CHAPTER 4
SOAP-Based Web Services

JAX-WS is an API for producing and consuming REST-style and SOAP-style web services. Chapter 2 introduced, with a RESTful example, the JAX-WS `@WebServiceProvider` annotation, which can be used for either a REST-style or a SOAP-based service. This chapter introduces the `@WebService` annotation, in effect a refinement of `@WebServiceProvider`; a `@WebService` is emphatically SOAP-based. JAX-WS is thus sufficiently rich and varied that it might better be described as a collection of APIs. JAX-WS is the successor to JAX-RPC, which derives from the XML-RPC discussed in Chapter 1. The reference implementation for JAX-WS is part of the open source GlassFish project and is named GlassFish Metro or just Metro for short. The current version of JAX-WS is 2.2.x. JAX-WS is officially part of enterprise Java but, with JDK 1.6 or greater, JAX-WS services can be compiled and published using only core Java. SOAP-based services in JAX-WS can be published with a standard Java web server such as Tomcat or Jetty; there is also a convenient `Endpoint` publisher, used earlier to publish a RESTful service.

Apache Axis2 is an alternative implementation of JAX-WS. Axis2, the successor to Axis, is based on JAX-WS but has additional features. Yet another JAX-WS implementation is Apache CXF. This chapter focuses on the Metro implementation of JAX-WS, but the next chapter includes an Axis2 service and client. The Metro, Axis2, and Apache CXF implementations of JAX-WS are sufficiently close that a programmer fluent in one implementation should be able to move easily into any other implementation.

SOAP is an XML dialect that has two W3C-sanctioned versions: 1.1 and 1.2; SOAP is officially no longer an acronym. The differences between the versions of SOAP are more about infrastructure than API. For example, the media type for a SOAP 1.1 message is `text/xml`, whereas this type changes in SOAP 1.2 to `application/soap+xml`. The SOAP 1.2 processing model is more thoroughly and precisely specified than is the SOAP 1.1 model; SOAP 1.2 has an official *binding framework* that opens the way to using transport protocols other than HTTP for delivering SOAP messages. In practice, however, HTTP remains the dominant transport for both SOAP 1.1 and SOAP 1.2. In the major SOAP

frameworks of Java and DotNet, SOAP 1.1 is the default, but both systems support SOAP 1.2 as well.

SOAP has a *basic profile*, which comes from the WS-I (Web Services Interoperability) consortium to encourage and support interoperability among web service languages and technologies. Beyond the basic profile are various initiatives, some of which are covered in later chapters; these initiatives (for instance, WS-Reliability and WS-Security) often are grouped under the acronym WS-*. WSIT (Web Services Interoperability Technology) is a related set of guidelines that promotes interoperability specifically between Java and DotNet, in particular DotNet's WCF (Windows Communication Foundation). WCF is a framework for developing service-oriented applications that would include but also go beyond SOAP-based web services *(metro.java.net/guide)*. From time to time, this chapter refers to one or another SOAP specification in order to clarify a particular service or client, but the emphasis in this chapter remains on coding services and their clients. Unless otherwise specified, the examples are in SOAP 1.1.

A SOAP-Based Web Service

JAX-WS, like JAX-RS, uses annotations, and machine-generated JAX-WS code is awash with these. The first example is stingy in its use of annotations in order to underscore exactly what is required for a SOAP-based service. Later examples introduce additional annotations.:

The `RandService` class (see Example 4-1) defines a SOAP-based service with two operations, each an annotated Java method:

- Operation *next1* takes no arguments and returns one randomly generated integer.
- Operation *nextN* takes one argument, the number of randomly generated integers desired, and returns a list (in this implementation, an array) of integers.

Example 4-1. A SOAP-based service with two operations

```
package rand;

import javax.jws.WebService;
import javax.jws.WebMethod;
import java.util.Random;

@WebService                                                    ❶
public class RandService {
    private static final int maxRands = 16;

    @WebMethod // optional but helpful annotation              ❷
    public int next1() { return new Random().nextInt(); }
    @WebMethod // optional but helpful annotation              ❸
    public int[ ] nextN(final int n) {
```

```
        final int k = (n > maxRands) ? maxRands : Math.abs(n);
        int[ ] rands = new int[k];
        Random r = new Random();
        for (int i = 0; i < k; i++) rands[i] = r.nextInt();
        return rands;
    }
}
```

The `@WebService` annotation (line 1) marks the `RandService` POJO class as a web service, and the `@WebMethod` annotation (lines 2 and 3) specifies which of the encapsulated methods is a service operation. In this example, the `RandService` class has only two methods and each of these is annotated as `@WebMethod`. The `@WebMethod` annotation is optional but recommended. In a class annotated as a `@WebService`, a `public` instance method is thereby a service *operation* even if the method is not annotated. This SOAP service code is compiled in the usual way, assuming JDK 1.6 or greater.

Recall that core Java 6 or greater includes the `Endpoint` class for publishing web services, SOAP-based (`@WebService`) and REST-style (`@WebServiceProvider`) alike. The class `RandPublisher` (see Example 4-2) is the `Endpoint` publisher for the `RandService`.

Example 4-2. An Endpoint published for the `RandService` SOAP-based web service

```
package rand;

import javax.xml.ws.Endpoint;
public class RandPublisher {
    public static void main(String[ ] args) {
        final String url = "http://localhost:8888/rs";                    ❶
        System.out.println("Publishing RandService at endpoint " + url);
        Endpoint.publish(url, new RandService());                         ❷
    }
}
```

The `publish` method used here (line 2) takes two arguments: a URL that specifies the service endpoint (line 1) and an instance of the service implementation class, in this case the `RandService` class (line 2). In the URL, the port number 8888 and the URI `/rs` are arbitrary, although a port number greater than 1023 is recommended because modern operating systems typically reserve port numbers below 1024 for particular applications (e.g., port 80 is typically reserved for HTTP requests to a web server). The `RandPublisher` as coded here runs indefinitely, but there are various way to control an `Endpoint` publisher's life span.

The web service publisher can be executed in the usual way:

```
% java rand.RandPublisher
```

The output should be similar to this:

```
Publishing RandService at endpoint http://localhost:8888/rs

com.sun.xml.internal.ws.model.RuntimeModeler getRequestWrapperClass
INFO: Dynamically creating request wrapper Class rand.jaxws.Next1
com.sun.xml.internal.ws.model.RuntimeModeler getResponseWrapperClass
Dynamically creating response wrapper bean Class rand.jaxws.Next1Response
com.sun.xml.internal.ws.model.RuntimeModeler getRequestWrapperClass
INFO: Dynamically creating request wrapper Class rand.jaxws.NextN
com.sun.xml.internal.ws.model.RuntimeModeler getResponseWrapperClass
INFO: Dynamically creating response wrapper bean Class rand.jaxws.NextNResponse
```

The first line of output is from the `RandPublisher` but the others are from the Java runtime. The dynamically created wrapper classes such as `Next1` and `Next1Response` are JAX-B artifacts that represent the incoming SOAP request (`Next1`) and the outgoing SOAP response (`Next1Response`).

Once the service is published, a utility such as *curl* can be used to confirm that the service is indeed up and running:

```
% curl http://localhost:8888/rs?xsd=1
```

This *curl* request contains the query string entry `xsd=1` that asks for the XML Schema associated with this service; the schema, like the JAX-B artifacts, is generated dynamically (see Example 4-3).

Example 4-3. The XML Schema generated dynamically for the `RandService`

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:tns="http://rand/" xmlns:xs="http://www.w3.org/2001/XMLSchema"
           version="1.0" targetNamespace="http://rand/">
  <xs:element name="next1" type="tns:next1"></xs:element>
  <xs:element name="next1Response" type="tns:next1Response"></xs:element>
  <xs:element name="nextN" type="tns:nextN"></xs:element>
  <xs:element name="nextNResponse" type="tns:nextNResponse"></xs:element>
  <xs:complexType name="next1"><xs:sequence></xs:sequence></xs:complexType>
  <xs:complexType name="next1Response">
    <xs:sequence>
      <xs:element name="return" type="xs:int"></xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="nextN">
    <xs:sequence>
      <xs:element name="arg0" type="xs:int"></xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="nextNResponse">
    <xs:sequence>
      <xs:element name="return" type="xs:int" minOccurs="0" maxOccurs="unbounded">
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

The schema will be studied carefully later. For now, the point of interest is that the schema provides a data type for each SOAP message that travels, in either direction, between the service and the client. Each message is of an XML Schema `complexType` as opposed to an simple type such as `xsd:date`, `xsd:string`, or `xsd:integer`.

In the `RandService` there are two SOAP messages (for instance, the messages `Next1` and `Next1Response`) per web service operation (in this case, the *next1* operation) because each operation implements the familiar *request/response* pattern: a client issues a request, delivered to the service as a `Next1` SOAP message, and gets a response, in this case a `Next1Response` message, in return. Accordingly, the schema contains four typed SOAP messages because the `RandService` has two operations in the request/response pattern, which means two messages per operation. The number of `complexType` occurrences in the XML Schema may exceed the total number of messages needed to implement the service's operations because special error messages, SOAP *faults*, also may be defined in the XML Schema. SOAP faults are covered in the next chapter.

The XML Schema types such as `Next1` and `Next1Response` are the XML counterparts to the JAX-B artifacts, noted earlier, with the same names. The schema types and the JAX-B types together allow the SOAP libraries to transform Java objects into XML documents (in particular, SOAP `Envelope` instances) and SOAP `Envelope` instances into Java objects. The `Endpoint` publisher's underlying SOAP libraries handle the generation of the JAX-B artifacts and the generation of the XML Schema.

