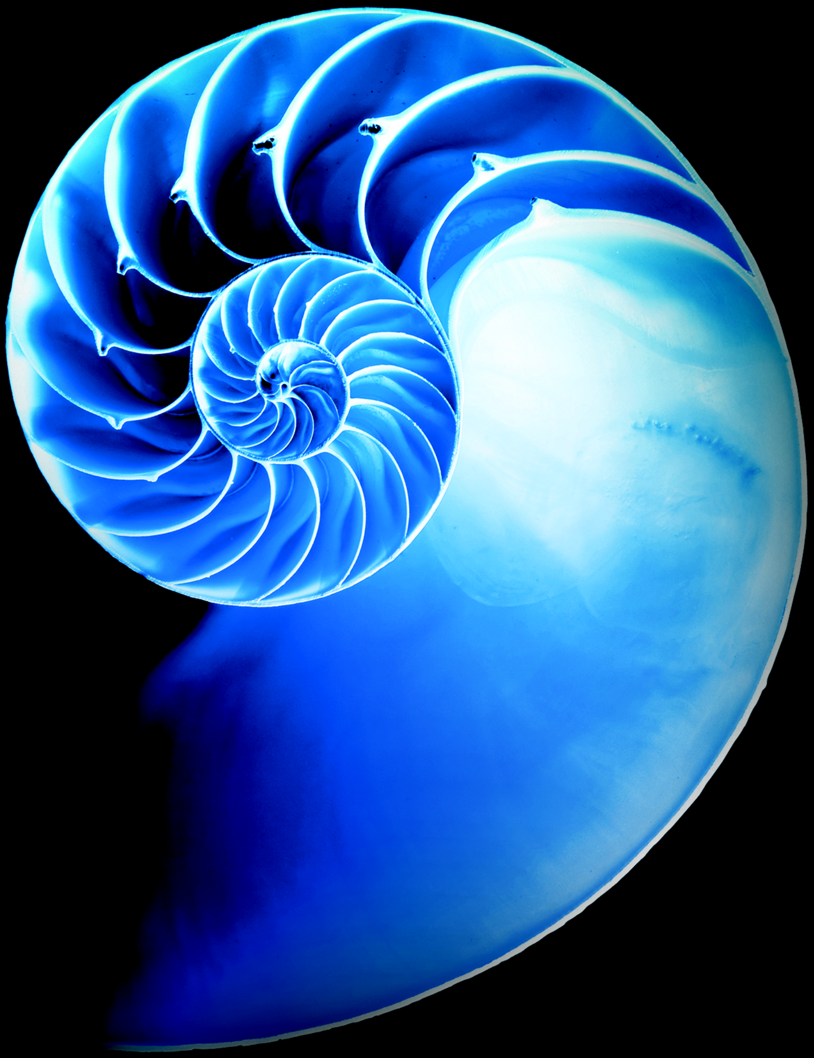
32 REST Web Services



Objectives

In this chapter you will learn:

- What a web service is.
- How to publish and consume web services in NetBeans.
- How XML, JSON, XML-Based Simple Object Access Protocol (SOAP) and Representational State Transfer (REST) Architecture enable Java web services.
- How to create client desktop and web applications that consume web services.
- How to use session tracking in web services to maintain client state information.
- How to connect to databases from web services.
- How to pass objects of user-defined types to and return them from a web service.

Outline

32.1 Introduction
32.2 Web Service Basics
32.3 Simple Object Access Protocol (SOAP)
32.4 Representational State Transfer (REST)
32.5 JavaScript Object Notation (JSON)
32.6 Publishing and Consuming SOAP-Based Web Services
32.6.1 Creating a Web Application Project and Adding a Web Service Class in NetBeans
31.6.2 Defining the `WelcomeSOAP` Web Service in NetBeans
31.6.3 Publishing the `WelcomeSOAP` Web Service from NetBeans
31.6.4 Testing the `WelcomeSOAP` Web Service with GlassFish Application Server's `Tester` Web Page
32.6.5 Describing a Web Service with the Web Service Description Language (WSDL)
31.6.6 Creating a Client to Consume the `WelcomeSOAP` Web Service
31.6.7 Consuming the `WelcomeSOAP` Web Service
32.7 Publishing and Consuming REST-Based XML Web Services
32.7.1 Creating a REST-Based XML Web Service
32.7.2 Consuming a REST-Based XML Web Service
32.8 Publishing and Consuming REST-Based JSON Web Services
32.8.1 Creating a REST-Based JSON Web Service
32.8.2 Consuming a REST-Based JSON Web Service
32.9 Session Tracking in a SOAP Web Service
31.9.1 Creating a `Blackjack` Web Service
31.9.2 Consuming the `Blackjack` Web Service
32.10 Consuming a Database-Driven SOAP Web Service
31.10.1 Creating the `Reservation` Database
31.10.2 Creating a Web Application to Interact with the `Reservation` Service
32.11 Equation Generator: Returning User-Defined Types
31.11.1 Creating the `Equation-GeneratorXML` Web Service
31.11.2 Consuming the `Equation-GeneratorXML` Web Service
31.11.3 Creating the `Equation-GeneratorJSON` Web Service
31.11.4 Consuming the `Equation-GeneratorJSON` Web Service
32.12 Wrap-Up

Summary | *Self-Review Exercises* | *Answers to Self-Review Exercises* | *Exercises* | *Making a Difference*

32.1 Introduction

This chapter introduces web services, which promote software portability and reusability in applications that operate over the Internet. A **web service** is a software component stored on one computer that can be accessed by an application (or other software component) on another computer over a network. Web services communicate using such technologies as XML, JSON and HTTP. In this chapter, we use two Java APIs that facilitate web services. The first, **JAX-WS**, is based on the **Simple Object Access Protocol (SOAP)**—an XML-based protocol that allows web services and clients to communicate, even if the client and the web service are written in different languages. The second, **JAX-RS**, uses **Representational State Transfer (REST)**—a network architecture that uses the web's traditional request/response mechanisms such as `GET` and `POST` requests. For more information on SOAP-based and REST-based web services, visit our Web Services Resource Centers:

```
www.deitel.com/WebServices/
www.deitel.com/RESTWebServices/
```

These Resource Centers include information about designing and implementing web services in many languages and about web services offered by companies such as Google, Amazon and eBay. You'll also find many additional tools for publishing and consuming web services. For more information about REST-based Java web services, check out the Jersey project:

```
jersey.java.net/
```

The XML used in this chapter is created and manipulated for you by the APIs, so you need not know the details of XML to use it here. To learn more about XML, read the following tutorials:

```
www.deitel.com/articles/xml_tutorials/20060401/XMLBasics/
www.deitel.com/articles/xml_tutorials/20060401/XMLStructuringData/
```

and visit our XML Resource Center:

```
www.deitel.com/XML/
```

Business-to-Business Transactions

Rather than relying on proprietary applications, businesses can conduct transactions via standardized, widely available web services. This has important implications for **business-to-business** (**B2B**) **transactions**. Web services are platform and language independent, enabling companies to collaborate without worrying about the compatibility of their hardware, software and communications technologies. Companies such as Amazon, Google, eBay, PayPal and many others make their server-side applications available to partners via web services.

By purchasing some web services and using other free ones that are relevant to their businesses, companies can spend less time developing applications and can create new ones that are more innovative. E-businesses for example, can provide their customers with enhanced shopping experiences. Consider an online music store. The store's website links to information about various artists, enabling users to purchase their music, to learn about the artists, to find more titles by those artists, to find other artists' music they may enjoy, and more. The store's website may also link to the site of a company that sells concert tickets and provides a web service that displays upcoming concert dates for various artists, allowing users to buy tickets. By consuming the concert-ticket web service on its site, the online music store can provide an additional service to its customers, increase its site traffic and perhaps earn a commission on concert-ticket sales. The company that sells concert tickets also benefits from the business relationship by selling more tickets and possibly by receiving revenue from the online music store for the use of the web service.

Any Java programmer with a knowledge of web services can write applications that "consume" web services. The resulting applications would invoke web services running on servers that could be thousands of miles away.

NetBeans

NetBeans is one of many tools that enable you to *publish* and/or *consume* web services. We demonstrate how to use NetBeans to implement web services using the JAX-WS and JAX-RS APIs and how to invoke them from client applications. For each example, we provide the web service's code, then present a client application that uses the web service. Our first examples build simple web services and client applications in NetBeans. Then we demonstrate web services that use more sophisticated features, such as manipulating databases with JDBC and manipulating class objects. For information on downloading and installing the NetBeans and the GlassFish server, see Section 30.1.

32.2 Web Service Basics

The machine on which a web service resides is referred to as a **web service host**. The client application sends a request over a network to the web service host, which processes the request and returns a response over the network to the application. This kind of distributed computing benefits systems in various ways. For example, an application without direct access to data on another system might be able to retrieve the data via a web service. Similarly, an application lacking the processing power to perform specific computations could use a web service to take advantage of another system's superior resources.

In Java, a web service is implemented as a class that resides on a server—it's not part of the client application. Making a web service available to receive client requests is known as **publishing a web service**; using a web service from a client application is known as **consuming a web service**.

32.3 Simple Object Access Protocol (SOAP)

The Simple Object Access Protocol (SOAP) is a platform-independent protocol that uses XML to interact with web services, typically over HTTP. You can view the SOAP specification at `www.w3.org/TR/soap/`. Each request and response is packaged in a **SOAP message**—XML markup containing the information that a web service requires to process the message. SOAP messages are written in XML so that they're computer readable, human readable and platform independent. Most **firewalls**—security barriers that restrict communication among networks—allow HTTP traffic to pass through, so that clients can browse the web by sending requests to and receiving responses from web servers. Thus, SOAP-based services can send and receive SOAP messages over HTTP connections with few limitations.

SOAP supports an extensive set of types, including the primitive types (e.g., `int`), as well as `DateTime`, `XmlNode` and others. SOAP can also transmit arrays of these types. When a program invokes a method of a SOAP web service, the request and all relevant information are packaged in a SOAP message enclosed in a **SOAP envelope** and sent to the server on which the web service resides. When the web service receives this SOAP message, it parses the XML representing the message, then processes the message's contents. The message specifies the *method* that the client wishes to execute and the *arguments* the client passed to that method. Next, the web service calls the method with the specified arguments (if any) and sends the response back to the client in another SOAP message. The client parses the response to retrieve the method's result. In Section 32.6, you'll build and consume a basic SOAP web service.

32.4 Representational State Transfer (REST)

Representational State Transfer (REST) refers to an architectural style for implementing web services. Such web services are often called **RESTful web services**. Though REST itself is not a standard, RESTful web services are implemented using web standards. Each method in a RESTful web service is identified by a unique URL. Thus, when the server receives a request, it immediately knows what operation to perform. Such web services can be used in a program or directly from a web browser. The results of a particular operation may be cached locally by the browser when the service is invoked with a `GET` request. This can make subsequent requests for the same operation faster by loading the result directly from the browser's cache. Amazon's web services (`aws.amazon.com`) are RESTful, as are many others.

RESTful web services are alternatives to those implemented with SOAP. Unlike SOAP-based web services, the request and response of REST services are not wrapped in envelopes. REST is also not limited to returning data in XML format. It can use a variety of formats, such as XML, JSON, HTML, plain text and media files. In Sections 32.7–32.8, you'll build and consume basic RESTful web services.

32.5 JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON) is an alternative to XML for representing data. JSON is a text-based data-interchange format used to represent objects in JavaScript as collections of name/value pairs represented as `String`s. It's commonly used in Ajax applications. JSON is a simple format that makes objects easy to read, create and parse and, because it's much less verbose than XML, allows programs to transmit data efficiently across the Internet. Each JSON object is represented as a list of property names and values contained in curly braces, in the following format:

```
{ propertyName1 : value1, propertyName2 : value2 }
```

Arrays are represented in JSON with square brackets in the following format:

```
[value1, value2, value3]
```

Each value in an array can be a string, a number, a JSON object, `true`, `false` or `null`. To appreciate the simplicity of JSON data, examine this representation of an array of address-book entries:

```
[{first: 'Cheryl', last: 'Black'},
 {first: 'James', last: 'Blue'},
 {first: 'Mike', last: 'Brown'},
 {first: 'Meg', last: 'Gold'}]
```

Many programming languages now support the JSON data format. An extensive list of JSON libraries sorted by language can be found at `www.json.org`.

32.6 Publishing and Consuming SOAP-Based Web Services

This section presents our first example of publishing (enabling for client access) and consuming (using) a web service. We begin with a SOAP-based web service.

32.6.1 Creating a Web Application Project and Adding a Web Service Class in NetBeans

When you create a web service in NetBeans, you focus on its logic and let the IDE and server handle its infrastructure. First you create a **Web Application project**. NetBeans uses this project type for web services that are invoked by other applications.

Creating a Web Application Project in NetBeans

To create a web application, perform the following steps:

1. Select **File > New Project...** to open the **New Project** dialog.
2. Select **Java Web** from the dialog's **Categories** list, then select **Web Application** from the **Projects** list. Click **Next >**.
3. Specify `WelcomeSOAP` in the **Project Name** field and specify where you'd like to store the project in the **Project Location** field. You can click the **Browse** button to select the location. Click **Next >**.
4. Select **GlassFish Server 4.1** from the **Server** drop-down list and **Java EE 7 Web** from the **Java EE Version** drop-down list.
5. Click **Finish** to create the project.

This creates a web application that will run in a web browser, similar to the projects used in Chapters 30 and 31.

Adding a Web Service Class to a Web Application Project

Perform the following steps to add a web service class to the project:

1. In the **Projects** tab in NetBeans, right click the **WelcomeSOAP** project's node and select **New > Web Service...** to open the **New Web Service** dialog.
2. Specify `WelcomeSOAP` in the **Web Service Name** field.
3. Specify `com.deitel.welcomesoap` in the **Package** field.
4. Click **Finish** to create the web service class.

