know the details of SOAP to use a service. There are variations to the dynamic model, one of which promotes client classes that are completely independent from any kind of predefined code.

Both static and dynamic invocations of web services can be done using the classes and interfaces defined in the JAX-RPC API. This API will standardize the access to web services for the J2EE world. For example, it specifies how to obtain a reference to a proxy object that can handle interaction with a WSDL-based web service. These proxies can be generated or created at run time by the JAX-RPC environment.

Finally, we learned how web services technology is not restricted to SOAP. Invocations to existing services can be made over a variety of protocols, for example, via RMI/IIOP or JMS. The supported protocol can be described in the `<binding>` element in WSDL. A run-time environment that takes advantage of these new bindings is the WSIF. It allows dynamic clients to be built strictly based on a service WSDL document. These clients can then build valid requests for that web service based on the supported access protocols.