Publishing a SOAP-Based Service with a Standalone Web Server

Publishing a `@WebService` with Tomcat or Jetty is almost the same as publishing a `@WebServiceProvider` (see "A RESTful Service as a @WebServiceProvider" on page 85) with these web servers. Here, for quick review, are the details.

Two configuration files are needed: the usual file *web.xml* and the additional file *sun-jaxws.xml*. Here is the *web.xml*, which would work for any implementation annotated as `@WebService` or `@WebServiceProvider`:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  <listener>
    <listener-class>
      com.sun.xml.ws.transport.http.servlet.WSServletContextListener
    </listener-class>
  </listener>
  <servlet>
    <servlet-name>jaxws</servlet-name>
    <servlet-class>
      com.sun.xml.ws.transport.http.servlet.WSServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
```

```
    </servlet>
    <servlet-mapping>
      <servlet-name>jaxws</servlet-name>
      <url-pattern>/*</url-pattern>
    </servlet-mapping>
  </web-app>
```

The Metro classes `WSServletContextListener` and `WSServlet` are in the JAR file currently named *webservices-rt.jar*, which can be downloaded with the rest of Metro JARs. A second Metro library file *webservices-api.jar* is also required. The JAR files in question should be in the *src* directory so that the Ant script can package them in the deployed WAR file. In any case, the `WSServletContextListener` parses the *sun-jaxws.xml* file. The `WSServlet` acts as the interceptor: the servlet receives incoming requests and dispatches these to the `RandService`.

The second configuration file, *sun-jaxws.xml*, is:

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints version="2.0"
           xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime">
  <endpoint implementation="rand.RandService" ❶
            name="RandService"
            url-pattern="/*"/>
</endpoints>
```

This file completes the routing by notifying the `WSServletContextListener` that the `WSServlet` should dispatch requests to a `RandService` instance (line 1).

With the two configuration files and the Metro library JARs in the *src* directory, the `RandService` can be deployed to Tomcat in the usual way:

```
% ant -Dwar.name=rand deploy
```

Once the `@WebService` has been deployed, a *curl* call or a browser can be used to verify that the service is up and running:

```
% curl http://localhost:8080/myWarFileName?xsd=1
```

If successful, this command returns the XML Schema associated with the service.

Even this first and rather simple example underscores a major appeal of SOAP-based services: underlying SOAP libraries handle the conversions between native language types (in this case, Java types) and XML Schema types. Figure 4-1 depicts the architecture.

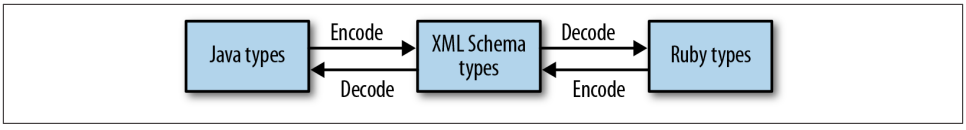

Figure 4-1. The architecture of a typical SOAP-based service

The RandService in Two Files

The `RandService` in the first example (see Example 4-1) combines, in a single source file, what JAX-WS calls the SEI (Service Endpoint Interface) and the SIB (Service Implementation Bean). The SEI specifies, at a high level that befits an interface, the service operations, and the SIB provides an implementation of the operations. A SIB can be one of the following:

- A POJO class such as `RandService` annotated as `@WebService` and encapsulating service operations, each annotated as a `@WebMethod`.
- A `@Stateless` Session EJB that is likewise annotated as a `@WebService`. EJBs in general predate JAX-WS; hence, this second type of SIB is an inviting way to expose legacy EJBs as web services.

Chapter 7 covers the EJB implementation of a `@WebService`. For now, the SIBs will be POJO classes. For convenience, most of my examples take the single-file approach, which combines the SEI and the SIB into one class annotated as a `@WebService`. The two-file approach is illustrated with the SEI `RandService` (see Example 4-4) in one file and the SIB `RandImpl` (see Example 4-5) in another file. The `RandService` is now an `interface`, whereas the `RandImpl` is a `class` that `implements` this interface.

Example 4-4. The Service Endpoint Interface for the revised `RandService`

```
package rand2;

import javax.jws.WebService;
import javax.jws.WebMethod;
import java.util.Random;

@WebService
public interface RandService {
    @WebMethod
    public int next1();
    @WebMethod
    public int[ ] nextN(final int n);
}
```

Example 4-5. The Service Implementation Bean for the revised `RandService`

```
package rand2;

import javax.jws.WebService;
import javax.jws.WebMethod;
import java.util.Random;

@WebService(endpointInterface = "rand2.RandService")
public class RandImpl implements RandService {
    private static final int maxRands = 16;

    @WebMethod
    public int next1() { return new Random().nextInt(); }
    @WebMethod
    public int[ ] nextN(final int n) {
        final int k = (n > maxRands) ? maxRands : Math.abs(n);
        int[ ] rands = new int[k];
        Random r = new Random();
        for (int i = 0; i < k; i++) rands[i] = r.nextInt();
        return rands;
    }
}
```

In the SIB class `RandImpl`, the `@WebService` interface has an attribute, the key/value pair:

```
    endpointInterface = "rand2.RandService"
```

that names the SEI. It is still important for the class `RandImpl` to employ the standard `implements` clause because only the `implements` clause prompts the compiler to make sure that the public methods declared in the SEI, in this case the two methods annotated with `@WebMethod`, are defined appropriately in the SIB.

The revised `RandService` has the same functionality as the original. The `Endpoint` publisher changes slightly:

```
    Endpoint.publish(url, new RandImpl()); // SIB, not SEI
```

The second argument to the `publish` changes to `RandImpl` precisely because, in the revision, `RandService` is an interface. In general, the second argument to the `static` version of `publish` is always the SIB. In a single-file case such as the original version of `RandService`, a single class is the combined SEI and SIB.

Clients Against the RandService

The claim that SOAP-based services are language-neutral needs to be taken on faith a bit longer. The first client against the `RandService` is in Java but the two thereafter are in C# and Perl. Starting with a Java client will help to clarify how the critical service contract, the WSDL document, can be put to good use in writing a client. The WSDL

will be studied in detail, but putting the WSDL to work first should help to motivate the detailed study.

A Java Client Against the RandService

Recall the XML Schema (see Example 4-3) that the `Endpoint` publisher generates dynamically when the `RandService` is published. The publisher likewise generates a WSDL, which can be requested as follows:

```
% curl http://localhost:8888/rs?wsdl
```

JDK 1.6 and greater ship with a utility, *wsimport*, that uses a WSDL to generate Java classes in support of programming a client against the service described in the WSDL. Here is how the utility can be used in the current example:

```
% wsimport -p client -keep http://localhost:8888/rs?wsdl
```

The `-p` flag stands for "package": the utility creates a directory named *client* and puts the generated Java code in this directory/package. The `-keep` flag generates source (*.java*) as well as compiled (*.class*) files; without this flag, only compiled files would be in the *client* directory. Sixteen files are generated in total, half source and half compiled. Among these are files with names such as `Next1` and `Next1Response`, the very names of the classes generated at the publication of the `RandService`. In any case, these client-side artifacts correspond to SOAP types described in the XML Schema document for the `RandService`.

How are the *wsimport*-generated files to be used? Two of these are of special interest:

- The class `RandServiceService` begins with the name of the published SOAP service, `RandService`, and has another `Service` stuck on the end. The `@WebService` annotation could be used to specify a less awkward name but, for now, the key point is that this class represents, to the client, the deployed web service.
- The interface `RandService` has the same name as the published service but there is a critical difference: this `RandService` is an *interface*, whereas the published `RandService` is a *class*. This interface, like any Java interface, declares methods—hence, the interface declares the operations encapsulated in published service and thereby specifies the invocation syntax for each operation. In this example, there are two such operations: *next1* and *nextN*.

The `RandServiceService` and `RandService` types are used in an idiomatic way to write the Java client against the service. The `RandClient` (see Example 4-6) is a sample client that illustrates the idiom.

Example 4-6. A Java client built with wsimport-generated artifacts

```
import client.RandServiceService;
import client.RandService;
import java.util.List;

public class RandClient {
    public static void main(String[ ] args) {
        // set-up
        RandServiceService service = new RandServiceService(); ❶
        RandService port = service.getRandServicePort();       ❷
        // sample calls
        System.out.println(port.next1());                      ❸
        System.out.println();
        List<Integer> nums = port.nextN(4);                    ❹
        for (Integer num : nums) System.out.println(num);      ❺
    }
}
```

The `RandClient` imports two types from the *wsimport*-generated artifacts: the class `RandServiceService` and the interface `RandService`. In the setup phase of the client code, the class's no-argument constructor is invoked to create an object that represents, on the client side, the service itself (line 1). Once this object is constructed, there is a *get* call with a distinct pattern:

```
service.get<name of interface type>Port() // line 2 pattern
```

In this case, the interface is named `RandService` and so the call is:

```
service.getRandServicePort() // line 2
```

This *get* method returns a reference to an object that encapsulates the two operations in the `RandService`, *next1* and *nextN*. The reference is named `port`, although any name would do, for reasons that will become clear once the WSDL is studied in detail. The `port` reference is then used to make two sample calls against the service. On a sample run, the output was:

```
53378846          // from line 3
-818435924        // from lines 4 and 5
104886422
1714126390
-2140389441
```

The first integer is returned from the call to `next1` and the next four integers from the call to `nextN`.