The IDE generates a sample web service class with the name from *Step 2* in the package from *Step 3*. You can find this class in your project's **Web Services** node. In this class, you'll define the methods that your web service makes available to client applications. When you eventually build your application, the IDE will generate other supporting files for your web service.

32.6.2 Defining the `WelcomeSOAP` Web Service in NetBeans

Figure 32.1 contains the completed `WelcomeSOAPService` code (reformatted to match the coding conventions we use in this book). First we discuss this code, then show how to use the NetBeans web service design view to add the `welcome` method to the class.

```java
// Fig. 32.1: WelcomeSOAP.java
// Web service that returns a welcome message via SOAP.
package com.deitel.welcomesoap;

import javax.jws.WebService; // program uses the annotation @WebService
import javax.jws.WebMethod; // program uses the annotation @WebMethod
import javax.jws.WebParam; // program uses the annotation @WebParam

```

Fig. 32.1 | Web service that returns a welcome message via SOAP. (Part 1 of 2.)

```
 9   @WebService() // annotates the class as a web service
10   public class WelcomeSOAP
11   {
12      // WebMethod that returns welcome message
13      @WebMethod(operationName = "welcome")
14      public String welcome(@WebParam(name = "name") String name)
15      {
16         return "Welcome to JAX-WS web services with SOAP, " + name + "!";
17      }
18   }
```

Fig. 32.1 | Web service that returns a welcome message via SOAP. (Part 2 of 2.)

Annotation `import` *Declarations*

Lines 5–7 import the annotations used in this example. By default, each new web service class created with the JAX-WS APIs is a POJO (plain old Java object), so you do *not* need to extend a class or implement an interface to create a web service.

`@WebService` *Annotation*

Line 9 contains a **`@WebService` annotation** (imported at line 5) which indicates that class `WelcomeSOAP` implements a web service. The annotation is followed by parentheses that may contain optional annotation attributes. The optional **`name` attribute** specifies the name of the service endpoint interface class that will be generated for the client. A **service endpoint interface (SEI)** class (sometimes called a **proxy class**) is used to interact with the web service—a client application consumes the web service by invoking methods on the service endpoint interface object. The optional **`serviceName` attribute** specifies the service name, which is also the name of the class that the client uses to obtain a service endpoint interface object. If the `serviceName` attribute is not specified, the web service's name is assumed to be the Java class name followed by the word `Service`. NetBeans places the `@WebService` annotation at the beginning of each new web service class you create. You can then add the `name` and `serviceName` properties in the parentheses following the annotation.

When you deploy a web application containing a class that uses the `@WebService` annotation, the server (GlassFish in our case) recognizes that the class implements a web service and creates all the **server-side artifacts** that support the web service—that is, the framework that allows the web service to wait for client requests and respond to those requests once it's deployed on an application server. Some popular open-source application servers that support Java web services include GlassFish (`glassfish.dev.java.net`), Apache Tomcat (`tomcat.apache.org`) and JBoss Application Server (`www.jboss.com/products/platforms/application`).

`WelcomeSOAP` *Service's* `welcome` *Method*

The `WelcomeSOAP` service has only one method, `welcome` (lines 13–17), which takes the user's name as a `String` and returns a `String` containing a welcome message. This method is tagged with the **`@WebMethod` annotation** to indicate that it can be called remotely. Any methods that are not tagged with `@WebMethod` are *not* accessible to clients that consume the web service. Such methods are typically utility methods within the web service class. The `@WebMethod` annotation uses the **`operationName`** attribute to specify the method name that is exposed to the web service's client. If the `operationName` is not specified, it's set to the actual Java method's name.

Common Programming Error 32.1

Failing to expose a method as a web method by declaring it with the `@WebMethod` annotation prevents clients of the web service from accessing the method. There's one exception—if none of the class's methods are declared with the `@WebMethod` annotation, then all the `public` methods of the class will be exposed as web methods.

Common Programming Error 32.2

Methods with the `@WebMethod` annotation cannot be `static`. An object of the web service class must exist for a client to access the service's web methods.

The `name` parameter to `welcome` is annotated with the **`@WebParam` annotation** (line 14). The optional `@WebParam` attribute **`name`** indicates the parameter name that is exposed to the web service's clients. If you don't specify the name, the actual parameter name is used.

Completing the Web Service's Code

[*Note:* If you enter the code in Fig. 32.1 manually, then you can skip the following steps.] NetBeans provides a web service design view in which you can define the method(s) and parameter(s) for your web services. To define the `WelcomeSOAP` class's `welcome` method, perform the following steps:

1. With `WelcomeSOAP.java` open in the editor, click the **Design** button at the top of the editor to show the design view (Fig. 32.2).

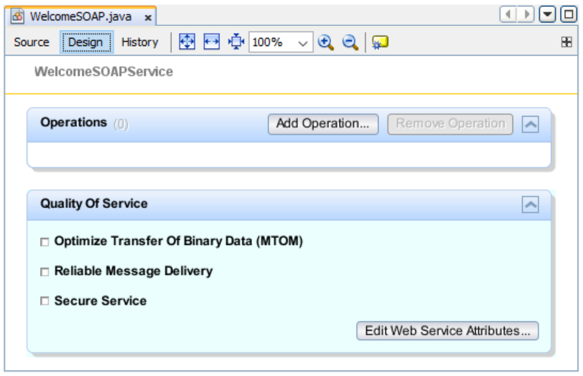
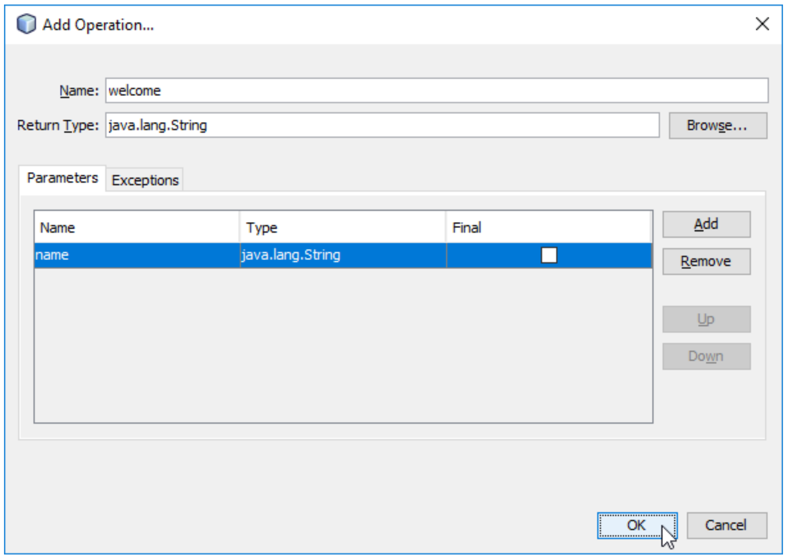


Fig. 32.2 | Web service design view.

2. Click the **Add Operation...** button to display the **Add Operation...** dialog (Fig. 32.3).
3. Specify the method name `welcome` in the **Name** field. The default **Return Type** (`String`) is correct for this example.



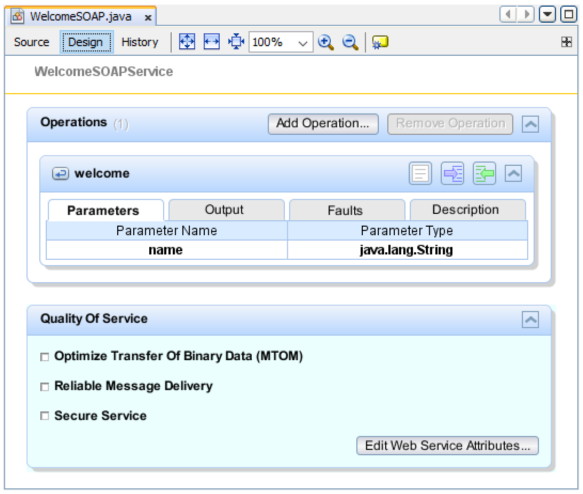
Name: welcome

Return Type: java.lang.String

Name	Type	Final
name	java.lang.String	☐

Fig. 32.3 | Adding an operation to a web service.

4. Add the method's `name` parameter by clicking the **Add** button to the right of the **Parameters** tab then entering `name` in the **Name** field. The parameter's default **Type** (`String`) is correct for this example.
5. Click **OK** to create the `welcome` method. The design view should now appear as shown in Fig. 32.3.



Parameter Name	Parameter Type
name	java.lang.String

☐ Optimize Transfer Of Binary Data (MTOM)

☐ Reliable Message Delivery

☐ Secure Service

Fig. 32.4 | Web service design view after new operation is added.

6. At the top of the design view, click the **Source** button to display the class's source code and add the code line 18 of Fig. 32.1 to the body of method `welcome`.

32.6.3 Publishing the `WelcomeSOAP` Web Service from NetBeans

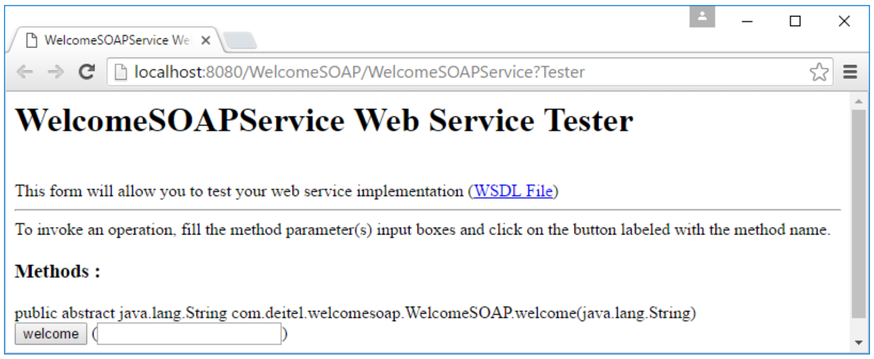
Now that you've created the `WelcomeSOAP` web service class, you'll use NetBeans to build and *publish* (that is, deploy) the web service so that clients can consume its services. NetBeans handles all the details of building and deploying a web service for you. This includes creating the framework required to support the web service. Right click the project name `WelcomeSOAP` in the **Projects** tab and select **Deploy** to build and deploy the web application to the GlassFish server. If GlassFish is not already running, NetBeans will start it.

32.6.4 Testing the `WelcomeSOAP` Web Service with GlassFish Application Server's `Tester` Web Page

Next, you'll test the `WelcomeSOAP` web service. We previously selected the GlassFish application server to execute this web application. This server can dynamically create a web page that allows you to test a web service's methods from a web browser. To use this capability:

1. Expand the project's **Web Services** in the NetBeans **Projects** tab.
2. Right click the web service class name (`WelcomeSOAP`) and select **Test Web Service**.

The GlassFish application server builds the `Tester` web page and loads it into your web browser. Figure 32.5 shows the `Tester` web page for the `WelcomeSOAP` web service. The web service's name is automatically the class name followed by `Service`.



WelcomeSOAPService Web Service Tester

This form will allow you to test your web service implementation (WSDL File)

To invoke an operation, fill the method parameter(s) input boxes and click on the button labeled with the method name.

Methods :

public abstract java.lang.String com.deitel.welcomesoap.WelcomeSOAP.welcome(java.lang.String)

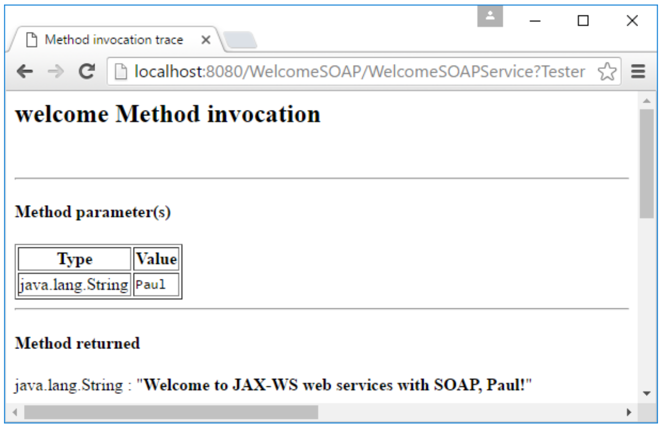
Fig. 32.5 | `Tester` web page created by GlassFish for the `WelcomeSOAP` web service.

Once you've deployed the web service, you can also type the URL

```
http://localhost:8080/WelcomeSOAP/WelcomeSOAPService?Tester
```

in your web browser to view the `Tester` web page. `WelcomeSOAPService` is the name that clients use to access the web service—this is simply the class name followed by `Service`.

To test `WelcomeSOAP`'s `welcome` web method, type your name in the text field to the right of the **welcome** button then click the button to invoke the method. Figure 32.6 shows the results of invoking `WelcomeSOAP`'s `welcome` method with the value `Paul`.



welcome Method invocation

Method parameter(s)

Type	Value
java.lang.String	Paul

Method returned

java.lang.String : "**Welcome to JAX-WS web services with SOAP, Paul!**"

Fig. 32.6 | Results of testing `WelcomeSOAP`'s `welcome` method.

Application Server Note

You can access the web service only when the application server is running. If NetBeans launches GlassFish for you, it will automatically shut it down when you close NetBeans. To keep it running, you can launch it independently of NetBeans before you deploy or run web applications. The GlassFish Quick Start Guide at

```
    https://glassfish.java.net/docs/4.0/quick-start-guide.pdf
```

shows how to manually start and stop the server.

Testing the `WelcomeSOAP` Web Service from Another Computer

If your computer is connected to a network and allows HTTP requests, then you can test the web service from another computer on the network by typing the following URL (where *host* is the hostname or IP address of the computer on which the web service is deployed) into a browser on another computer:

```
    http://host:8080/WelcomeSOAP/WelcomeSOAPService?Tester
```

32.6.5 Describing a Web Service with the Web Service Description Language (WSDL)

To consume a web service, a client must determine its functionality and how to use it. For this purpose, web services normally contain a **service description**. This is an XML document that conforms to the **Web Service Description Language** (**WSDL**)—an XML vocabulary that defines the methods a web service makes available and how clients interact with them. The WSDL document also specifies lower-level information that clients might need, such as the required formats for requests and responses.

WSDL documents help applications determine how to interact with the web services described in the documents. You do not need to understand WSDL to take advantage of it—the GlassFish application server generates a web service's WSDL dynamically for you, and client tools can parse the WSDL to help create the client-side service endpoint interface class that a client uses to access the web service. Since GlassFish (and most other servers) generate the WSDL dynamically, clients always receive a deployed web service's most up-to-date description. To access the `WelcomeSOAP` web service, the client code will need the following WSDL URL:

```
    http://localhost:8080/WelcomeSOAP/WelcomeSOAPService?WSDL
```

Accessing the `WelcomeSOAP` Web Service's WSDL from Another Computer

Eventually, you'll want clients on other computers to use your web service. Such clients need the web service's WSDL, which they would access with the following URL:

```
    http://host:8080/WelcomeSOAP/WelcomeSOAPService?WSDL
```

where *host* is the hostname or IP address of the server that hosts the web service. As we discussed in Section 32.6.4, this works only if your computer allows HTTP connections from other computers—as is the case for publicly accessible web and application servers.

32.6.6 Creating a Client to Consume the `WelcomeSOAP` Web Service

Now you'll consume the web service from a client application. A web service client can be any type of application or even another web service. You enable a client application to consume a web service by **adding a web service reference** to the application.

Service Endpoint Interface (SEI)

An application that consumes a web service consists of an object of a service endpoint interface (SEI) class (sometimes called a *proxy class*) that's used to interact with the web service and a client application that consumes the web service by invoking methods on the service endpoint interface object. The client code invokes methods on the service endpoint interface object, which handles the details of passing method arguments to and receiving return values from the web service on the client's behalf. This communication can occur over a local network, over the Internet or even with a web service on the same computer. The web service performs the corresponding task and returns the results to the service endpoint interface object, which then returns the results to the client code. Figure 32.7 depicts the interactions among the client code, the SEI object and the web service. As you'll soon see, NetBeans creates these service endpoint interface classes for you.

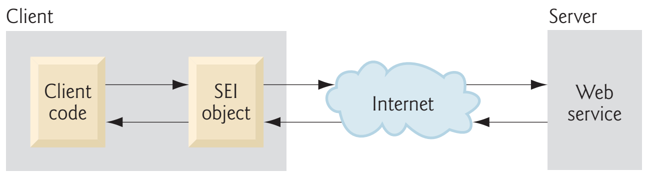


Fig. 32.7 | Interaction between a web service client and a web service.

Requests to and responses from web services created with JAX-WS (one of many different web service frameworks) are typically transmitted via SOAP. Any client capable of generating and processing SOAP messages can interact with a web service, regardless of the language in which the web service is written.

We now use NetBeans to create a client Java desktop GUI application. Then you'll add a web service reference to the project so the client can access the web service. When you add the reference, the IDE creates and compiles the **client-side artifacts**—the framework of Java code that supports the client-side service endpoint interface class. The client then calls methods on an object of the service endpoint interface class, which uses the rest of the artifacts to interact with the web service.

Creating a Desktop Application Project in NetBeans

Before performing the steps in this section, ensure that the `WelcomeSOAP` web service has been deployed and that the GlassFish application server is running (see Section 32.6.3). Perform the following steps to create a client Java desktop application in NetBeans:

1. Select **File > New Project...** to open the **New Project** dialog.
2. Select **Java** from the **Categories** list and **Java Application** from the **Projects** list, then click **Next >**.
3. Specify the name `WelcomeSOAPClient` in the **Project Name** field and uncheck the **Create Main Class** checkbox. Later, you'll add a subclass of `JFrame` that contains a `main` method.
4. Click **Finish** to create the project.

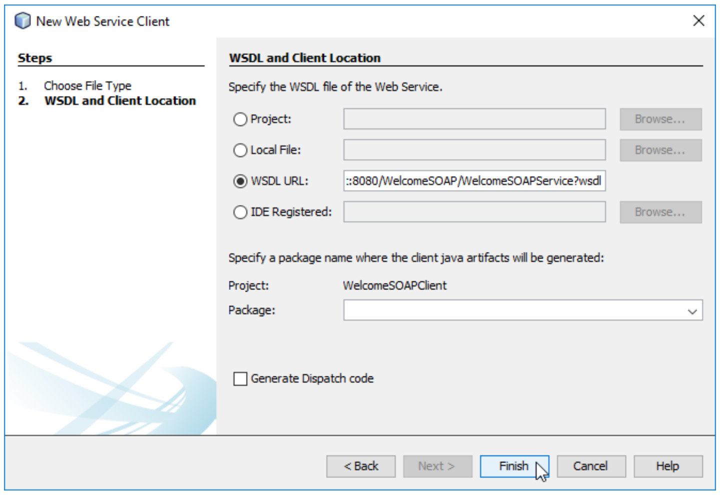
Step 2: Adding a Web Service Reference to an Application

Next, you'll add a web service reference to your application so that it can interact with the `WelcomeSOAP` web service. To add a web service reference, perform the following steps.

1. Right click the project name (`WelcomeSOAPClient`) in the NetBeans **Projects** tab and select **New > Web Service Client...** from the pop-up menu to display the **New Web Service Client** dialog.
2. In the **WSDL URL** field, specify the URL `http://localhost:8080/WelcomeSOAP/WelcomeSOAPService?WSDL` (Fig. 32.8). This URL tells the IDE where to find the web service's WSDL description. [*Note:* If the GlassFish application server is located on a different computer, replace `localhost` with the hostname or IP address of that computer.] The IDE uses this WSDL description to generate the client-side artifacts that compose and support the service endpoint interface.
3. For the other options, leave the default settings, then click **Finish** to create the web service reference and dismiss the **New Web Service Client** dialog.

In the NetBeans **Projects** tab, the `WelcomeSOAPClient` project now contains a **Web Service References** folder with the `WelcomeSOAP` web service's service endpoint interface (Fig. 32.9). The service endpoint interface's name is listed as `WelcomeSOAPService`.

When you specify the web service you want to consume, NetBeans accesses and copies its WSDL information to a file in your project (named `WelcomeSOAPService.wsdl` in this example). You can view this file by double clicking the `WelcomeSOAPService` node in the project's **Web Service References** folder. If the web service changes, the client-side artifacts and the client's copy of the WSDL file can be regenerated by right clicking the `Welcome-SOAPService` node shown in Fig. 32.9 and selecting **Refresh...**. Figure 32.9 also shows the IDE-generated client-side artifacts, which appear in the **Generated Sources (jax-ws)** folder.


New Web Service Client

Steps
1. Choose File Type
2. **WSDL and Client Location**

WSDL and Client Location

Specify the WSDL file of the Web Service.

( ) Project:
( ) Local File:
(●) WSDL URL: ::8080/WelcomeSOAP/WelcomeSOAPService?wsdl
( ) IDE Registered:

Specify a package name where the client java artifacts will be generated:

Project: WelcomeSOAPClient

Package:

☐ Generate Dispatch code

Fig. 32.8 | **New Web Service Client** dialog.

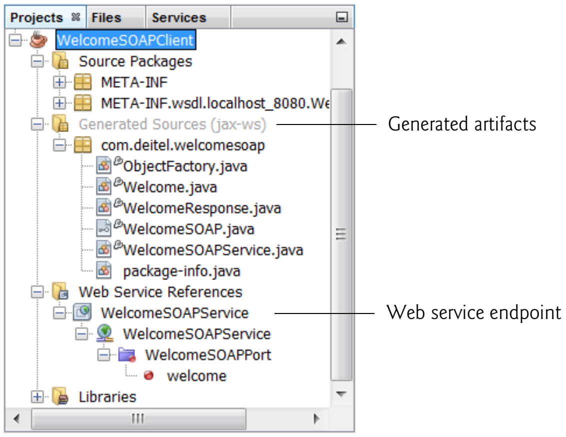

```
WelcomeSOAPClient
  Source Packages
    META-INF
    META-INF.wsdl.localhost_8080.We
  Generated Sources (jax-ws)
    com.deitel.welcomesoap
      ObjectFactory.java
      Welcome.java
      WelcomeResponse.java
      WelcomeSOAP.java
      WelcomeSOAPService.java
      package-info.java
  Web Service References
    WelcomeSOAPService
      WelcomeSOAPService
        WelcomeSOAPPort
          welcome
  Libraries
```

Fig. 32.9 | NetBeans **Project** tab after adding a web service reference to the project.

32.6.7 Consuming the `WelcomeSOAP` Web Service

For this example, we use a GUI application to interact with the `WelcomeSOAP` web service. To build the client application's GUI, add a subclass of `JFrame` to the project by performing the following steps:

1. Right click the project name (`WelcomeSOAPClient`) in the NetBeans **Project** tab and select **New > JFrame Form...** to display the **New JFrame Form** dialog.
2. Specify `WelcomeSOAPClientJFrame` in the **Class Name** field.
3. Specify `com.deitel.welcomesoapclient` in the **Package** field.
4. Click **Finish** to close the **New JFrame Form** dialog.

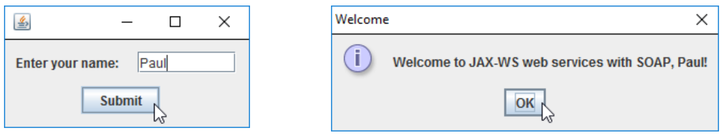
Next, use the NetBeans GUI design tools to build the GUI shown in the sample screen captures at the end of Fig. 32.10. The GUI consists of a **Label**, a **Text Field** and a **Button**.

The application in Fig. 32.10 uses the `WelcomeSOAP` web service to display a welcome message to the user. To save space, we do not show the NetBeans autogenerated `initComponents` method, which contains the code that creates the GUI components, positions them and registers their event handlers. To view the complete source code, open the `WelcomeSOAPClientJFrame.java` file in this example's folder under `src\java\com\deitel\welcomesoapclient`. NetBeans places the GUI component instance-variable declarations at the end of the class (lines 114–116). Java allows instance variables to be declared anywhere in a class's body as long as they're placed outside the class's methods. We continue to declare our own instance variables at the top of the class.

```java
 1  // Fig. 32.10: WelcomeSOAPClientJFrame.java
 2  // Client desktop application for the WelcomeSOAP web service.
 3  package com.deitel.welcomesoapclient;
 4
 5  import com.deitel.welcomesoap.WelcomeSOAP;
 6  import com.deitel.welcomesoap.WelcomeSOAPService;
 7  import javax.swing.JOptionPane;
 8
 9  public class WelcomeSOAPClientJFrame extends javax.swing.JFrame
10  {
11     // references the service endpoint interface object (i.e., the proxy)
12     private WelcomeSOAP welcomeSOAPProxy;
13
14     // no-argument constructor
15     public WelcomeSOAPClientJFrame()
16     {
17        initComponents();
18
19        try
20        {
21           // create the objects for accessing the WelcomeSOAP web service
22           WelcomeSOAPService service = new WelcomeSOAPService();
23           welcomeSOAPProxy = service.getWelcomeSOAPPort();
24        }
25        catch (Exception exception)
26        {
27           exception.printStackTrace();
28           System.exit(1);
29        }
30     }
31
32     // The initComponents method is autogenerated by NetBeans and is called
33     // from the constructor to initialize the GUI. This method is not shown
34     // here to save space. Open WelcomeSOAPClientJFrame.java in this
35     // example's folder to view the complete generated code.
36
```

Fig. 32.10 | Client desktop application for the `WelcomeSOAP` web service. (Part 1 of 2.)

```
87     // call the web service with the supplied name and display the message
88     private void submitJButtonActionPerformed(
89        java.awt.event.ActionEvent evt)
90     {
91        String name = nameJTextField.getText(); // get name from JTextField
92
93        // retrieve the welcome string from the web service
94        String message = welcomeSOAPProxy.welcome(name);
95        JOptionPane.showMessageDialog(this, message,
96           "Welcome", JOptionPane.INFORMATION_MESSAGE);
97     }
98
99     // main method begins execution
100    public static void main(String args[])
101    {
102       java.awt.EventQueue.invokeLater(
103          new Runnable()
104          {
105             public void run()
106             {
107                new WelcomeSOAPClientJFrame().setVisible(true);
108             }
109          }
110       );
111    }
112
113    // Variables declaration - do not modify
114    private javax.swing.JLabel nameJLabel;
115    private javax.swing.JTextField nameJTextField;
116    private javax.swing.JButton submitJButton;
117    // End of variables declaration
118 }
```



Welcome to JAX-WS web services with SOAP, Paul!

Fig. 32.10 | Client desktop application for the `WelcomeSOAP` web service. (Part 2 of 2.)

Lines 5–6 import the classes `WelcomeSOAP` and `WelcomeSOAPService` that enable the client application to interact with the web service. Notice that we do not have `import` declarations for most of the GUI components used in this example. When you create a GUI in NetBeans, it uses fully qualified class names (such as `javax.swing.JFrame` in line 9), so `import` declarations are unnecessary.

Line 12 declares a variable of type `WelcomeSOAP` that will refer to the service endpoint interface object. Line 22 in the constructor creates an object of type `WelcomeSOAPService`. Line 23 uses this object's `getWelcomeSOAPPort` method to obtain the `WelcomeSOAP` service endpoint interface object that the application uses to invoke the web service's methods.