The `RandClient` does reveal an oddity about the *wsimport*-generated artifacts. In the `RandService`, the method `nextN` begins:

```
public int[ ] nextN(...
```

The return type is `int[ ]`, an array of `int` values. In the *wsimport*-generated interface `RandService`, the method `nextN` begins:

```
public List<Integer> nextN(...
```

The *wsimport* utility is within its rights to replace `int[ ]` with `List<Integer>`, as a `List` has a `toArray` method that returns an array; with automatic boxing/unboxing, the Java types `Integer` and `int` are interchangeable in the current context. The point is that the programmer typically needs to inspect at least the *wsimport*-generated interface, in this example `RandService`, in order to determine the argument and return types of every operation.

Companion Utilities: wsimport and wsgen

The *wsimport* utility eases the task of writing a Java client against a service that has a WSDL as the service contract. This utility has a client-side focus, although the utility can be helpful on the server side as well; a later example illustrates. The *wsgen* utility, which also ships with core Java 1.6 or greater, has a server-side focus. For example, *wsgen* can be used to generate a WSDL. The command:

```
% wsgen -cp . -wsdl rand.RandService
```

generates a WSDL file named *RandServiceService.wsdl*. However, this WSDL has a placeholder for the service endpoint rather than a usable URL:

```
...
<soap:address location="REPLACE_WITH_ACTUAL_URL"/>
...
```

When a service publisher such as `Endpoint`, Tomcat, Jetty, and the like generate the WSDL, the WSDL includes a usable URL.

The *wsgen* utility has another use. When the `RandService` is published with `Endpoint`, the publisher outputs information about dynamically generated classes, in this case `Next1`, `Next1Response`, `NetxN`, and `NextNResponse`. As noted earlier, these are JAX-B artifacts that the Java runtime uses to convert Java types into XML types and vice versa. The *wsgen* utility can be used to generate the JAX-B artifacts as files on the local system. For example, the command:

```
% wsgen -cp . rand.RandService
```

automatically creates a package/directory *rand/jaxws* and then populates this directory with *Next1.class*, *Next1Response.class*, *NextN.class*, and *NextNResponse.class*. Now if the `Endpoint` publisher is started after these files have been created, the publisher does not generate the JAX-B artifacts dynamically but instead uses the ones that *wsgen* already created.

A final, obvious point about the interaction between the Java client and the Java service deserves mention: the SOAP is completely transparent. The underlying SOAP libraries generate the SOAP on the sending side and parse the SOAP on the receiving side so that the Java code on both sides can remain agnostic about what type of payload is being sent and received. SOAP transparency is a major selling point for SOAP-based services.

A C# Client Against the RandService

The next client is in C#, a DotNet language similar to Java; DotNet has a *wsdl* utility similar to Java's *wsimport* utility. The *wsdl* utility can be targeted at the dynamically generated WSDL for the `RandService`:

```
% wsdl http://localhost:8888/rs?wsdl
```

This command generates a single file with an awkward name: *RandServiceService.cs* (see Example 4-7).

Example 4-7. A C# client, built with wsdl-generated code, against the `RandService`

```
using System;
using System.ComponentModel;
using System.Diagnostics;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Xml.Serialization;

// This source code was auto-generated by wsdl, Version=4.0.30319.1.
...
public partial class RandServiceService :
        System.Web.Services.Protocols.SoapHttpClientProtocol {
    private System.Threading.SendOrPostCallback next1OperationCompleted;
    private System.Threading.SendOrPostCallback nextNOperationCompleted;

    public RandServiceService() { this.Url = "http://localhost:8888/rs"; }
    ...
    public int next1() {
        object[] results = this.Invoke("next1", new object[0]);
        return ((int)(results[0]));
    }
    ...
    public System.Nullable<int>[]
             nextN([System.Xml.Serialization.XmlElementAttribute(
                Form=System.Xml.Schema.XmlSchemaForm.Unqualified)] int arg0) {
        object[] results = this.Invoke("nextN", new object[] {arg0});
        return ((System.Nullable<int>[])(results[0]));
    }
    ...
}
...
```

The code excised from the C# `RandServiceService` class supports asynchronous calls against the Java `RandService`. Java, too, supports both synchronous (blocking) and asynchronous (nonblocking) calls against a web service's operations, as a sample client against the `RandService` later illustrates. For now, only synchronous calls are of interest. Here is a sample C# client that uses the *wsdl*-generated code to make calls against the `RandService`:

```
class RandClient {
  static void Main() {
     RandServiceService service = new RandServiceService();       ❶
     Console.WriteLine("Call to next1():\n" + service.next1());   ❷
     Console.WriteLine("\nCall to nextN(4):");
     int?[] nums = service.nextN(4);                               ❸
     foreach (int num in nums) Console.WriteLine(num);
  }
}
```

The C# client code is simpler than its Java counterpart because the `new` operation with the no-argument constructor `RandServiceService()` (line 1) creates an object that encapsulates the client-side operations *next1* and *nextN*. The C# code does not require the `getRandServicePort()` call from the Java client. The call to *next1* (line 2) is basically the same in the C# and Java clients, but the C# call to *nextN* has unusual syntax. The return type `int?[]` (line 3) signifies an integer array that may have `null` as its value; the type `int[]` signifies an integer array that cannot be `null`. On a sample run, the C# client output is:

```
Call to next1():
680641940
Call to nextN(4):
1783826925
260390049
-48376976
-914903224
```

The C# example does illustrate language interoperability for SOAP-based services, although C# and Java are at least cousins among programming languages. The next sample client is written in a language quite different from Java.

A Perl Client Against the RandService

The final client (see Example 4-8) against the Java `RandService` is in Perl. This client makes it easy to display the SOAP messages that go back and forth between client and service; the Perl library `SOAP::Lite` has an excellent, easy-to-use tracer.

Example 4-8. A Perl client against the RandService

```
#!/usr/bin/perl -w

use SOAP::Lite +trace => 'debug';
```

```
use strict;

my $soap =                                                          ❶
    SOAP::Lite->uri('http://rand/')->proxy('http://localhost:8888/rs/');
my $num = $soap->next1()->result();                                 ❷
print "Response is: $num\n";                                        ❸
```

In line 1, the Perl client constructs a `SOAP::Lite` object (the reference is `$soap`) that communicates with the `RandService`. The `uri` value of `http://rand/` is the namespace that identifies a particular service available at the `proxy` (that is, the URL) value of `http://localhost:8888/rs`. A given service endpoint, a URL, could host any number of services, with a URI identifying each. In line 2, the call to `next1` returns a SOAP message:

```
<?xml version="1.0" ?>
<S:Envelope
    xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:next1Response xmlns:ns2="http://rand/">
      <return>1774649411</return>
    </ns2:next1Response>
  </S:Body>
</S:Envelope>
```

The cascaded call to `result` (also line 2) extracts the value `1774649411` from the SOAP envelope, and the value is assigned to the variable `$num`. The client program prints the value and exits. This Perl-to-Java request again confirms the language transparency of a SOAP-based service.

The Perl client is especially useful because of its trace capabilities. Example 4-9 is the HTTP request that the Perl client generates on a sample run; Example 4-10 is the HTTP response from the Java service. In the request, the body of the POST request contains a SOAP envelope, so named because of the local name `Envelope` in the XML tag's qualified name `soap:Envelope`.

Example 4-9. The HTTP request from the Perl client to the `RandService`

```
POST http://localhost:8888/rs HTTP/1.1
Accept: text/xml
Accept: multipart/*
Accept: application/soap
Content-Length: 420
Content-Type: text/xml; charset=utf-8
SOAPAction: ""

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

```
                soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <tns:next1 xsi:nil="true" /> ❶
   </soap:Body>
</soap:Envelope>
```

Example 4-10. The HTTP response from the `RandService` to the Perl client

```
HTTP/1.1 200 OK
Content-Type: text/xml;charset="utf-8"
Client-Peer: 127.0.0.1:8888
Client-Response-Num: 1
Client-Transfer-Encoding: chunked

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:next1Response xmlns:ns2="http://rand/">
         <return>1774649411</return>                    ❶
      </ns2:next1Response>
   </S:Body>
</S:Envelope>
```

Also in the request, this code in line 1 means that the *next1* operation takes no arguments:

```
<tns:next1 xsi:nil="true"/>
```

The HTTP response is more complicated than the request because there is a return value (line 1):