The **Submit** button handler (lines 88–97) first retrieves the name the user entered from `nameJTextField`. It then calls the `welcome` method on the service endpoint interface object (line 94) to retrieve the welcome message from the web service. This object communicates with the web service on the client's behalf. Once the message has been retrieved, lines 95–96 display it in a message box by calling `JOptionPane`'s `showMessageDialog` method.

32.7 Publishing and Consuming REST-Based XML Web Services

The previous section used a service endpoint interface (proxy) object to pass data to and from a Java web service using the SOAP protocol. Now, we access a Java web service using the REST architecture. We recreate the `WelcomeSOAP` example to return data in plain XML format. You can create a **Web Application** project as you did in Section 32.6 to begin. Name the project `WelcomeRESTXML`.

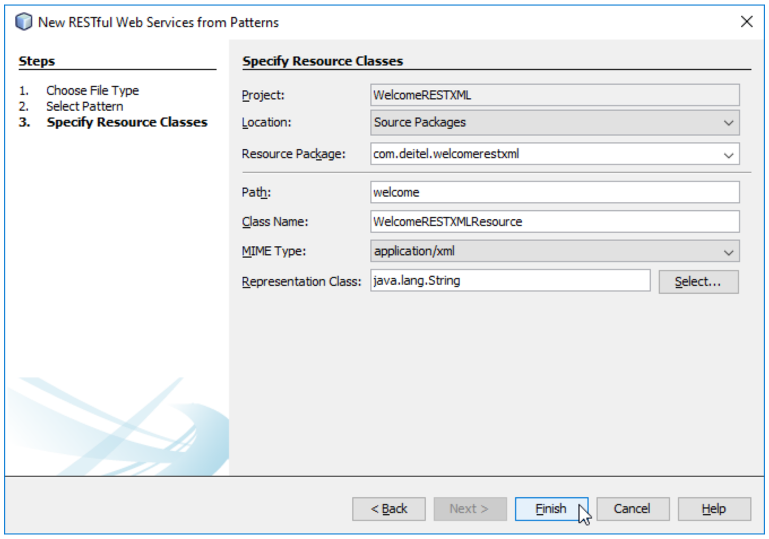
32.7.1 Creating a REST-Based XML Web Service

NetBeans provides various templates for creating RESTful web services, including ones that can interact with databases on the client's behalf. In this chapter, we focus on simple RESTful web services. To create a RESTful web service:

1. Right-click the **WelcomeRESTXML** node in the **Projects** tab, and select **New > Other…** to display the **New File** dialog.
2. Select **Web Services** under **Categories**, then select **RESTful Web Services from Patterns** and click **Next >**.
3. Under **Select Pattern**, ensure **Simple Root Resource** is selected, and click **Next >**.
4. Set the **Resource Package** to `com.deitel.welcomerestxml`, the **Path** to `welcome` and the **Class Name** to `WelcomeRESTXMLResource`. Leave the **MIME Type** and **Representation Class** set to `application/xml` and `java.lang.String`, respectively. The correct configuration is shown in Fig. 32.11.
5. Click **Finish** to create the web service.

NetBeans generates the class and sets up the proper annotations. The class is placed in the project's **RESTful Web Services** folder. The code for the completed service is shown in Fig. 32.12. You'll notice that the completed code does not include some of the code generated by NetBeans. We removed the pieces that were unnecessary for this simple web service. The autogenerated `putXml` method is not necessary, because this example does not modify state on the server. The `UriInfo` instance variable is not needed, because we do not use HTTP query parameters. We also removed the autogenerated constructor, because we have no code to place in it.

Lines 6–9 contain the `import`s for the JAX-RS annotations that help define the RESTful web service. The **`@Path` annotation** on the `WelcomeRESTXMLResource` class (line 12) indicates the URI for accessing the web service. This URI is appended to the web application project's URL to invoke the service. Methods of the class can also use the `@Path` annotation (line 17). Parts of the path specified in curly braces indicate parameters—they're placeholders for values that are passed to the web service as part of the path.



Specify Resource Classes

Project: WelcomeRESTXML
Location: Source Packages
Resource Package: com.deitel.welcomerestxml
Path: welcome
Class Name: WelcomeRESTXMLResource
MIME Type: application/xml
Representation Class: java.lang.String

Fig. 32.11 | Creating the `WelcomeRESTXML` RESTful web service.

```java
// Fig. 32.12: WelcomeRESTXMLResource.java
// REST web service that returns a welcome message as XML.
package com.deitel.welcomerestxml;

import java.io.StringWriter;
import javax.ws.rs.GET; // annotation to indicate method uses HTTP GET
import javax.ws.rs.Path; // annotation to specify path of resource
import javax.ws.rs.PathParam; // annotation to get parameters from URI
import javax.ws.rs.Produces; // annotation to specify type of data
import javax.xml.bind.JAXB; // utility class for common JAXB operations

@Path("welcome") // URI used to access the resource
public class WelcomeRESTXMLResource
{
   // retrieve welcome message
   @GET // handles HTTP GET requests
   @Path("{name}") // URI component containing parameter
   @Produces("application/xml") // response formatted as XML
   public String getXml(@PathParam("name") String name)
   {
      String message = "Welcome to JAX-RS web services with REST and " +
         "XML, " + name + "!"; // our welcome message
      StringWriter writer = new StringWriter();
      JAXB.marshal(message, writer); // marshal String as XML
      return writer.toString(); // return XML as String
   }
}
```

Fig. 32.12 | REST web service that returns a welcome message as XML.

The base path for the service is the project's `webresources` directory. For example, to get a welcome message for someone named John, the complete URL is

```
http://localhost:8080/WelcomeRESTXML/webresources/welcome/John
```

Arguments in a URL can be used as arguments to a web service method. To do so, you bind the parameters specified in the `@Path` specification to parameters of the web service method with the **`@PathParam` annotation**, as shown in line 19. When the request is received, the server passes the argument(s) in the URL to the appropriate parameter(s) in the web service method.

The **`@GET` annotation** denotes that this method is accessed via an HTTP `GET` request. The `putXml` method the IDE created for us had an `@PUT` annotation, which indicates that the method is accessed using the HTTP `PUT` method. Similar annotations exist for HTTP `POST`, `DELETE` and `HEAD` requests.

The **`@Produces` annotation** denotes the content type returned to the client. It's possible to have multiple methods with the same HTTP method and path but different `@Produces` annotations, and JAX-RS will call the method matching the content type requested by the client. Standard Java method overloading rules apply, so such methods must have different names. The **`@Consumes` annotation** for the autogenerated `putXml` method (which we deleted) restricts the content type that the web service will accept from a `PUT` operation.

Line 10 imports the **`JAXB` class** from package `javax.xml.bind`. **JAXB (Java Architecture for XML Binding)** is a set of classes for converting POJOs to and from XML. There are many related classes in the same package that implement the serializations we perform, but the `JAXB` class contains easy-to-use wrappers for common operations. After creating the welcome message (lines 21–22), we create a `StringWriter` (line 23) to which JAXB will output the XML. Line 24 calls the `JAXB` class's `static` method **`marshal`** to convert the `String` containing our message to XML format. Line 25 calls `StringWriter`'s `toString` method to retrieve the XML text to return to the client.

Testing the RESTful Web Service

Section 32.6.4 demonstrated testing a SOAP service using GlassFish's `Tester` page. GlassFish does not provide a testing facility for RESTful services, but you can enter the web service's URL directly in your browser to test the web service. To do so:

1. First, deploy the web service's project. Right click the `WelcomeRESTXML` project in the NetBeans **Projects** tab and select **Deploy**. This will compile and deploy the web service, if you have not yet done so.
2. Open a web browser and enter the following URL in the browser's address bar: `http://localhost:8080/WelcomeRESTXML/webresources/welcome/Paul`—you can replace `Paul` with your own name.

Once GlassFish deploys a REST web service, a client can access it on the server (in this case, `localhost` at port `8080`) at the location

/*ProjectName*/`webresources`/*methodName*

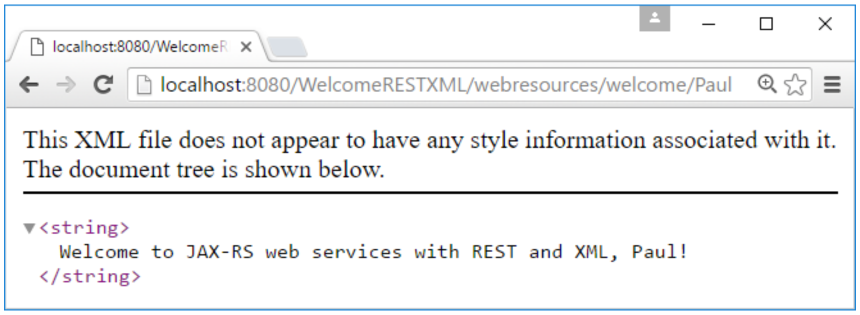
If the method requires parameters, as in this example, each parameter follows the method name in the form

/*argument*

and the arguments are passed to the method's parameters in the same order as they're declared in the method's parameter list. So the URL

```
http://localhost:8080/WelcomeRESTXML/webresources/welcome/Paul
```

invokes the `WelcomeRESTXML` web service's `welcome` method and passes `Paul` to the method's `name` parameter. The web service then returns an XML response that's displayed directly in the web browser (Fig. 32.13).



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<string>
   Welcome to JAX-RS web services with REST and XML, Paul!
 </string>
```

Fig. 32.13 | Test page for the `WelcomeRESTXML` web service.

WADL

WADL (Web Application Description Language) has similar design goals to WSDL, but describes RESTful services instead of SOAP services. You can access this app's WADL at

```
http://localhost:8080/WelcomeRESTJSON/webresources/application.wadl
```

Client-code-generation tools can use this description to help implement a client that interacts with this web service.

32.7.2 Consuming a REST-Based XML Web Service

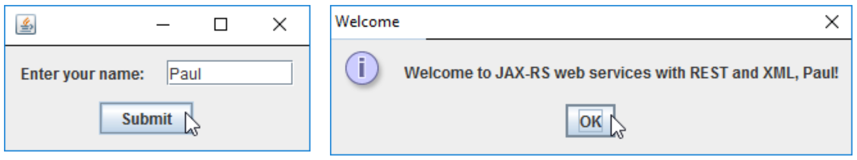
As we did with SOAP, we create a Java application that retrieves the welcome message from the web service and displays it to the user. First, create a Java application with the name `WelcomeRESTXMLClient`. RESTful web services do *not* require web service references, so you can begin building the GUI immediately by creating a `JFrame` form called `WelcomeRESTXMLClientJFrame` and placing it in the `com.deitel.welcomerestxmlclient` package. The GUI is identical to the one in Fig. 32.10, including the names of the GUI elements. To create the GUI quickly, you can simply copy and paste the GUI from the **Design** view of the `WelcomeSOAPClientJFrame` class and paste it into the **Design** view of the `WelcomeRESTXMLClientJFrame` class. Figure 32.14 contains the completed code.

```
// Fig. 32.14: WelcomeRESTXMLClientJFrame.java
// Client that consumes the WelcomeRESTXML service.
package com.deitel.welcomerestxmlclient;

import javax.swing.JOptionPane;
```

Fig. 32.14 | Client that consumes the `WelcomeRESTXML` service. (Part 1 of 3.)

```java
import javax.xml.bind.JAXB; // utility class for common JAXB operations

public class WelcomeRESTXMLClientJFrame extends javax.swing.JFrame
{
   // no-argument constructor
   public WelcomeRESTXMLClientJFrame()
   {
      initComponents();
   }

   // The initComponents method is autogenerated by NetBeans and is called
   // from the constructor to initialize the GUI. This method is not shown
   // here to save space. Open WelcomeRESTXMLClientJFrame.java in this
   // example's folder to view the complete generated code.

   // call the web service with the supplied name and display the message
   private void submitJButtonActionPerformed(
      java.awt.event.ActionEvent evt)
   {
      String name = nameJTextField.getText(); // get name from JTextField

      // the URL for the REST service
      String url = "http://localhost:8080/WelcomeRESTXML/" +
         "webresources/welcome/" + name;

      // read from URL and convert from XML to Java String
      String message = JAXB.unmarshal(url, String.class);

      // display the message to the user
      JOptionPane.showMessageDialog(this, message,
            "Welcome", JOptionPane.INFORMATION_MESSAGE);
   }

   // main method begins execution
   public static void main(String args[])
   {
      java.awt.EventQueue.invokeLater(
         new Runnable()
         {
            public void run()
            {
               new WelcomeRESTXMLClientJFrame().setVisible(true);
            }
         }
      );
   }

   // Variables declaration - do not modify
   private javax.swing.JLabel nameJLabel;
   private javax.swing.JTextField nameJTextField;
   private javax.swing.JButton submitJButton;
   // End of variables declaration
}
```

Fig. 32.14 | Client that consumes the `WelcomeRESTXML` service. (Part 2 of 3.)



Welcome to JAX-RS web services with REST and XML, Paul!

Fig. 32.14 | Client that consumes the `WelcomeRESTXML` service. (Part 3 of 3.)

You can access a RESTful web service with classes from Java API. As in the RESTful XML web service, we use the JAXB library. The JAXB class (imported on line 6) has a `static` **`unmarshal`** method that takes as arguments a filename or URL as a `String`, and a `Class<T>` object indicating the Java class to which the XML will be converted (line 83). In this example, the XML contains a `String` object, so we use the Java compiler shortcut `String.class` to create the `Class<String>` object we need as the second argument. The `String` returned from the call to the `unmarshal` method is then displayed to the user via `JOptionPane`'s `showMessageDialog` method (lines 86–87), as it was with the SOAP service. The URL used in this example to extract data from the web service matches the URL we used to test the web service directly in a web browser.

32.8 Publishing and Consuming REST-Based JSON Web Services

While XML was designed primarily as a document interchange format, JSON is designed as a *data* exchange format. Data structures in most programming languages do not map directly to XML constructs—for example, the distinction between elements and attributes is not present in programming-language data structures. JSON is a subset of the JavaScript programming language, and its components—objects, arrays, strings, numbers—can be easily mapped to constructs in Java and other programming languages.

The standard Java libraries do not currently provide capabilities for working with JSON, but there are many open-source JSON libraries for Java and other languages; you can find a list of them at `json.org`. We chose the Gson library from `https://github.com/google/gson`, which provides a simple way to convert POJOs to and from JSON. A JAR file containing the library can be downloaded from

```
http://search.maven.org/remotecontent?filepath=com/google/code/
   gson/gson/2.7/gson-2.7.jar
```

32.8.1 Creating a REST-Based JSON Web Service

To begin, create a `WelcomeRESTJSON` web application, then create the web service by following the steps in Section 32.7.1. In *Step 4*, change the **Resource Package** to `com.deitel.welcomerestjson`, the **Class Name** to `WelcomeRESTJSONResource` and the **MIME Type** to `application/json`. Additionally, you must download the Gson library's JAR file, then add it to the project as a library. To add the JAR file to the project, right click your project's **Libraries** folder, select **Add JAR/Folder...** locate the downloaded Gson JAR file and click **Open**. The complete code for the service is shown in Fig. 32.15.

```
 1  // Fig. 32.15: WelcomeRESTJSONResource.java
 2  // REST web service that returns a welcome message as JSON.
 3  package com.deitel.welcomerestjson;
 4
 5  import com.google.gson.Gson; // converts POJO to JSON and back again
 6  import javax.ws.rs.GET; // annotation to indicate method uses HTTP GET
 7  import javax.ws.rs.Path; // annotation to specify path of resource
 8  import javax.ws.rs.PathParam; // annotation to get parameters from URI
 9  import javax.ws.rs.Produces; // annotation to specify type of data
10
11  @Path("welcome") // path used to access the resource
12  public class WelcomeRESTJSONResource
13  {
14     // retrieve welcome message
15     @GET // handles HTTP GET requests
16     @Path("{name}") // takes name as a path parameter
17     @Produces("application/json") // response formatted as JSON
18     public String getJson(@PathParam("name") String name)
19     {
20        // add welcome message to field of TextMessage object
21        TextMessage message = new TextMessage(); // create wrapper object
22        message.setMessage(String.format("%s, %s!",
23           "Welcome to JAX-RS web services with REST and JSON", name));
24
25        return new Gson().toJson(message); // return JSON-wrapped message
26     }
27  }
28
29  // private class that contains the message we wish to send
30  class TextMessage
31  {
32     private String message; // message we're sending
33
34     // returns the message
35     public String getMessage()
36     {
37        return message;
38     }
39
40     // sets the message
41     public void setMessage(String value)
42     {
43        message = value;
44     }
45  }
```

Fig. 32.15 | REST web service that returns a welcome message as JSON.

All the annotations and the basic structure of the `WelcomeRESTJSONResource` class are the same as REST XML example. The argument to the `@Produces` attribute (line 17) is `"application/json"`. The `TextMessage` class (lines 30–45) addresses a difference between JSON and XML. JSON does not permit strings or numbers to stand on their own—they must be encapsulated in a composite data type. So, we created class `TextMessage` to encapsulate the `String` representing the message.

When a client invokes this web service, line 21 creates the `TextMessage` object, then lines 22–23 set its contained message. Next, line 25 creates a **`Gson`** object (from package `com.google.gson.Gson`) and calls its **`toJson`** method to convert the `TextMessage` into its JSON `String` representation. We return this `String`, which is then sent back to the client in the web service's response. There are multiple overloads of the `toJson` method, such as one that sends its output to a `Writer` instead of returning a `String`.

RESTful services returning JSON can be tested in the same way as those returning XML. Follow the procedure outlined in Section 32.7.1, but use the URL

```
http://localhost:8080/WelcomeRESTJSON/webresources/welcome/Paul
```

to invoke the web service. In this case, the browser will display the JSON response

```
{"message":"Welcome to JAX-RS web services with REST and JSON, Paul!"}
```

32.8.2 Consuming a REST-Based JSON Web Service

We now create a Java application that retrieves the welcome message from the web service and displays it to the user. First, create a Java application with the name `WelcomeRESTJSONClient`. Then, create a `JFrame` form called `WelcomeRESTXMLClientJFrame` and place it in the `com.deitel.welcomerestjsonclient` package. The GUI is identical to the one in Fig. 32.10. To create the GUI quickly, copy it from the **Design** view of the `WelcomeSOAPClientJFrame` class and paste it into the **Design** view of the `WelcomeRESTJSONClientJFrame` class. Figure 32.16 contains the completed code.

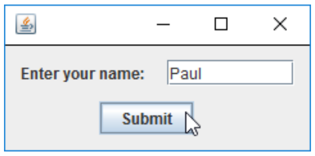
```java
// Fig. 32.16: WelcomeRESTJSONClientJFrame.java
// Client that consumes the WelcomeRESTJSON service.
package com.deitel.welcomerestjsonclient;

import com.google.gson.Gson; // converts POJO to JSON and back again
import java.io.InputStreamReader;
import java.net.URL;
import javax.swing.JOptionPane;

public class WelcomeRESTJSONClientJFrame extends javax.swing.JFrame
{
   // no-argument constructor
   public WelcomeRESTJSONClientJFrame()
   {
      initComponents();
   }

   // The initComponents method is autogenerated by NetBeans and is called
   // from the constructor to initialize the GUI. This method is not shown
   // here to save space. Open WelcomeRESTJSONClientJFrame.java in this
   // example's folder to view the complete generated code.

```

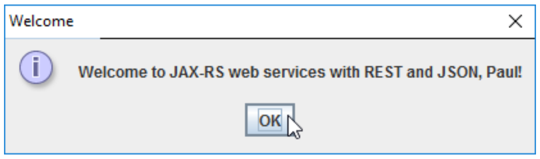
Fig. 32.16 | Client that consumes the `WelcomeRESTJSON` service. (Part 1 of 3.)

```
   // call the web service with the supplied name and display the message
   private void submitJButtonActionPerformed(
      java.awt.event.ActionEvent evt)
   {
      String name = nameJTextField.getText(); // get name from JTextField

      // retrieve the welcome string from the web service
      try
      {
         // the URL of the web service
         String url = "http://localhost:8080/WelcomeRESTJSON/" +
            "webresources/welcome/" + name;

         // open URL, using a Reader to convert bytes to chars
         InputStreamReader reader =
            new InputStreamReader(new URL(url).openStream());

         // parse the JSON back into a TextMessage
         TextMessage message =
            new Gson().fromJson(reader, TextMessage.class);

         // display message to the user
         JOptionPane.showMessageDialog(this, message.getMessage(),
            "Welcome", JOptionPane.INFORMATION_MESSAGE);
      }
      catch (Exception exception)
      {
         exception.printStackTrace(); // show exception details
      }
   }

   // main method begin execution
   public static void main(String args[])
   {
      java.awt.EventQueue.invokeLater(
         new Runnable()
         {
            public void run()
            {
               new WelcomeRESTJSONClientJFrame().setVisible(true);
            }
         }
      );
   }

   // Variables declaration - do not modify
   private javax.swing.JLabel nameJLabel;
   private javax.swing.JTextField nameJTextField;
   private javax.swing.JButton submitJButton;
   // End of variables declaration
}

```

Fig. 32.16 | Client that consumes the `WelcomeRESTJSON` service. (Part 2 of 3.)

```
125 // private class that contains the message we are receiving
126 class TextMessage
127 {
128    private String message; // message we're receiving
129
130    // returns the message
131    public String getMessage()
132    {
133       return message;
134    }
135
136    // sets the message
137    public void setMessage(String value)
138    {
139       message = value;
140    }
141 }
```


Enter your name: Paul



Welcome to JAX-RS web services with REST and JSON, Paul!

Fig. 32.16 | Client that consumes the `WelcomeRESTJSON` service. (Part 3 of 3.)

Lines 83–84 create the URL `String` that is used to invoke the web service. Lines 87–88 create a `URL` object using this `String`, then call the `URL`'s **`openStream`** method to invoke the web service and obtain an `InputStream` from which the client can read the response. The `InputStream` is wrapped in an `InputStreamReader` so it can be passed as the first argument to the `Gson` class's **`fromJson`** method. This method is overloaded. The version we use takes as arguments a `Reader` from which to read a JSON `String` and a `Class<T>` object indicating the Java class to which the JSON `String` will be converted (line 92). In this example, the JSON `String` contains a `TextMessage` object, so we use the Java compiler shortcut `TextMessage.class` to create the `Class<TextMessage>` object we need as the second argument. Lines 95–96 display the message in the `TextMessage` object.

The `TextMessage` classes in the web service and client are unrelated. Technically, the client can be written in any programming language, so the manner in which a response is processed can vary greatly. Since our client is written in Java, we duplicated the `TextMessage` class in the client so we could easily convert the JSON object back to Java.

32.9 Session Tracking in a SOAP Web Service

Section 30.8 described the advantages of using session tracking to maintain client-state information so you can personalize the users' browsing experiences. Now we'll incorporate *session tracking* into a web service. Suppose a client application needs to call several methods from the same web service, possibly several times each. In such a case, it can be beneficial for the web service to maintain state information for the client, thus eliminating the need for client information to be passed between the client and the web service multiple times. For example, a web service that provides local restaurant reviews could store the client user's street address during the initial request, then use it to return personalized, localized results in subsequent requests. Storing session information also enables a web service to distinguish between clients.

32.9.1 Creating a `Blackjack` Web Service

Our next example is a web service that assists you in developing a blackjack card game. The `Blackjack` web service (Fig. 32.17) provides web methods to shuffle a deck of cards, deal a card from the deck and evaluate a hand of cards. After presenting the web service, we use it to serve as the dealer for a game of blackjack (Fig. 32.18). The `Blackjack` web service uses an `HttpSession` object to maintain a unique deck of cards for each client application. Several clients can use the service at the same time, but web method calls made by a specific client use only the deck of cards stored in that client's session. Our example uses the following blackjack rules:

> *Two cards each are dealt to the dealer and the player. The player's cards are dealt face up. Only the first of the dealer's cards is dealt face up. Each card has a value. A card numbered 2 through 10 is worth its face value. Jacks, queens and kings each count as 10. Aces can count as 1 or 11—whichever value is more beneficial to the player (as we'll soon see). If the sum of the player's two initial cards is 21 (i.e., the player was dealt a card valued at 10 and an ace, which counts as 11 in this situation), the player has "blackjack" and immediately wins the game—if the dealer does not also have blackjack (which would result in a "push"—i.e., a tie). Otherwise, the player can begin taking additional cards one at a time. These cards are dealt face up, and the player decides when to stop taking cards. If the player "busts" (i.e., the sum of the player's cards exceeds 21), the game is over and the player loses. When the player is satisfied with the current set of cards, the player "stands" (i.e., stops taking cards), and the dealer's hidden card is revealed. If the dealer's total is 16 or less, the dealer must take another card; otherwise, the dealer must stand. The dealer must continue taking cards until the sum of the dealer's cards is greater than or equal to 17. If the dealer exceeds 21, the player wins. Otherwise, the hand with the higher point total wins. If the dealer and the player have the same point total, the game is a "push," and no one wins. The value of an ace for a dealer depends on the dealer's other card(s) and the casino's house rules. A dealer typically must hit for totals of 16 or less and must stand for totals of 17 or more. However, for a "soft 17"—a hand with a total of 17 with one ace counted as 11—some casinos require the dealer to hit and some require the dealer to stand (we require the dealer to stand). Such a hand is known as a "soft 17" because taking another card cannot bust the hand.*

The web service (Fig. 32.17) stores each card as a `String` consisting of a number, `1`–`13`, representing the card's face (ace through king, respectively), followed by a space and a digit, `0`–`3`, representing the card's suit (hearts, diamonds, clubs or spades, respectively). For example, the jack of clubs is represented as `"11 2"` and the two of hearts as `"2 0"`. To create and deploy this web service, follow the steps that we presented in Sections 32.6.2–32.6.3 for the `WelcomeSOAP` service.

```java
// Fig. 32.17: Blackjack.java
// Blackjack web service that deals cards and evaluates hands
package com.deitel.blackjack;

import com.sun.xml.ws.developer.servlet.HttpSessionScope;
import java.util.ArrayList;
import java.util.Random;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

@HttpSessionScope // enable web service to maintain session state
@WebService()
public class Blackjack
{
   private ArrayList<String> deck; // deck of cards for one user session
   private static final Random randomObject = new Random();

   // deal one card
   @WebMethod(operationName = "dealCard")
   public String dealCard()
   {
      String card = "";
      card = deck.get(0); // get top card of deck
      deck.remove(0); // remove top card of deck
      return card;
   }

   // shuffle the deck
   @WebMethod(operationName = "shuffle")
   public void shuffle()
   {
      // create new deck when shuffle is called
      deck = new ArrayList<String>();

      // populate deck of cards
      for (int face = 1; face <= 13; face++) // loop through faces
         for (int suit = 0; suit <= 3; suit++) // loop through suits
            deck.add(face + " " + suit); // add each card to deck

      String tempCard; // holds card temporarily during swapping
      int index; // index of randomly selected card

      for (int i = 0; i < deck.size() ; i++) // shuffle
      {
         index = randomObject.nextInt(deck.size() - 1);

         // swap card at position i with randomly selected card
         tempCard = deck.get(i);
         deck.set(i, deck.get(index));
         deck.set(index, tempCard);
      }
   }
```

Fig. 32.17 | Blackjack web service that deals cards and evaluates hands. (Part 1 of 2.)