```
<return>1774649411</return>
```

The WSDL document specifies that the response from the `RandService` occurs in an element tagged `return`.

Why Does the Perl Client Not Invoke nextN as Well as next1?

The Perl client invokes the *next1* operation but not the parametrized *nextN* operation in the `RandService`. Were the Perl client to invoke *nextN*, the response from the `RandService` would be an empty list. When the published `RandService` receives a request, the Java runtime uses a SAX (Simple API for XML) parser to parse the incoming request; this parser belches on the SOAP request that the Perl library generates. In particular, the SAX parser fails to extract the *nextN* argument, which specifies how many randomly generated integers are to be returned. A glitch such as this is not uncommon in web services, including SOAP-based ones. Web services, REST-style and SOAP-based alike, are remarkably but not perfectly interoperable among programming languages.

The examples so far illustrate that the WSDL document can be used even if its detailed structure remains unknown. Now is the time to take a close look at how the WSDL is structured.

The WSDL Service Contract in Detail

The WSDL document, which is XML, is structured as follows:

- The document or root element is named `definitions`. This is appropriate because the WSDL defines the web service thoroughly enough that utilities such as *wsimport* can use the WSDL to generate code, typically but not exclusively client-side support code.
- The first child element of `definitions`, named `types`, is technically optional but almost always present in a modern WSDL. This element contains (or links to) an XML Schema or the equivalent—a grammar that specifies the data types for the messages involved in the service. In a modern SOAP-based web service, the arguments passed to web service operations are typed—but the SOAP messages themselves are also typed. For this reason, the receiver of a SOAP message can check, typically at the library level, whether the received message satisfies the constraints that the message's type impose.
- Next come one or more `message` elements, which list the messages whose data types are given in the `types` section immediately above. Every `message` has a corresponding `complexType` entry in the schema from the `types` section, assuming that the `types` section is nonempty.
- The `portType` section comes next. There is always exactly one `portType` element. The `portType` is essentially the service *interface*: a specification of the service's operations and the message patterns that the operations exemplify. For example, in the request/response pattern, the client begins the conversation with a request message and the service counters with a response message. In the solicit/response pattern, by contrast, the service starts the conversation with a solicitation message and the client counters with a response. There is also the one-way pattern (client to server only) and the notification pattern (server to client only). Richer conversational patterns can be built out of these simpler ones. The `message` items in the preceding section are the components of an operation, and the `portType` section defines an `operation` by placing `message` items in a specific order.
- Next come one or more `binding` sections, which provide implementation detail such as the transport used in the service (for instance, HTTP rather than SMTP), the service *style*, and the SOAP version (that is, 1.1 or 1.2). By default, Java generates a single `binding` section but DotNet generates two: one for SOAP 1.1 and another for SOAP 1.2.

- The last section, named `service`, brings all of the previous details together to define key attributes such as the *service endpoint*—that is, the URL at which the service can be accessed. Nested in the `service` element are one or more `port` subelements, where a `port` is a `portType` plus a `binding`:

  ```
  port = portType + binding
  ```

  Since there is only one `portType` in a WSDL, the number of `port` subelements equals the number of `binding` elements.

The biggest section in a WSDL is typically the `types` section because an XML Schema tends to be wordy. An example from Amazon, introduced shortly, illustrates. For now, the WSDL (see Example 4-11) for the `RandService` is only about a page or so in size.

Example 4-11. The dynamically generated WSDL for the `RandService`

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:tns="http://rand/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             targetNamespace="http://rand/" name="RandServiceService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://rand/"
                  schemaLocation="http://localhost:8888/rs?xsd=1"></xsd:import>
    </xsd:schema>
  </types>
  <message name="next1">
    <part name="parameters" element="tns:next1"></part>
  </message>
  <message name="next1Response">
    <part name="parameters" element="tns:next1Response"></part>
  </message>
  <message name="nextN">
    <part name="parameters" element="tns:nextN"></part>
  </message>
  <message name="nextNResponse">
    <part name="parameters" element="tns:nextNResponse"></part>
  </message>
  <portType name="RandService">
    <operation name="next1">
      <input message="tns:next1"></input>
      <output message="tns:next1Response"></output>
    </operation>
    <operation name="nextN">
      <input message="tns:nextN"></input>
      <output message="tns:nextNResponse"></output>
    </operation>
  </portType>
  <binding name="RandServicePortBinding" type="tns:RandService">
```

```
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                  style="document"></soap:binding>
    <operation name="next1">
      <soap:operation soapAction=""></soap:operation>
      <input>
        <soap:body use="literal"></soap:body>
      </input>
      <output>
        <soap:body use="literal"></soap:body>
      </output>
    </operation>
    <operation name="nextN">
      <soap:operation soapAction=""></soap:operation>
      <input>
        <soap:body use="literal"></soap:body>
      </input>
      <output>
        <soap:body use="literal"></soap:body>
      </output>
    </operation>
  </binding>
  <service name="RandServiceService">
    <port name="RandServicePort" binding="tns:RandServicePortBinding">
      <soap:address location="http://localhost:8888/rs"></soap:address>
    </port>
  </service>
</definitions>
```

The first three WSDL sections (`types`, `message`, and `portType`) present the service abstractly in that no implementation details are present. The `binding` and `service` sections provide the concrete detail by specifying, for example, the type of transport used in the service as well as the service endpoint.

The `portType` is of particular interest because it characterizes the service in terms of operations, not simply messages; operations consist of one or more messages exchanged in a specified pattern. The two areas of immediate interest in the WSDL for a programmer writing a client against a service would be the `portType` and the `service`; the `portType` section informs the programmer about what calls can be made against the service, and the `service` section gives the service endpoint, the URL through which the service can be reached.

XML is not fun to read, but the basic profile WSDL for the `RandService` is not unduly forbidding. Perhaps the best way to read the document is from top to bottom.

The types Section

This section contains or links to an XML Schema or equivalent. (In the case of Java, the schema is a separate document shown in Example 4-3; in the case of DotNet, the schema

is included in the WSDL.) To understand how the schema relates to its WSDL, consider this segment of the XML Schema from Example 4-3:

```
<xs:element name="nextNResponse" type="tns:nextNResponse">                ❶
</xs:element>
...
<xs:complexType name="nextNResponse">                                     ❷
  <xs:sequence>
    <xs:element name="return"                                             ❸
                type="xs:int" minOccurs="0" maxOccurs="unbounded">
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

The `xs:element` in line 1 has a specified `type`, in this case `tns:nextNResponse`. The type is the `complexType` in line 2. XML Schema has built-in simple types such as `xsd:int` and `xsd:string`, but XML Schema is also *extensible* in that new complex types can be added as needed. The `complexType` in this case is for the `nextNResponse` message that the service returns to the client. Here is that message from the WSDL in Example 4-11:

```
<message name="nextNResponse">
  <part name="parameters" element="tns:nextNResponse"></part>   ❶
</message>
```

The `message` has an `element` attribute (line 1) with `tns:nextNResponse` as the value; `tns:nextNResponse` is the name of the `element` in line 1 of the XML Schema. The WSDL, in defining a message, points back to the XML Schema section that provides the data type for the message.

The `complexType` section of the WSDL indicates that a `nextNResponse` message returns zero or more integers (XML type `xs:int`). The zero leaves open the possibility that the service, in this case written in Java, might return `null` instead of an actual array or equivalent (e.g., `List<Integer>`). At this point a human editor might intervene by changing the `minOccurs` in line 3 from `0` to `1`. (If the `minOccurs` attribute were dropped altogether, the value would default to 1.) The dynamically generated WSDL may not capture the intended design of a service; hence, the WSDL may need to be edited by hand.

The message Section

Each `message` element in the WSDL points to an element and, more important, to a `complexType` in the WSDL's XML Schema. The result is that all of the messages are typed. The `RandService` exposes two operations and each follows the request/response pattern; hence, the WSDL has four `message` elements: two for the `next1` and `nextN` requests and two for the corresponding responses named `next1Response` and `nextNResponse`, respectively.

The portType Section

This section contains one or more `operation` elements, each of which defines an operation in terms of messages defined in the immediately preceding section. For example, here is the definition for the *nextN* operation:

```
<operation name="nextN">
  <input message="tns:nextN"></input>
  <output message="tns:nextNResponse"></output>
</operation>
```

The `input` message precedes the `output` message, which signals that the pattern is request/response. Were the order reversed, the pattern would be solicit/response. The term *input* is to be understood from the service's perspective: an `input` message goes into the service and an `output` message comes out from the service. Each `input` and `output` element names the message defined in a `message` section, which in turn refers to an XML Schema `complexType`. Accordingly, each `operation` can be linked to the typed messages that make up the `operation`.

The binding Section

This section and the next, `service`, provide implementation details about the service. In theory, but rarely in practice, there are several options or *degrees of freedom* with respect to the service that the WSDL defines, and a `binding` section selects among these options. One option for a SOAP-based service such as the `RandService` is the SOAP version: 1.1 or 1.2. SOAP 1.1 is the default in Java; hence, the one and only `binding` section is for SOAP 1.1. In DotNet, a dynamically generated WSDL usually has two `binding` sections: one for SOAP 1.1 and the other for SOAP 1.2. However, the very same DotNet WSDL typically has only one service endpoint or URL; this means the same deployed service is for SOAP 1.1 and SOAP 1.2, thereby signaling that no difference between the two SOAP versions comes into play for the service.

There are three other options to be considered: transport (line 1) and style (line 2) are two of the three. Here is the first subelement in the `binding` section, a subelement that makes choices on these two options:

```
<soap:binding transport="http://schemas.xmlsoap.org/soap/http"  ❶
              style="document"></soap:binding>                  ❷
```

The `transport` value is a URI that ends with `soap/http`, which can be summed up as *SOAP over HTTP*. Another option would be SMTP (Simple Mail Transport Protocol) or even TCP (Transmission Control Protocol, which underlies HTTP), but in practice, HTTP is the dominant transport. HTTP in this context includes HTTPS. The other option (line 2) concerns the service `style`, in this case set to `document`. A web service in `document` style always has an XML Schema or equivalent that types the service's constituent messages. The other choice for `style` is misleadingly named `rpc`, which is

short for *remote procedure call*. The name is misleading because a `document`-style service such as the `RandService` can and typically does follow the request/response pattern, which is the RPC pattern. In the context of a WSDL, `rpc` style really means that messages themselves are not typed, only their arguments and return values are typed. The WSDL for an `rpc` style service may have no `types` section at all or only an abbreviated one. In modern SOAP-based services, `document` style dominates and represents best practice. Indeed, both Java and DotNet toyed for a time with the idea of dropping support altogether for `rpc` style. The issue of `rpc` style will come up again later but only briefly.

The `document` style deserves to be the default. This style can support services with rich, explicitly defined Java data types such as `Employee` or `ChessTournament` because the service's WSDL can define, for the XML side, the required types in an XML Schema. Any service pattern, including request/response, is possible under the document style.

The last option concerns `use`, more accurately called *encoding*, because the choice determines how the service's data types are to be encoded and decoded. The WSDL has to specify how the data types used in an implementation language such as Java are to be serialized into and deserialized out of WSDL-compliant types—the types laid out in the WSDL's XML Schema or equivalent (see Example 4-12). For example, Java and Ruby have similar but subtly different data types. In a conversation based on SOAP messages, a conversation in which the SOAP remains transparent, the two languages would need the ability to serialize from instances of native types to XML and to deserialize from XML to instances of native types.

Example 4-12. Encoding and decoding XML

```
           encode                    decode
Java types-------->XML Schema types-------->Ruby types

Java types<--------XML Schema types<--------Ruby types
           decode                    encode
```

The attribute

```
    use = 'literal'
```

means the service's type definitions in the WSDL *literally* follow the WSDL's schema. The alternative to `literal` is named `encoded`, which means that the service's type definitions come from implicit encoding rules, typically the rules in the SOAP 1.1 specification. However, the use of `encoded` does not comply with WS-I (Web Services Interoperability) standards.

The service Section

This section brings the pieces together. Recall that a WSDL has but one `portType` section but may have multiple `binding` sections. The `service` element has `port` subelements,

where a `port` is a `portType` linked to a `binding`; hence, the number of `port` subelements equals the number of `binding` sections in the WSDL. In this example, there is one `binding` and, therefore, one `port` subelement:

```
<port name="RandServicePort" binding="tns:RandServicePortBinding">
   <soap:address location="http://localhost:8888/rs"></soap:address> ❶
</port>
```

The `address` subelement specifies a `location` (line 1), whose value is commonly called the service endpoint. A web service with two significantly different bindings (for instance, one for HTTP and another for SMTP) would have different `location` values to reflect the different bindings.

Java and XML Schema Data Type Bindings

The foregoing examination of the WSDL, and in particular its XML Schema, prompts an obvious question: Which Java data types bind to which XML Schema data types? Table 4-1 summarizes the bindings.

Table 4-1. Java and XML Schema data type bindings

Java data type	XML schema data type
boolean	xsd:boolean
byte	xsd:byte
short	xsd:short
short	xsd:unsignedByte
int	xsd:int
int	xsd:unsignedShort
long	xsd:long
long	xsd:unsignedInt
float	xsd:float
double	xsd:double
byte[ ]	xsd:hexBinary
byte[ ]	xsd:base64Binary
java.math.BigInteger	xsd:integer
java.math.BigDecimal	xsd:decimal
java.lang.String	xsd:string
java.lang.String	xsd:anySimpleType
javax.xml.datatype.XMLGregorianCalendar	xsd:dateTime
javax.xml.datatype.XMLGregorianCalendar	xsd:time
javax.xml.datatype.XMLGregorianCalendar	xsd:date
javax.xml.datatype.XMLGregorianCalendar	xsd:g

Java data type	XML schema data type
javax.xml.datatype.Duration	xsd:duration
javax.xml.namespace.QName	xsd:QName
javax.xml.namespace.QName	xsd:NOTATION
java.lang.Object	xsd:anySimpleType

The bindings in Table 4-1 are automatic in the sense that, in a JAX-WS service, the SOAP infrastructure does the conversions without application intervention. Conversions also are automatic for arrays of any type in Table 4-1. For example, an array of `BigInteger` instances converts automatically to an array of `xsd:integer` instances, and vice versa. Programmer-defined classes whose properties reduce to any type in Table 4-1 or to arrays of these likewise convert automatically. For example, an `Employee` class that has properties such as `firstName` (`String`), `lastName` (`String`), `id` (`int`), `salary` (`float`), `age` (`short`), `hobbies` (`String[ ]`), and the like would convert automatically to XML Schema types. The upshot is that the vast majority of the data types used in everyday Java programming convert automatically to and from XML Schema types. The glaring exception is the `Map`—a collection of key/value pairs. However, a `Map` is readily implemented as a pair of coordinated arrays: one for the keys, the other for the values.

Code First or Contract First?

Should the web service code be used to generate the WSDL or should the WSDL, designed beforehand, be used to guide the coding of the web service? This question sums up the *code first versus contract first* controversy. The examples so far take the code-first approach: the service publisher (for example, `Endpoint` or Tomcat) automatically generates the WSDL for the service. The code-first approach has the obvious appeal of being easy. Yet the code-first approach has drawbacks, including:

- If the service changes under a code-first approach, the WSDL thereby changes—and client code generated from the WSDL (using, for instance, *wsimport*) needs to be regenerated. In this sense, the code-first approach is not client friendly. The code-first approach compromises a basic principle of software development: a service contract, once published, should be treated as immutable so that client-side code written against a published service never has to be rewritten.
- The code-first approach goes against the *language neutrality* at the core of web services. If a service contract is done first, the implementation language remains open.
- The code-first approach does not address tricky but common problems such as `null` arguments or return values. Consider, for example, a very simple service that includes an operation to return the current time as a string:

```
@WebMethod
public String getTime() { return new java.util.Date().toString(); }
```

Here is the relevant entry in the XML Schema from the automatically generated WSDL:

```
<xs:element name="return" type="xs:string" minOccurs="0"></xs:element>
```

The `minOccurs` value of `0` allows the *getTime* operation to return `null`. Suppose, however, the service needs to ensure that returned string has no fewer than, say, 28 characters, which rules out `null` as a return value. (In Java, a stringified `Date` has 28 characters.) The relevant schema section might look like this:

```
<xs:element minOccurs = "1"
            maxOccurs = "1"
            nillable  = "false"
            name      = "currentTime">
    <simpleType>
      <restriction base  = "string">
        <minLength value = "28"/>
      </restriction>
    </simpleType>
</xs:element>
```

By the way, the `minOccurs` and `maxOccurs` elements, each with a value of `1`, could be dropped altogether because `1` is the default value for these attributes. The point here is that a schema entry such as this must be handcrafted. Even a clever use of Java annotations is not sufficient to produce this entry automatically.

Given the ease of the code-first approach—not to mention the economic pressures and hectic pace of software development—there seems to be little chance that a contract-first approach to web services will eclipse the dominant code-first approach. Nonetheless, a code-first-generated WSDL and its accompanying schema can be refined as needed to ensure that this contract document reflects service requirements.

Wrapped and Unwrapped Document Style

The source for the `RandService` class begins as follows:

```
@WebService
public class RandService {
...
```

The default style, `document`, could be overridden with an additional annotation:

```
@WebService
@SOAPBinding(style = Style.RPC) // versus Style.DOCUMENT, the default
public class RandService {
...
```

The `RandService` is simple enough that the difference would be transparent to clients against the service. Of interest here is how the different styles impact the underlying SOAP messages.

Consider a very simple SOAP-based service with operations named `add`, `subtract`, `multiply`, and `divide`. Each operation expects two arguments, the numbers on which to operate. Under the original SOAP 1.1 specification, a request message for `document` style—what is now called *unwrapped* or *bare* document style—would look like Example 4-13:

Example 4-13. Unwrapped document style

```
<?xml version="1.0" ?>
<!-- Unwrapped document style -->
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soapenv:Body>
     <num1 xmlns:ans="http://arith/">27</num1>
     <num2 xmlns:ans="http://arith/">94</num2>
  </soapenv:Body>
</soapenv:Envelope>
```

The `Body` of the SOAP message contains two elements at the same level, the elements tagged `num1` and `num2`; each element is a child of the `soapenv:Body` element. The glaring omission is the name of the operation, for instance, `add`. This name might occur instead, for example, in the request URL:

```
http://some.server.org/add
```

It is peculiar that the SOAP envelope should contain the named arguments but not the named operation. Under `rpc` style, however, the operation would be the one and only child of the `Body` element; the operation then would have, as its own child elements, the arguments. Here is the contrasting SOAP message in *rpc* style or, what now comes to the same thing, *wrapped document style* (Example 4-14).

Example 4-14. Wrapped document style, the same as rpc style