```

   // determine a hand's value
   @WebMethod(operationName = "getHandValue")
   public int getHandValue(@WebParam(name = "hand") String hand)
   {
      // split hand into cards
      String[] cards = hand.split("\t");
      int total = 0; // total value of cards in hand
      int face; // face of current card
      int aceCount = 0; // number of aces in hand

      for (int i = 0; i < cards.length; i++)
      {
         // parse string and get first int in String
         face = Integer.parseInt(
            cards[i].substring(0, cards[i].indexOf(" ")));

         switch (face)
         {
            case 1: // if ace, increment aceCount
               ++aceCount;
               break;
            case 11: // jack
            case 12: // queen
            case 13: // king
               total += 10;
               break;
            default: // otherwise, add face
               total += face;
               break;
         }
      }

      // calculate optimal use of aces
      if (aceCount > 0)
      {
         // if possible, count one ace as 11
         if (total + 11 + aceCount - 1 <= 21)
            total += 11 + aceCount - 1;
         else // otherwise, count all aces as 1
            total += aceCount;
       }

      return total;
   }
}
```

Fig. 32.17 | `Blackjack` web service that deals cards and evaluates hands. (Part 2 of 2.)

Session Tracking in Web Services: `@HttpSessionScope` *Annotation*

In JAX-WS 2.2, it's easy to enable session tracking in a web service. You simply precede your web service class with the **`@HttpSessionScope` annotation**. This annotation is located in package `com.sun.xml.ws.developer.servlet`. To use this package you must add the JAX-WS 2.2 library to your project. To do so, right click the **Libraries** node in your `Black-`

jack web application project and select **Add Library…**. Then, in the dialog that appears, locate and select **JAX-WS 2.2**, then click **Add Library**. Once a web service is annotated with @HttpSessionScope, the server automatically maintains a separate instance of the class for each client session. Thus, the deck instance variable (line 16) will be maintained separately for each client.

Client Interactions with the Blackjack Web Service

A client first calls the Blackjack web service's shuffle web method (lines 30–53) to create a new deck of cards (line 34), populate it (lines 37–39) and shuffle it (lines 41–52). Lines 37–39 generate Strings in the form "*face suit*" to represent each possible card in the deck.

Lines 20–27 define the dealCard web method. Method shuffle *must* be called before method dealCard is called the first time for a client—otherwise, deck could be null. The method gets the top card from the deck (line 24), removes it from the deck (line 25) and returns the card's value as a String (line 26). Without using session tracking, the deck of cards would need to be passed back and forth with each method call. Session tracking makes the dealCard method easy to call (it requires no arguments) and eliminates the overhead of sending the deck over the network multiple times.

Method getHandValue (lines 56–98) determines the total value of the cards in a hand by trying to attain the highest score possible without going over 21. Recall that an ace can be counted as either 1 or 11, and all face cards count as 10. This method does not use the session object, because the deck of cards is not used in this method.

As you'll soon see, the client application maintains a hand of cards as a String in which each card is separated by a tab character. Line 60 splits the hand of cards (represented by hand) into individual cards by calling String method split and passing to it a String containing the delimiter characters (in this case, just a tab). Method split uses the delimiter characters to separate tokens in the String. Lines 65–85 count the value of each card. Lines 68–69 retrieve the first integer—the face—and use that value in the switch statement (lines 71–84). If the card is an ace, the method increments variable aceCount. We discuss how this variable is used shortly. If the card is an 11, 12 or 13 (jack, queen or king), the method adds 10 to the total value of the hand (line 79). If the card is anything else, the method increases the total by that value (line 82).

Because an ace can have either of two values, additional logic is required to process aces. Lines 88–95 process the aces after all the other cards. If a hand contains several aces, only one ace can be counted as 11. The condition in line 91 determines whether counting one ace as 11 and the rest as 1 will result in a total that does not exceed 21. If this is possible, line 92 adjusts the total accordingly. Otherwise, line 94 adjusts the total, counting each ace as 1.

Method getHandValue maximizes the value of the current cards without exceeding 21. Imagine, for example, that the dealer has a 7 and receives an ace. The new total could be either 8 or 18. However, getHandValue always maximizes the value of the cards without going over 21, so the new total is 18.

32.9.2 Consuming the Blackjack Web Service

The blackjack application in Fig. 32.18 keeps track of the player's and dealer's cards, and the web service tracks the cards that have been dealt. The constructor (lines 34–83) sets up the GUI (line 36), changes the window's background color (line 40) and creates the Blackjack web service's service endpoint interface object (lines 46–47). In the GUI, each player has 11 `JLabel`s—the maximum number of cards that can be dealt without automatically exceeding 21 (i.e., four aces, four twos and three threes). These `JLabel`s are placed in an `ArrayList` of `JLabel`s (lines 59–82), so we can index the `ArrayList` during the game to determine the `JLabel` that will display a particular card image.

```
 1  // Fig. 32.18: BlackjackGameJFrame.java
 2  // Blackjack game that uses the Blackjack Web Service.
 3  package com.deitel.blackjackclient;
 4
 5  import com.deitel.blackjack.Blackjack;
 6  import com.deitel.blackjack.BlackjackService;
 7  import java.awt.Color;
 8  import java.util.ArrayList;
 9  import javax.swing.ImageIcon;
10  import javax.swing.JLabel;
11  import javax.swing.JOptionPane;
12  import javax.xml.ws.BindingProvider;
13
14  public class BlackjackGameJFrame extends javax.swing.JFrame
15  {
16     private String playerCards;
17     private String dealerCards;
18     private ArrayList<JLabel> cardboxes; // list of card image JLabels
19     private int currentPlayerCard; // player's current card number
20     private int currentDealerCard; // blackjackProxy's current card number
21     private BlackjackService blackjackService; // used to obtain proxy
22     private Blackjack blackjackProxy; // used to access the web service
23
24     // enum of game states
25     private enum GameStatus
26     {
27        PUSH, // game ends in a tie
28        LOSE, // player loses
29        WIN, // player wins
30        BLACKJACK // player has blackjack
31     }
32
33     // no-argument constructor
34     public BlackjackGameJFrame()
35     {
36        initComponents();
37
38        // due to a bug in NetBeans, we must change the JFrame's background
39        // color here rather than in the designer
40        getContentPane().setBackground(new Color(0, 180, 0));
41
42        // initialize the blackjack proxy
43        try
44        {
45           // create the objects for accessing the Blackjack web service
46           blackjackService = new BlackjackService();
47           blackjackProxy = blackjackService.getBlackjackPort();
```

Fig. 32.18 | Blackjack game that uses the `Blackjack` web service. (Part 1 of 10.)

```java

            // enable session tracking
            ((BindingProvider) blackjackProxy).getRequestContext().put(
               BindingProvider.SESSION_MAINTAIN_PROPERTY, true);
         }
         catch (Exception e)
         {
            e.printStackTrace();
         }

         // add JLabels to cardBoxes ArrayList for programmatic manipulation
         cardboxes = new ArrayList<JLabel>();

         cardboxes.add(dealerCard1JLabel);
         cardboxes.add(dealerCard2JLabel);
         cardboxes.add(dealerCard3JLabel);
         cardboxes.add(dealerCard4JLabel);
         cardboxes.add(dealerCard5JLabel);
         cardboxes.add(dealerCard6JLabel);
         cardboxes.add(dealerCard7JLabel);
         cardboxes.add(dealerCard8JLabel);
         cardboxes.add(dealerCard9JLabel);
         cardboxes.add(dealerCard10JLabel);
         cardboxes.add(dealerCard11JLabel);
         cardboxes.add(playerCard1JLabel);
         cardboxes.add(playerCard2JLabel);
         cardboxes.add(playerCard3JLabel);
         cardboxes.add(playerCard4JLabel);
         cardboxes.add(playerCard5JLabel);
         cardboxes.add(playerCard6JLabel);
         cardboxes.add(playerCard7JLabel);
         cardboxes.add(playerCard8JLabel);
         cardboxes.add(playerCard9JLabel);
         cardboxes.add(playerCard10JLabel);
         cardboxes.add(playerCard11JLabel);
      }

      // play the dealer’s hand
      private void dealerPlay()
      {
         try
         {
            // while the value of the dealers's hand is below 17
            // the dealer must continue to take cards
            String[] cards = dealerCards.split("\t");

            // display dealer's cards
            for (int i = 0; i < cards.length; i++)
            {
               displayCard(i, cards[i]);
            }

```

Fig. 32.18 | Blackjack game that uses the `Blackjack` web service. (Part 2 of 10.)

```java
            while (blackjackProxy.getHandValue(dealerCards) < 17)
            {
               String newCard = blackjackProxy.dealCard(); // deal new card
               dealerCards += "\t" + newCard; // deal new card
               displayCard(currentDealerCard, newCard);
               ++currentDealerCard;
               JOptionPane.showMessageDialog(this, "Dealer takes a card",
                  "Dealer's turn", JOptionPane.PLAIN_MESSAGE);
            }

            int dealersTotal = blackjackProxy.getHandValue(dealerCards);
            int playersTotal = blackjackProxy.getHandValue(playerCards);

            // if dealer busted, player wins
            if (dealersTotal > 21)
            {
               gameOver(GameStatus.WIN);
               return;
            }

            // if dealer and player are below 21
            // higher score wins, equal scores is a push
            if (dealersTotal > playersTotal)
            {
               gameOver(GameStatus.LOSE);
            }
            else if (dealersTotal < playersTotal)
            {
               gameOver(GameStatus.WIN);
            }
            else
            {
               gameOver(GameStatus.PUSH);
            }
         }
         catch (Exception e)
         {
            e.printStackTrace();
         }
      }

      // displays the card represented by cardValue in specified JLabel
      private void displayCard(int card, String cardValue)
      {
         try
         {
            // retrieve correct JLabel from cardBoxes
            JLabel displayLabel = cardboxes.get(card);

            // if string representing card is empty, display back of card
            if (cardValue.equals(""))
            {
```

Fig. 32.18 | Blackjack game that uses the `Blackjack` web service. (Part 3 of 10.)

```java
               displayLabel.setIcon(new ImageIcon(getClass().getResource(
                  "/com/deitel/java/blackjackclient/" +
                  "blackjack_images/cardback.png")));
               return;
            }

            // retrieve the face value of the card
            String face = cardValue.substring(0, cardValue.indexOf(" "));

            // retrieve the suit of the card
            String suit =
               cardValue.substring(cardValue.indexOf(" ") + 1);

            char suitLetter; // suit letter used to form image file

            switch (Integer.parseInt(suit))
            {
               case 0: // hearts
                  suitLetter = 'h';
                  break;
               case 1: // diamonds
                  suitLetter = 'd';
                  break;
               case 2: // clubs
                  suitLetter = 'c';
                  break;
               default: // spades
                  suitLetter = 's';
                  break;
            }

            // set image for displayLabel
            displayLabel.setIcon(new ImageIcon(getClass().getResource(
               "/com/deitel/java/blackjackclient/blackjack_images/" +
               face + suitLetter + ".png")));
         }
         catch (Exception e)
         {
            e.printStackTrace();
         }
      }

      // displays all player cards and shows appropriate message
      private void gameOver(GameStatus winner)
      {
         String[] cards = dealerCards.split("\t");

         // display blackjackProxy's cards
         for (int i = 0; i < cards.length; i++)
         {
            displayCard(i, cards[i]);
         }

```

Fig. 32.18 | Blackjack game that uses the `Blackjack` web service. (Part 4 of 10.)

```
      // display appropriate status image
      if (winner == GameStatus.WIN)
      {
         statusJLabel.setText("You win!");
      }
      else if (winner == GameStatus.LOSE)
      {
         statusJLabel.setText("You lose.");
      }
      else if (winner == GameStatus.PUSH)
      {
         statusJLabel.setText("It's a push.");
      }
      else // blackjack
      {
         statusJLabel.setText("Blackjack!");
      }

      // display final scores
      int dealersTotal = blackjackProxy.getHandValue(dealerCards);
      int playersTotal = blackjackProxy.getHandValue(playerCards);
      dealerTotalJLabel.setText("Dealer: " + dealersTotal);
      playerTotalJLabel.setText("Player: " + playersTotal);

      // reset for new game
      standJButton.setEnabled(false);
      hitJButton.setEnabled(false);
      dealJButton.setEnabled(true);
   }

   // The initComponents method is autogenerated by NetBeans and is called
   // from the constructor to initialize the GUI. This method is not shown
   // here to save space. Open BlackjackGameJFrame.java in this
   // example's folder to view the complete generated code

   // handles dealJButton click
   private void dealJButtonActionPerformed(
      java.awt.event.ActionEvent evt)
   {
      String card; // stores a card temporarily until it's added to a hand

      // clear card images
      for (int i = 0; i < cardboxes.size(); i++)
      {
         cardboxes.get(i).setIcon(null);
      }

      statusJLabel.setText("");
      dealerTotalJLabel.setText("");
      playerTotalJLabel.setText("");

      // create a new, shuffled deck on remote machine
      blackjackProxy.shuffle();
```

Fig. 32.18 | Blackjack game that uses the `Blackjack` web service. (Part 5 of 10.)

```java

      // deal two cards to player
      playerCards = blackjackProxy.dealCard(); // add first card to hand
      displayCard(11, playerCards); // display first card
      card = blackjackProxy.dealCard(); // deal second card
      displayCard(12, card); // display second card
      playerCards += "\t" + card; // add second card to hand

      // deal two cards to blackjackProxy, but only show first
      dealerCards = blackjackProxy.dealCard(); // add first card to hand
      displayCard(0, dealerCards); // display first card
      card = blackjackProxy.dealCard(); // deal second card
      displayCard(1, ""); // display back of card
      dealerCards += "\t" + card; // add second card to hand

      standJButton.setEnabled(true);
      hitJButton.setEnabled(true);
      dealJButton.setEnabled(false);

      // determine the value of the two hands
      int dealersTotal = blackjackProxy.getHandValue(dealerCards);
      int playersTotal = blackjackProxy.getHandValue(playerCards);

      // if hands both equal 21, it is a push
      if (playersTotal == dealersTotal && playersTotal == 21)
      {
         gameOver(GameStatus.PUSH);
      }
      else if (dealersTotal == 21) // blackjackProxy has blackjack
      {
         gameOver(GameStatus.LOSE);
      }
      else if (playersTotal == 21) // blackjack
      {
         gameOver(GameStatus.BLACKJACK);
      }

      // next card for blackjackProxy has index 2
      currentDealerCard = 2;

      // next card for player has index 13
      currentPlayerCard = 13;
   }

   // handles standJButton click
   private void hitJButtonActionPerformed(
      java.awt.event.ActionEvent evt)
   {
      // get player another card
      String card = blackjackProxy.dealCard(); // deal new card
      playerCards += "\t" + card; // add card to hand

```

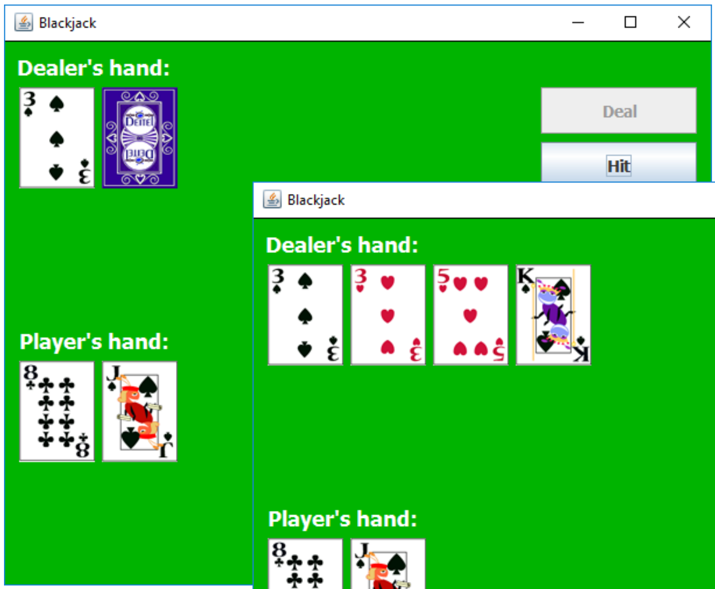
Fig. 32.18 | Blackjack game that uses the `Blackjack` web service. (Part 6 of 10.)

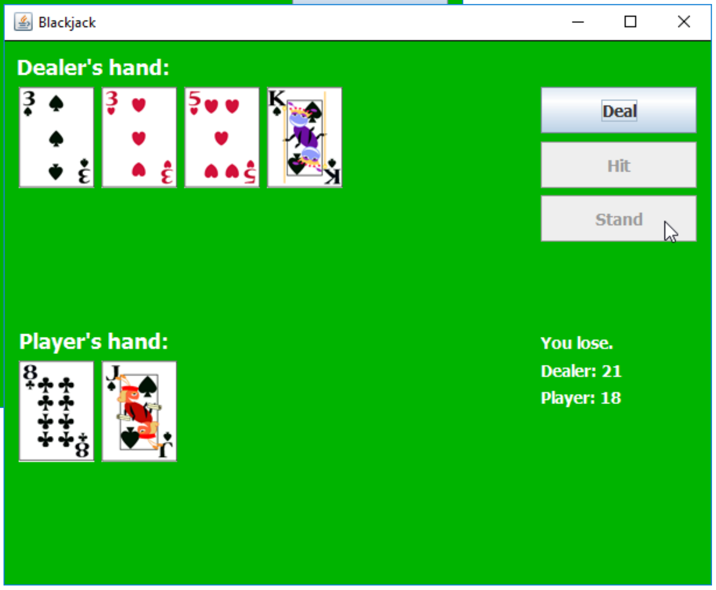
```java
         // update GUI to display new card
         displayCard(currentPlayerCard, card);
         ++currentPlayerCard;

         // determine new value of player's hand
         int total = blackjackProxy.getHandValue(playerCards);

         if (total > 21) // player busts
         {
            gameOver(GameStatus.LOSE);
         }
         else if (total == 21) // player cannot take any more cards
         {
            hitJButton.setEnabled(false);
            dealerPlay();
         }
      }

      // handles standJButton click
      private void standJButtonActionPerformed(
         java.awt.event.ActionEvent evt)
      {
         standJButton.setEnabled(false);
         hitJButton.setEnabled(false);
         dealJButton.setEnabled(true);
         dealerPlay();
      }

      // begins application execution
      public static void main(String args[])
      {
         java.awt.EventQueue.invokeLater(
            new Runnable()
            {
               public void run()
               {
                  new BlackjackGameJFrame().setVisible(true);
               }
            }
         );
      }

      // Variables declaration - do not modify
      private javax.swing.JButton dealJButton;
      private javax.swing.JLabel dealerCard10JLabel;
      private javax.swing.JLabel dealerCard11JLabel;
      private javax.swing.JLabel dealerCard1JLabel;
      private javax.swing.JLabel dealerCard2JLabel;
      private javax.swing.JLabel dealerCard3JLabel;
      private javax.swing.JLabel dealerCard4JLabel;
      private javax.swing.JLabel dealerCard5JLabel;
      private javax.swing.JLabel dealerCard6JLabel;
      private javax.swing.JLabel dealerCard7JLabel;
```

Fig. 32.18 | Blackjack game that uses the `Blackjack` web service. (Part 7 of 10.)

```
    private javax.swing.JLabel dealerCard8JLabel;
    private javax.swing.JLabel dealerCard9JLabel;
    private javax.swing.JLabel dealerJLabel;
    private javax.swing.JLabel dealerTotalJLabel;
    private javax.swing.JButton hitJButton;
    private javax.swing.JLabel playerCard10JLabel;
    private javax.swing.JLabel playerCard11JLabel;
    private javax.swing.JLabel playerCard1JLabel;
    private javax.swing.JLabel playerCard2JLabel;
    private javax.swing.JLabel playerCard3JLabel;
    private javax.swing.JLabel playerCard4JLabel;
    private javax.swing.JLabel playerCard5JLabel;
    private javax.swing.JLabel playerCard6JLabel;
    private javax.swing.JLabel playerCard7JLabel;
    private javax.swing.JLabel playerCard8JLabel;
    private javax.swing.JLabel playerCard9JLabel;
    private javax.swing.JLabel playerJLabel;
    private javax.swing.JLabel playerTotalJLabel;
    private javax.swing.JButton standJButton;
    private javax.swing.JLabel statusJLabel;
    // End of variables declaration
}
```

a) Dealer and player hands after the user clicks the **Deal** `JButton`.



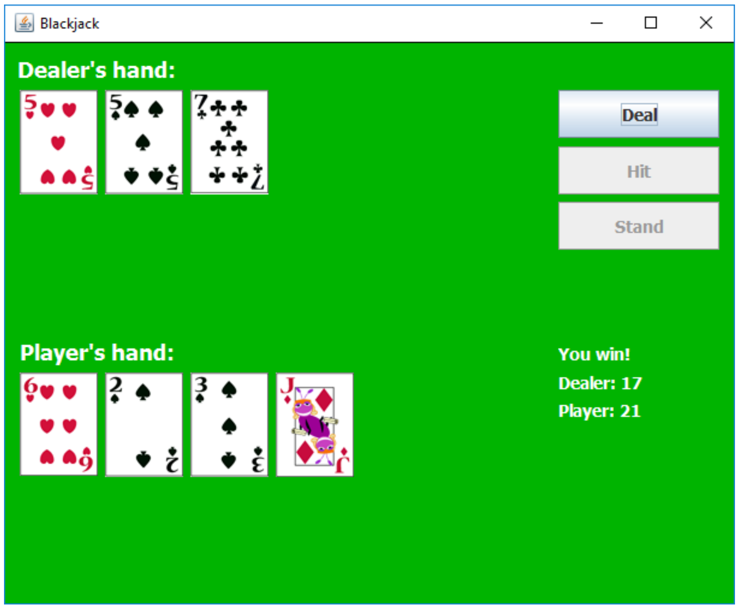


You lose.
Dealer: 21
Player: 18

b) Dealer and player hands after the user clicks **Stand**. In this case, the result is a loss for the player because the dealer has 21 and the player has 18.

Fig. 32.18 | Blackjack game that uses the `Blackjack` web service. (Part 8 of 10.)

c) Hands after the user clicks **Hit** twice and draws 21. In this case, the player wins with the higher hand.



d) Hands after the player is dealt blackjack.

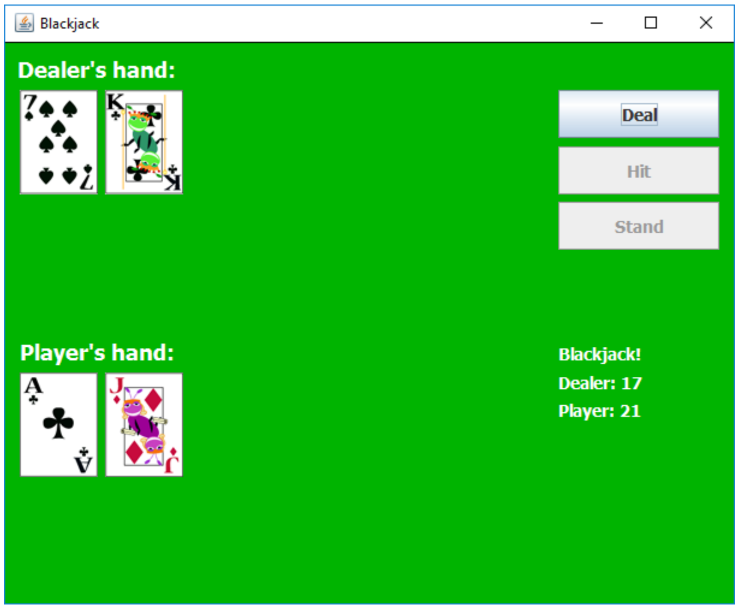


Fig. 32.18 | Blackjack game that uses the `Blackjack` web service. (Part 9 of 10.)

e) Hands after the dealer is dealt blackjack

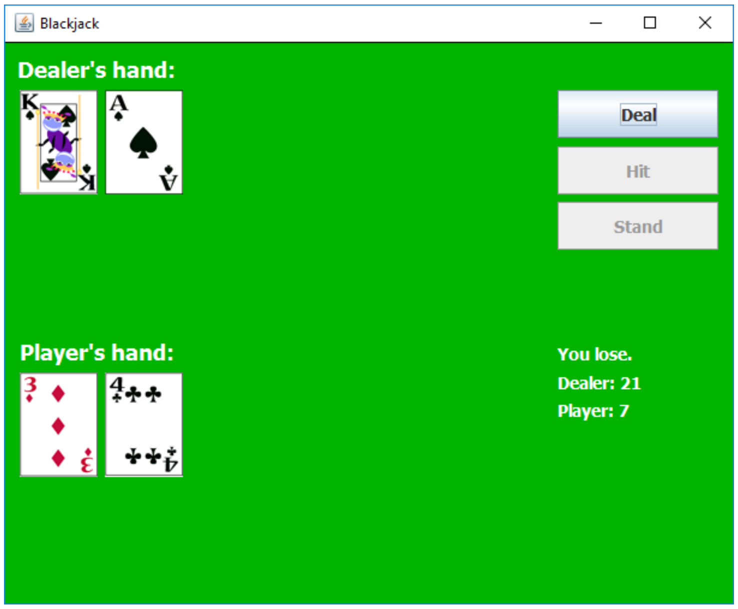


Fig. 32.18 | Blackjack game that uses the `Blackjack` web service. (Part 10 of 10.)

Configuring the Client for Session Tracking

When interacting with a JAX-WS web service that performs session tracking, the client application must indicate whether it wants to allow the web service to maintain session information. Lines 50–51 in the constructor perform this task. We first cast the service endpoint interface object to interface type `BindingProvider`. A **`BindingProvider`** enables the client to manipulate the request information that will be sent to the server. This information is stored in an object that implements interface **`RequestContext`**. The `BindingProvider` and `RequestContext` are part of the framework that is created by the IDE when you add a web service client to the application. Next, we invoke the `BindingProvider`'s **`getRequestContext` method** to obtain the `RequestContext` object. Then we call the `RequestContext`'s **`put` method** to set the property