```
<?xml version="1.0" ?>
<!-- Wrapped document or rpc style -->
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soapenv:Body>
     <add xmlns:ans="http://arith/">  ❶
       <num1>27</num1>
       <num2>94</num2>
     </addNums>
  </soapenv:Body>
</soapenv:Envelope>
```

The `add` element (line 1) now acts as a *wrapper* for the argument elements, in this case `num1` and `num2`. The wrapped convention, unofficial but dominant in SOAP frameworks, gives a `document`-style service the look and feel of an `rpc`-style service—at the message level. The document style still has the advantage of a full XML Schema that types the messages. In Java as in DotNet, the default style for any SOAP-based service is *wrapped document*; hence, a service such as `RandService`, with only the `@WebService` annotation, is wrapped document in style. This style is often shortened to *wrapped doc/lit*: wrapped document style with literal encoding.

Another Practical Use for the WSDL and Its XML Schema

A utility such as *wsimport* consumes a WSDL and produces Java classes that ease the task of writing a client against the service defined in the WSDL. However, the *wsimport* utility can also be used to generate service-side code (see the section, "wsimport Artifacts for the Service Side" on page 171). There are other practical uses for the WSDL. Consider a scenario (see Figure 4-2) in which incoming SOAP requests target operations encapsulated in a document-style SOAP-based service.

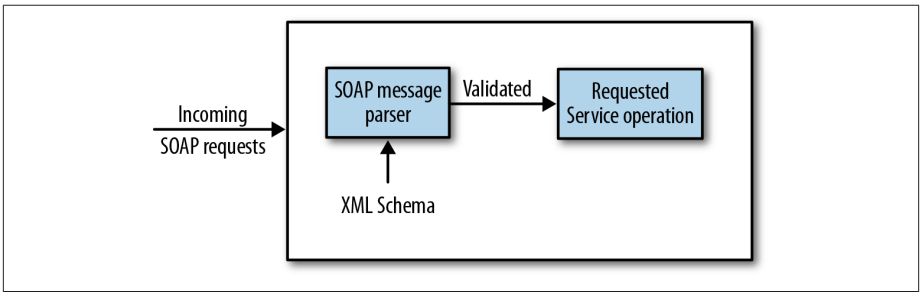

Figure 4-2. Using the WSDL's XML Schema for message validation

If the service is *doc/lit*, there is a WSDL with an attendant XML Schema; this schema can be used in a validating parse of the incoming SOAP messages. In XML, a *validating parse* checks whether a document, in this case a SOAP document, is syntactically correct but also whether the document conforms to a grammar, the XML Schema. If the parser does not validate the incoming message, there is no point in wasting CPU cycles on the execution of the service operation. Every document-style service has an XML Schema or equivalent as part of service's WSDL. Accordingly, this schema can be used to check whether SOAP messages satisfy the grammar that the schema represents.

wsimport Artifacts for the Service Side

The *wsimport* utility produces, from a WSDL document, code that directly supports client calls against a web service. This same code can be used, with a few adjustments, to program a service. This section illustrates with a simple example.

Here are two operations for a temperature conversion service written in C#:

```
[WebMethod]
public double c2f(double t) { return 32.0 + (t * 9.0 / 5.0); }
[WebMethod]
public double f2c(double t) { return (5.0 / 9.0) * (t - 32.0); }
```

The `c2f` operation converts from centigrade to fahrenheit and the `f2c` method converts from fahrenheit to centigrade.

DotNet, by default, generates a WSDL with SOAP 1.1 and SOAP 1.2 bindings. This temperature conversion service is simple enough that the two bindings have the same implementation. In general, however, the *wsimport* utility can handle multiple bindings with the *-extension* flag. Assuming that the WSDL for the service is in the file *tc.wsdl*, the command:

```
% wsimport -p tempConvert -keep -extension tc.wsdl
```

generates the usual artifacts: Java *.class* files that represent the `c2f` and `f2c` request messages and their corresponding responses, together with various support files. Of interest here is the `interface`—the Java file that represents the `portType` section of the WSDL. Here is the file, cleaned up for readability:

```
package tempConvert;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.bind.annotation.XmlSeeAlso;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;

@WebService(name = "ServiceSoap",
            targetNamespace = "http://tempConvertURI.org/")
@XmlSeeAlso({
    ObjectFactory.class
})
public interface ServiceSoap {
    @WebMethod(operationName = "c2f",
               action = "http://tempConvertURI.org/c2f")
    @WebResult(name = "c2fResult",
               targetNamespace = "http://tempConvertURI.org/")
    @RequestWrapper(localName = "c2f",
                    targetNamespace = "http://tempConvertURI.org/",
```

```
                    className = "tempConvert.C2F")
    @ResponseWrapper(localName = "c2fResponse",
                     targetNamespace = "http://tempConvertURI.org/",
                     className = "tempConvert.C2FResponse")
    public double c2F(
        @WebParam(name = "t",
                  targetNamespace = "http://tempConvertURI.org/")
        double t);
    @WebMethod(operationName = "f2c",
               action = "http://tempConvertURI.org/f2c")
    @WebResult(name = "f2cResult",
               targetNamespace = "http://tempConvertURI.org/")
    @RequestWrapper(localName = "f2c",
                    targetNamespace = "http://tempConvertURI.org/",
                    className = "tempConvert.F2C")
    @ResponseWrapper(localName = "f2cResponse",
                     targetNamespace = "http://tempConvertURI.org/",
                     className = "tempConvert.F2CResponse")
    public double f2C(
        @WebParam(name = "t",
                  targetNamespace = "http://tempConvertURI.org/")
        double t);
}
```

The `ServiceSoap` interface, like any interface, *declares* but does not *define* methods, which in this case represent service operations. If the semantics of these two operations `c2f` and `f2c` are understood, then converting this *wsimport* artifact to a web service is straightforward:

- Change the `interface` to a POJO `class`.

  ```
  ...
  public class ServiceSoap {
  ...
  ```

- Implement the *c2f* and *f2c* operations by defining the methods. Java and C# are sufficiently close that the two implementations would be indistinguishable. For example, here is the body of *c2f* in either language.

  ```
  public double c2f(double t) { return 32.0 + (t * 9.0 / 5.0); }
  ```

  Not every language is as close to Java as C#, of course. Whatever the original implementation of a service, the challenge is the same: to understand what a service operation is supposed to do so that the operation can be re-implemented in Java.

- Publish the service with, for example, `Endpoint` or a web server such as Tomcat or Jetty.

  Although the *wsimport* utility could be used to help write a SOAP-based service in Java, the main use of this utility is still in support of clients against a SOAP-based service. The point to underscore is that the WSDL is sufficiently rich in detail to

support useful code on either the service or the client side. The next section returns to the Amazon E-Commerce service to illustrate this very point.

SOAP-Based Clients Against Amazon's E-Commerce Service

Chapter 3 has two Java clients against the RESTful Amazon E-Commerce service. The first client parses the XML document from Amazon in order to extract the desired information, in this case the author of a specified Harry Potter book, J. K. Rowling. The second client uses JAX-B to deserialize the returned XML document into a Java object, whose *get*-methods are then used to extract the same information. This section introduces two more clients against the E-Commerce service; in this case the service is SOAP-based and, therefore, the clients are as well. The SOAP-based clients use a *handler*, Java code that has access to every outgoing and incoming SOAP message. In the case of Amazon, the handler's job is to inject into the SOAP request the authentication information that Amazon requires, in particular a digest based on the *secretKey* used in both of the RESTful clients of Chapter 2. A message digest generated with the *secretKey*, rather than the *secretKey* itself, is sent from the client to the Amazon service; hence, the *secretKey* itself does not travel over the wire. SOAP handlers are the focus of the next chapter; for now, a handler is used but not analyzed.

The SOAP-based clients against Amazon's E-Commerce service, like the other SOAP-based Java clients in this chapter, rely upon *wsimport*-generated classes as building blocks. There are some key points about the SOAP-based service and its clients:

- The WSDL and *wsimport*.

  The WSDL for the SOAP-based version of Amazon's E-Commerce service is available at:

  ```
  http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl
  ```

  This WSDL is more than 1,500 lines in size, with most of these lines in the XML Schema. The *wsimport* utility can be applied to this WSDL in the usual way:

  ```
  % wsimport -p amazon -keep \
    http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl\
    -b custom.xml
  ```

  The result is a directory/package named *amazon* filled with client-support classes generated from the WSDL. The part at the end, with *custom.xml*, is explained shortly.

- Client-side API styles.

  The Amazon SOAP-based service follows best design practices and is, therefore, *wrapped doc/lit*. However, *wsimport* can generate different *client* APIs from one and the same Amazon WSDL. This point deserves elaboration. Consider a very simple

operation in a SOAP-based service, which takes two arguments, text and a pattern, and returns the number of times that the pattern occurs in the text. For example, the text might be the novel *War and Peace* and the pattern might be the name of one of the heroines, *Natasha*. The operation is named *getCount*. There are different ways in which this operation might be implemented in Java. Perhaps the obvious implementation would have the declaration:

```
public int getCount(String text, String pattern);
```

This version takes two arguments, the text and the pattern, and returns the count as an `int`. Yet the client of a SOAP-based web service, following in the footsteps of DCE/RPC, can distinguish between *in* and *out* parameters—arguments passed *into* the service and ones passed *out of* this same service and back to the client. This possibility opens the way to a quite different version of *getCount*:

```
public void getCount(String text, String pattern, Holder result);
```

The return type for `getCount` is now `void`, which means that the *count* must be returned in some other way. The third parameter, of the special type `Holder`, embeds the desired count of `pattern` occurrences in the `text`. This programming style is uncommon in Java and, accordingly, might be judged inferior to the two-argument version of `getCount` that returns the *count* directly as an `int`. The point of interest is that *wsimport* can generate client-side artifacts in either style, and, perhaps surprisingly, the second style is the default for *wsimport*. In Java, the first style is:

```
SOAPBinding.ParameterStyle.BARE
```

and the second style is:

```
SOAPBinding.ParameterStyle.WRAPPED
```

The critical point is that these *parameter styles* refer to the *wsimport* artifacts generated from a service WSDL—the parameter styles do not refer to the structure of the service itself, which remains *wrapped doc/lit*. Java's *wsimport* utility can present this service style, on the client side, in different ways, known as parameter styles in Java.

- Authentication credentials in a SOAP-based client.

  A SOAP-based client against E-Commerce must send the same authentication credentials as a RESTful client: a registered user's *accessId* and a hash value generated with the *secretKey*. In a REST-style client, these credentials are sent in the query string of a GET request. A SOAP-based client is different in that its requests are all POSTs, even if the intended web service operation is a *read*. In a SOAP-based exchange over HTTP, the request is a SOAP envelope that is the body of a POST request. Accordingly, a SOAP-based client must process the required credentials in a different way. In this section, the credential processing is partly the job of a SOAP handler, which is examined carefully in the next chapter.

In Chapter 3, the clients against the RESTful E-Commerce service did *lookup* operations. For contrast, the SOAP-based client does a *search* against the Amazon E-Commerce service. The `AmazonClientBareStyle` (see Example 4-15) is the first and friendliest SOAP-based client.

Example 4-15. A SOAP-based Amazon client in bare parameter style

```
package amazon;

import amazon.AWSECommerceService;
import amazon.AWSECommerceServicePortType;
import amazon.ItemSearchRequest;
import amazon.ItemSearchResponse;
import amazon.ItemSearch;
import amazon.Items;
import amazon.Item;
import amazon.AwsHandlerResolver;
import java.util.List;

class AmazonClientBareStyle {
    public static void main(String[ ] args) {
        if (args.length < 2) {
            System.err.println("AmazonClientBareStyle <accessId> <secretKey>");
            return;
        }
        final String accessId = args[0];
        final String secretKey = args[1];

        AWSECommerceService service = new AWSECommerceService();                    ❶
        service.setHandlerResolver(new AwsHandlerResolver(secretKey));              ❷
        AWSECommerceServicePortType port = service.getAWSECommerceServicePort();    ❸
        ItemSearchRequest request = new ItemSearchRequest();                        ❹
        request.setSearchIndex("Books");
        request.setKeywords("Austen");
        ItemSearch itemSearch= new ItemSearch();                                    ❺
        itemSearch.setAWSAccessKeyId(accessId);
        itemSearch.setAssociateTag("kalin");
        itemSearch.getRequest().add(request);
        ItemSearchResponse response = port.itemSearch(itemSearch);                  ❻
        List<Items> itemsList = response.getItems();                                ❼
        int i = 1;
        for (Items next : itemsList)
           for (Item item : next.getItem())
               System.out.println(String.format("%2d: ", i++) +
                                  item.getItemAttributes().getTitle());
    }
}
```

The ZIP file with the sample code includes an executable JAR with the code from Example 4-15 and its dependencies. The JAR can be executed as follows:

```
% java -jar AmazonClientBare.jar <accessId> <secretKey>
```

The `AmazonClientBareStyle` highlights what SOAP-based services have to offer to their clients. The *wsimport*-generated classes include the `AWSECommerceService` with a no-argument constructor. This class represents, to the client, the E-Commerce service. The usual two-step occurs: in line 1 an `AWSECommerceService` instance is constructed and in line 3 the `getAWSECommerceServicePort` method is invoked. The object reference `port` can now be used, in line 6, to launch a search against the E-Commerce service, which results in an `ItemSearchResponse`. Line 2 in the setup hands over the user's *secretKey* to the client-side handler, which uses the *secretKey* to generate a hash value as a message authentication code, which Amazon can then verify on the service side.

The remaining code, from line 7 on, resembles the code in the second RESTful client against the E-Commerce service. Here is a quick review of the SOAP-based code:

```
List<Items> itemsList = response.getItems();                          ❶
int i = 1;
for (Items next : itemsList)
   for (Item item : next.getItem())                                   ❷
      System.out.println(String.format("%2d: ", i++) +
                         item.getItemAttributes().getTitle());        ❸
```

The `ItemSearchResponse` from Amazon encapsulates a list of `Items` (line 1), each of whose members is itself a list. The nested `for` loop iterates (line 2) over the individual `Item` instances, printing the title of each book found (line 3). By the way, the search returns the default number of items found, 10; it is possible to ask for all of the items found. On a sample run, the output was:

```
 1: Persuasion (Dover Thrift Editions)
 2: Pride and Prejudice (The Cambridge Edition of the Works of Jane Austen)
 3: Emma (Dover Thrift Editions)
 4: Northanger Abbey (Dover Thrift Editions)
 5: Mansfield Park
 6: Love and Friendship
 7: Jane Austen: The Complete Collection (With Active Table of Contents)
 8: Lady Susan
 9: Jane Austen Collection: 18 Works, Pride and Prejudice, Love and Friendship,
        Emma, Persuasion, Northanger Abbey, Mansfield Park, Lady Susan & more!
10: The Jane Austen Collection: 28 Classic Works
```

Now is the time to clarify the *custom.xml* file used in the *wsimport* command against the Amazon WSDL. The filename *custom.xml* is arbitrary and, for review, here is the *wsimport* command:

```
% wsimport -p amazon -keep \
  http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl \
  -b custom.xml
```

The file *custom.xml* is:

```
<jaxws:bindings
    wsdlLocation =
```

```
        "http://ecs.amazonaws.com/AWSECommerceService/AWSECommerceService.wsdl"
        xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
      <jaxws:enableWrapperStyle>false</jaxws:enableWrapperStyle>  ❶
    </jaxws:bindings>
```

The key element in the file sets the `enableWrapperStyle` for the parameters to `false` (line 1). The result is the bare parameter style evident in the `AmazonClientBareStyle` code. The alternative to this style is the default one, the client-side wrapped style. The `AmazonClientWrappedStyle` (see Example 4-16) is a SOAP-based Amazon client in the default style.

Example 4-16. A SOAP-based Amazon client in wrapped parameter style

```
package amazon2;

import amazon2.AWSECommerceService;
import amazon2.AWSECommerceServicePortType;
import amazon2.ItemSearchRequest;
import amazon2.ItemSearch;
import amazon2.Items;
import amazon2.Item;
import amazon2.OperationRequest;
import amazon2.SearchResultsMap;
import amazon2.AwsHandlerResolver;

import javax.xml.ws.Holder;
import java.util.List;
import java.util.ArrayList;

class AmazonClientWrappedStyle {
    public static void main(String[ ] args) {
      if (args.length < 2) {
        System.err.println("java AmazonClientWrappedStyle <accessId> <secretKey>");
        return;
      }
      final String accessId = args[0];
      final String secretKey = args[1];

      AWSECommerceService service = new AWSECommerceService();
      service.setHandlerResolver(new AwsHandlerResolver(secretKey));
      AWSECommerceServicePortType port = service.getAWSECommerceServicePort();
      ItemSearchRequest request = new ItemSearchRequest();
      request.setSearchIndex("Books");
      request.setKeywords("Austen");
      ItemSearch search = new ItemSearch();
      search.getRequest().add(request);
      search.setAWSAccessKeyId(accessId);
      search.setAssociateTag("kalin");
      Holder<OperationRequest> operationRequest = null;       ❶
      Holder<List<Items>> items = new Holder<List<Items>>(); ❷
      port.itemSearch(search.getMarketplaceDomain(),          ❸
                      search.getAWSAccessKeyId(),
```

```
                    search.getAssociateTag(),
                    search.getXMLEscaping(),
                    search.getValidate(),
                    search.getShared(),
                    search.getRequest(),
                    operationRequest,                    ❹
                    items);                              ❺
      Items retval = items.value.get(0);                 ❻
      int i = 1;
      List<Item> item_list = retval.getItem();           ❼
      for (Item item : item_list)
          System.out.println(String.format("%2d: ", i++) +
                             item.getItemAttributes().getTitle());
   }
}
```

The `AmazonClientWrappedStyle` code uses *wsimport*-generated classes created with the following command:

```
% wsimport -p amazon2 -keep \
  http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl
```

The WSDL is the same as in previous examples, but the style of the *wsimport*-classes changes from bare to wrapped, a change reflected in the `AmazonClientWrappedStyle` code. The change is evident at lines 1 and 2, which declare two object references of type `Holder`. As the name suggests, a `Holder` parameter is meant to hold some value returned from the E-Commerce service: the `operationRequest` holds metadata about the request, whereas `items` holds the book list that results from a successful search. This idiom is common in C or C++ but rare—and, therefore, clumsy—in Java. The `Holder` parameters are the last two (lines 4 and 5) of the nine parameters in the revised `itemSearch` (line 3). On a successful search, `items` refers to a `value` (line 6) from which a list of `Items` is extracted. This code, too, is awkward in Java. This list of `Items` has a `getItem` method (line 7), which yields a `List<Item>` from which the individual `Item` instances, each representing a Jane Austen book, can be extracted.

The `AmazonClientWrappedStyle` client is clearly the clumsier of the two clients against SOAP-based E-Commerce service, a service that has a single WSDL and whose response payloads to the two clients are identical in structure. The two clients differ markedly in their APIs, however. The bare style API would be familiar to most Java programmers, but the wrapped style, with its two `Holder` types, would seem a bit alien even to an experienced Java programmer. Nonetheless, the wrapped style remains the default in Java and in DotNet.

Asynchronous Clients Against SOAP-Based Services

All of the SOAP-based clients examined so far make *synchronous* or *blocking* calls against a web service. For example, consider these two lines from the bare style client against the E-Commerce service:

```
ItemSearchResponse response = port.itemSearch(itemSearch); ❶
List<Items> itemsList = response.getItems();               ❷
```

The call in line 1 to `itemSearch` *blocks* in the sense that line 2 does not execute until `itemSearch` returns a value, perhaps `null`. There are situations in which a client might need the invocation of `itemSearch` to return immediately so that other application logic could be performed in the meantime. In this case, a *nonblocking* or *asynchronous* call to `itemSearch` would be appropriate.

The `RandClientAsync` (see Example 4-17) is an asynchronous client against the `Rand Service` (see Example 4-1).

Example 4-17. A client that makes asynchronous requests against the `RandService`

```
import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;
import java.util.List;
import clientAsync.RandServiceService;
import clientAsync.RandService;
import clientAsync.NextNResponse;

public class RandClientAsync {
    public static void main(String[ ] args) {
        RandServiceService service = new RandServiceService();
        RandService port = service.getRandServicePort();
        port.nextNAsync(4, new MyHandler());                          ❶
        try {
            Thread.sleep(5000); // in production, do something useful!
        }
        catch(Exception e) { }
        System.out.println("\nmain is exiting...");
    }
    static class MyHandler implements AsyncHandler<NextNResponse> {   ❷
        public void handleResponse(Response<NextNResponse> future) {  ❸
            try {
                NextNResponse response = future.get();                ❹
                List<Integer> nums = response.getReturn();            ❺
                for (Integer num : nums) System.out.println(num);     ❻
            }
            catch(Exception e) { System.err.println(e); }
        }
    }
}
```

Although an asynchronous client also could be coded against the E-Commerce service, the far simpler `RandService` makes the client itself relatively straightforward; it is then easier to focus on the asynchronous part of the API. No changes are required in the `RandService` or its publication, under either `Endpoint` or a web server such as Tomcat. The *wsimport* command again takes a customization file, in this example *customAsync.xml*; the filename is arbitrary. The *wsimport* command is:

```
wsimport -p clientAsync -keep http://localhost:8888/rs?wsdl -b customAsync.xml
```

The customized binding file is:

```
<jaxws:bindings
    wsdlLocation="http://localhost:8888/rs?wsdl"
    xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
  <jaxws:enableAsyncMapping>true</jaxws:enableAsyncMapping> ❶
</jaxws:bindings>
```

The customized binding sets the `enableAsyncMapping` to `true` (line 1). The *wsimport* utility generates the same classes as in the earlier examples: `Next1`, `Next1Response`, and so on. The request/response classes such as `Next1` and `Next1Response` have additional methods, however, to handle the asynchronous calls, and these classes still have the methods that make synchronous calls.

The setup in the asynchronous client is the familiar two-step: first create a service instance and then invoke the `getRandService` method on this instance. The dramatic change is line 1, the asynchronous call, which now takes two arguments:

```
port.nextNAsync(4, new MyHandler());
```

Although the `nextNAsync` method does return a value, my code does not bother to assign this value to a variable. The reason is that the Java runtime passes the `NextNResponse` message from the `RandService` to the client's event handler, an instance of `MyHandler`, which then extracts and prints the randomly generated integers from the service.

The call to `nextNAsync`, a method declared together with `nextN` in the *wsimport*-generated `RandService` interface, takes two arguments: the number of requested random numbers and an event handler, in this case a `MyHandler` instance. The handler class `MyHandler` must implement the `AsyncHandler` interface (line 2) by defining the `handleResponse` method (line 3). The `handleResponse` method follows the standard Java pattern for event handlers: the method has `void` as its return type and it expects one argument, an event triggered by a `Response<NextNResponse>` that arrives at the client.

When the client runs, the `main` thread executes the asynchronous call to `nextNAsync`, which returns immediately. To prevent the `main` thread from exiting `main` and thereby ending the application, the client invokes `Thread.sleep`. This is contrived, of course; in a production environment, the `main` thread presumably would go on to do meaningful work. In this example, the point is to illustrate the execution pattern. When the

RandService returns the requested integers, the Java runtime starts a (daemon) thread to execute the handleResponse callback, which prints the requested integers. In the meantime, the main thread eventually wakes up and exits main, thereby terminating the client's execution. On a sample run, the output was:

```
1616290443
-984786015
1002134912
311238217
main is exiting...
```

The daemon thread executing handleResponse prints the four integers, and the main thread prints the good-bye message.

Java and DotNet take different approaches toward generating, from a WSDL, support for asynchronous calls against a service. DotNet automatically generates methods for synchronous and asynchronous calls against the service; Java takes the more conservative approach of generating the asynchronous artifacts only if asked to do so with a customized binding such as the one used in this example. The key point is that Java API, like its DotNet counterpart, fully supports synchronous and asynchronous calls against SOAP-based services such as the RandService.

How Are WSDL and UDDI Related?

WSDL documents, as service contracts, should be publishable and discoverable, as are the services that they describe. A UDDI (Universal Description Discovery and Integration) registry is one way to publish a WSDL so that potential clients can discover the document and ultimately consume the web service that the WSDL describes. UDDI has a type system that accommodates WSDL documents as well as other kinds of formal service contracts. From a UDDI perspective, a WSDL appears as a two-part document. One part, which comprises the types through the binding sections, is the UDDI *service interface*. The other part, which comprises any import directives and the service section, is the UDDI *service implementation*. In WSDL, the service interface (portType section) and service implementation (binding sections) are two parts of the same document. In UDDI, they are two separate documents, and these UDDI terms do not match up exactly with their WSDL counterparts.

A WSDL does not explain service semantics or, in plainer terms, what the service is about. The WSDL does explain, in a clear and precise way, the service's invocation syntax: the names of the service operations (e.g., an operation such as *getTime*); the operation pattern (e.g., request/response rather than solicit/response); the number, order, and type of arguments that each operation expects; faults, if any, associated with a service operation; and the number, order, and types of response values from an operation. The W3C is pursuing initiatives in web semantics under the rubric of WSDL-S (Semantics).

As of now, however, a WSDL is useful only if a programmer already understands what the service is about. The WSDL can guide you through technical aspects of a web service, but this document presupposes rather than provides an insight into service semantics.

What's Next?

JAX-WS has two distinct but related APIs for SOAP-based web services. One API, with annotations such as `@WebService` and `@WebMethod`, focuses on what might be called the application level. On the service side, annotations are used to create web services and to specify their operations. Additional annotations such as `@WebParam` are available for fine-tuning and documenting different aspects of a service and its operations. On the client side, the application API enables clients to draw upon *wsimport*-generated classes to access a SOAP-based service and to invoke its operations. A central feature of the application level is the WSDL contract, which captures in XML the service and its operations, including essential details such as the invocation syntax for the service operations, the encoding/decoding scheme for data types, the transport used for messages, and the service endpoint. Frameworks such as JAX-WS and DotNet come with utilities that put the WSDL to practical use.

A major appeal of the application level in JAX-WS is that the SOAP itself—the XML—remains hidden on the service and the client sides. The underlying SOAP libraries serialize from Java into XML and deserialize from XML into Java, thereby allowing both service and service client to work in Java data structures and in familiar Java programming idioms. JAX-B and related utilities allow REST-style services in Java to work around the XML or JSON payloads typical of REST-style services, but JAX-WS, which uses JAX-B under the hood, takes the further step of automating the serialization/deserialization. JAX-WS is programmer-friendly on both the service and client side.

The examples in this chapter have remained, for the most part, at the JAX-WS application level but have touched on another level, the handler level. A second JAX-WS API, with the `Handler` interface and its two subinterfaces `LogicalHandler` and `SOAPHandler`, provides access to the underlying SOAP. A `SOAPHandler` gives the programmer access to the entire SOAP message, whereas the convenient `LogicalHandler` gives the programmer access to the payload of the SOAP body. A `SOAPHandler` often is called a *message handler* and a `LogicalHandler` is called simply a *logical handler*.

The handler API allows the programmer to inspect and, if needed, to manipulate the SOAP that the underlying libraries generate. The distinction between the JAX-WS application and handler APIs in web services corresponds roughly to the distinction between the *servlet* and *filter* API in Java-based websites. (One important difference is that JAX-WS handlers are available on both the client and the service side.) This chapter introduced but did not explore the handler API in the clients against Amazon's E-Commerce service. In the REST-style clients of Chapter 3 against the E-Commerce

service, the clients made GET requests, and critical pieces of information (e.g., the user's *accessId* and an HMAC hash generated from the user's *secretKey*) had to be sent in the query string. The very same pieces of information are required in a SOAP-based client, but such a client, even when invoking a *read* operation on the E-Commerce service, sends a POST request whose body is a SOAP envelope. The JAX-WS libraries generate the SOAP envelope, but a client-side handler, an instance of the mentioned but not listed `AwsServiceHandler` class, inserts the user's *accessId*, the HMAC hash, and a strictly formatted timestamp into the SOAP envelope, in particular into the SOAP body. The next chapter takes a close look at the `AwsServiceHandler` class.

The next chapter goes down to the JAX-WS handler level by explaining, first, how the class `AwsServiceHandler` works in the E-Commerce service clients. This chapter then does a full example with a client-side and a service-side handler, an example that deliberately mimics the way that Amazon's more complicated E-Commerce service works. JAX-WS also exposes to the programmer the *transport level*, which is almost always HTTP(S). On either the client side or the service side, JAW-WS code can inspect and, if appropriate, modify the HTTP messages that carry SOAP messages as their payloads. JAX-WS thus covers three distinct levels of SOAP-based web services: the application level, the handler level, and the transport level. Chapter 5 also looks at some miscellaneous but related topics: SOAP faults in the application and handler levels, binary payloads in SOAP services, and the Axis2 implementation of JAX-WS, which is an alternative to the Metro implementation.