```
    BindingProvider.SESSION_MAINTAIN_PROPERTY
```

to `true`. This enables the client side of the session-tracking mechanism, so that the web service knows which client is invoking the service's web methods.

Method `gameOver`

Method `gameOver` (lines 195–233) displays all the dealer's cards, shows the appropriate message in `statusJLabel` and displays the final point totals of both the dealer and the player. Method `gameOver` receives as an argument a member of the `GameStatus` `enum` (defined in lines 25–31). The `enum` constants represent whether the player tied, lost or won the game; its four members are `PUSH`, `LOSE`, `WIN` and `BLACKJACK`.

Method `dealJButtonActionPerformed`

When the player clicks the **Deal** `JButton`, method `dealJButtonActionPerformed` (lines 543–602) clears all of the `JLabel`s that display cards or game status information. Next, the deck is shuffled (line 559), and the player and dealer receive two cards each (lines 562–573). Lines 580–581 then total each hand. If the player and the dealer both obtain scores of 21, the program calls method `gameOver`, passing `GameStatus.PUSH` (line 586). If only the dealer has 21, the program passes `GameStatus.LOSE` to method `gameOver` (line 590). If only the player has 21 after the first two cards are dealt, the program passes `GameStatus.BLACKJACK` to method `gameOver` (line 594).

Method `hitJButtonActionPerformed`

If `dealJButtonActionPerformed` does not call `gameOver`, the player can take more cards by clicking the **Hit** `JButton`, which calls `hitJButtonActionPerformed` in lines 605–628. Each time a player clicks **Hit**, the program deals the player one more card (line 609) and displays it in the GUI (line 613). If the player exceeds 21, the game is over and the player loses (line 621). If the player has exactly 21, the player is not allowed to take any more cards (line 625), and method `dealerPlay` is called (line 626).

Method `dealerPlay`

Method `dealerPlay` (lines 86–139) displays the dealer's cards, then deals cards to the dealer until the dealer's hand has a value of 17 or more (lines 100–108). If the dealer exceeds 21, the player wins (line 116); otherwise, the values of the hands are compared, and `gameOver` is called with the appropriate argument (lines 122–133).

Method `standJButtonActionPerformed`

Clicking the **Stand** `JButton` indicates that a player does not want to be dealt another card. Method `standJButtonActionPerformed` (lines 631–638) disables the **Hit** and **Stand** buttons, enables the **Deal** button, then calls method `dealerPlay`.

Method `displayCard`

Method `displayCard` (lines 142–192) updates the GUI to display a newly dealt card. The method takes as arguments an integer index for the `JLabel` in the `ArrayList` that must have its image set and a `String` representing the card. An empty `String` indicates that we wish to display the card face down. If method `displayCard` receives a `String` that's not empty, the program extracts the face and suit from the `String` and uses this information to display the correct image. The `switch` statement (lines 167–181) converts the number representing the suit to an integer and assigns the appropriate character to variable `suitLetter` (`h` for hearts, `d` for diamonds, `c` for clubs and `s` for spades). The character in `suitLetter` is used to complete the image's filename (lines 184–186). *You must add the folder* `blackjack_images` *to your project so that lines 152–154 and 184–186 can access the images properly.* To do so, copy the folder `blackjack_images` from this chapter's examples folder and paste it into the project's `src\com\deitel\java\blackjackclient` folder.

32.10 Consuming a Database-Driven SOAP Web Service

Our prior examples accessed web services from desktop applications created in NetBeans. However, we can just as easily use them in web applications created with NetBeans. In fact, because web-based businesses are becoming increasingly popular, it's common for web applications to consume web services. In this section, we present an airline reservation web service that receives information regarding the type of seat a customer wishes to reserve and makes a reservation if such a seat is available. Later in the section, we present a web application that allows a customer to specify a reservation request, then uses the airline reservation web service to attempt to execute the request.

32.10.1 Creating the `Reservation` Database

Our web service uses a `reservation` database containing a single table named `Seats` to locate a seat matching a client's request. Review the steps presented in Section 31.2.1 for configuring a data source and the `addressbook` database. Then perform those steps for the `reservation` database used in this example. This chapter's examples directory contains the `Seats.sql` SQL script to create the `seats` table and populate it with sample data. The sample data is shown in Fig. 32.19.

number	location	class	taken
1	Aisle	Economy	0
2	Aisle	Economy	0
3	Aisle	First	0
4	Middle	Economy	0
5	Middle	Economy	0
6	Middle	First	0
7	Window	Economy	0
8	Window	Economy	0
9	Window	First	0
10	Window	First	0

Fig. 32.19 | Data from the `seats` table.

Creating the Reservation Web Service

You can now create a web service that uses the `reservation` database (Fig. 32.20). We used the `@DataSourceDefinition` annotation (lines 17–23) to create a data source named

```
    java:global/jdbc/reservation
```

for accessing the database.

```
1   // Fig. 32.20: Reservation.java
2   // Airline reservation web service.
3   package com.deitel.reservation;
4
```

Fig. 32.20 | Airline reservation web service. (Part 1 of 3.)

```java
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.annotation.Resource;
import javax.annotation.sql.DataSourceDefinition;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.sql.DataSource;

// define the data source
@DataSourceDefinition(
   name = "java:global/jdbc/reservation",
   className = "org.apache.derby.jdbc.ClientDataSource",
   url = "jdbc:derby://localhost:1527/reservation",
   databaseName = "reservation",
   user = "APP",
   password = "APP")

@WebService()
public class Reservation
{
   // allow the server to inject the DataSource
   @Resource(lookup="java:global/jdbc/reservation")
   DataSource dataSource;

   // a WebMethod that can reserve a seat
   @WebMethod(operationName = "reserve")
   public boolean reserve(@WebParam(name = "seatType") String seatType,
      @WebParam(name = "classType") String classType)
   {
      Connection connection = null;
      PreparedStatement lookupSeat = null;
      PreparedStatement reserveSeat = null;

      try
      {
         connection = DriverManager.getConnection(
            DATABASE_URL, USERNAME, PASSWORD);
         lookupSeat = connection.prepareStatement(
            "SELECT \"number\" FROM \"seats\" WHERE (\"taken\" = 0) " +
            "AND (\"location\" = ?) AND (\"class\" = ?)");
         lookupSeat.setString(1, seatType);
         lookupSeat.setString(2, classType);
         ResultSet resultSet = lookupSeat.executeQuery();

         // if requested seat is available, reserve it
         if (resultSet.next())
         {
            int seat = resultSet.getInt(1);
            reserveSeat = connection.prepareStatement(
               "UPDATE \"seats\" SET \"taken\"=1 WHERE \"number\"=?");
```

Fig. 32.20 | Airline reservation web service. (Part 2 of 3.)

```
58              reserveSeat.setInt(1, seat);
59              reserveSeat.executeUpdate();
60              return true;
61           }
62
63           return false;
64        }
65        catch (SQLException e)
66        {
67           e.printStackTrace();
68           return false;
69        }
70        catch (Exception e)
71        {
72           e.printStackTrace();
73           return false;
74        }
75        finally
76        {
77           try
78           {
79              lookupSeat.close();
80              reserveSeat.close();
81              connection.close();
82           }
83           catch (Exception e)
84           {
85              e.printStackTrace();
86              return false;
87           }
88        }
89     }
90  }
```

Fig. 32.20 | Airline reservation web service. (Part 3 of 3.)

The airline reservation web service has a single web method—`reserve` (lines 33–89)—which searches the `Seats` table to locate a seat matching a user's request. The method takes two arguments—a `String` representing the desired seat type (i.e., `"Window"`, `"Middle"` or `"Aisle"`) and a `String` representing the desired class type (i.e., `"Economy"` or `"First"`). If it finds an appropriate seat, method `reserve` updates the database to make the reservation and returns `true`; otherwise, no reservation is made, and the method returns `false`. The statements at lines 45–50 and lines 56–59 that query and update the database use objects of JDBC types `ResultSet` and `PreparedStatement`.

Software Engineering Observation 32.1

Using `PreparedStatement`s to create SQL statements is highly recommended to secure against so-called SQL injection attacks in which executable code is inserted into SQL code. The site `www.owasp.org/index.php/Preventing_SQL_Injection_in_Java` provides a summary of SQL injection attacks and ways to mitigate against them.

Our database contains four columns—the seat number (i.e., `1`–`10`), the seat type (i.e., `Window`, `Middle` or `Aisle`), the class type (i.e., `Economy` or `First`) and a column containing either `1` (true) or `0` (false) to indicate whether the seat is taken. Lines 45–50 retrieve the seat numbers of any available seats matching the requested seat and class type. This statement fills the `resultSet` with the results of the query

```
      SELECT number
      FROM seats
      WHERE (taken = 0) AND (type = type) AND (class = class)
```

The parameters *type* and *class* in the query are replaced with values of method `reserve`'s `seatType` and `classType` parameters.

If `resultSet` is not empty (i.e., at least one seat is available that matches the selected criteria), the condition in line 53 is `true` and the web service reserves the first matching seat number. Recall that `ResultSet` method `next` returns `true` if a nonempty row exists, and positions the cursor on that row. We obtain the seat number (line 55) by accessing `resultSet`'s first column (i.e., `resultSet.getInt(1)`—the first column in the row). Then lines 56–59 configure a `PreparedStatement` and execute the SQL:

```
      UPDATE seats
      SET taken = 1
      WHERE (number = number)
```

which marks the seat as taken in the database. The parameter *number* is replaced with the value of `seat`. Method `reserve` returns `true` (line 60) to indicate that the reservation was successful. If there are no matching seats, or if an exception occurred, method `reserve` returns `false` (lines 63, 68, 73 and 86) to indicate that no seats matched the user's request.

32.10.2 Creating a Web Application to Interact with the `Reservation` Service

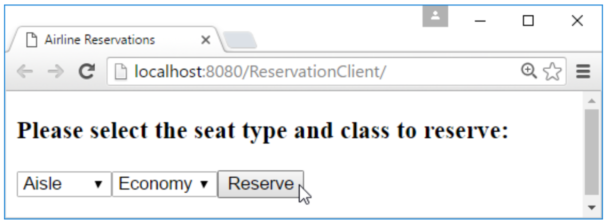
This section presents a `ReservationClient` JSF web application that consumes the `Reservation` web service. The application allows users to select `"Aisle"`, `"Middle"` or `"Window"` seats in `"Economy"` or `"First"` class, then submit their requests to the web service. If the database request is not successful, the application instructs the user to modify the request and try again. The application presented here was built using the techniques presented in Chapters 30–31. We assume that you've already read those chapters and thus know how to build a Facelets page and a corresponding JavaBean.

`index.xhtml`

`index.xhtml` (Fig. 32.21) defines two `h:selectOneMenu`s and an `h:commandButton`. The `h:selectOneMenu` at lines 16–20) displays all the seat types from which users can select. The one at lines 21–24) provides choices for the class type. The values of these are stored in the `seatType` and `classType` properties of the `reservationBean` (Fig. 32.22). Users click the **Reserve** button (lines 25–26) to submit requests after making selections from the `h:selectOneMenu`s. Clicking the button calls the `reservationBean`'s `reserveSeat` method. The page displays the result of each attempt to reserve a seat in line 28.

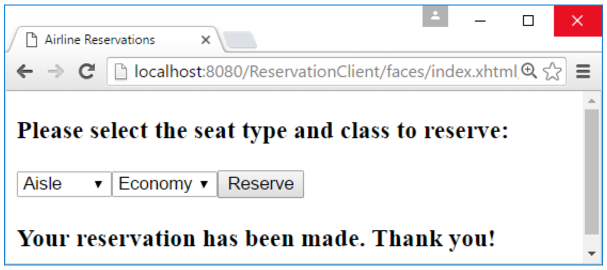
```
<?xml version='1.0' encoding='UTF-8' ?>

<!-- Fig. 31.21: index.xhtml -->
<!-- Facelets page that allows a user to select a seat -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:f="http://java.sun.com/jsf/core">
   <h:head>
      <title>Airline Reservations</title>
   </h:head>
   <h:body>
      <h:form>
         <h3>Please select the seat type and class to reserve:</h3>
         <h:selectOneMenu value="#{reservationBean.seatType}">
            <f:selectItem itemValue="Aisle" itemLabel="Aisle" />
            <f:selectItem itemValue="Middle" itemLabel="Middle" />
            <f:selectItem itemValue="Window" itemLabel="Window" />
         </h:selectOneMenu>
         <h:selectOneMenu value="#{reservationBean.classType}">
            <f:selectItem itemValue="Economy" itemLabel="Economy" />
            <f:selectItem itemValue="First" itemLabel="First" />
         </h:selectOneMenu>
         <h:commandButton value="Reserve"
            action="#{reservationBean.reserveSeat}"/>
      </h:form>
      <h3>#{reservationBean.result}</h3>
   </h:body>
</html>
```

a) Selecting a seat



Please select the seat type and class to reserve:

b) Seat reserved successfully

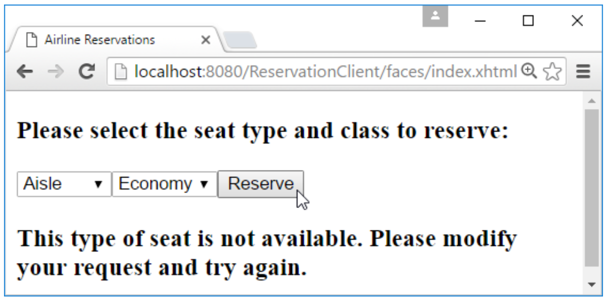


Please select the seat type and class to reserve:

Your reservation has been made. Thank you!

Fig. 32.21 | Facelets page that allows a user to select a seat. (Part 1 of 2.)

c) Attempting to reserve an aisle economy seat when no more are available—because no seats match the requested seat type and class, the user is asked to try again



Please select the seat type and class to reserve:

This type of seat is not available. Please modify your request and try again.

Fig. 32.21 | Facelets page that allows a user to select a seat. (Part 2 of 2.)

`ReservationBean.java`

Class `ReservationBean` (Fig. 32.22) defines the `seatType`, `classType` and `result` properties and the `reserveSeat` method that are used in the `index.xhtml` page. When the user clicks the **Reserve** button in `index.xhtml`, method `reserveSeat` (lines 59–76) executes. Lines 63–64 use the service endpoint interface object (created in lines 24–25) to invoke the web service's `reserve` method, passing the selected seat type and class type as arguments. If `reserve` returns `true`, line 67 sets `result` to a message thanking the user for making a reservation; otherwise, lines 69–70 set `result` to a message notifying the user that the requested seat type is not available and instructing the user to try again.

```java
1   // Fig. 31.22: ReservationBean.java
2   // Bean for seat reservation client.
3   package reservationclient;
4
5   import com.deitel.reservation.Reservation;
6   import com.deitel.reservation.ReservationService;
7   import java.io.Serializable;
8   import javax.faces.bean.ManagedBean;
9
10  @Named("reservationBean")
11  @javax.faces.view.ViewScoped
12  public class ReservationBean implements Serializable
13  {
14     // references the service endpoint interface object (i.e., the proxy)
15     private Reservation reservationServiceProxy; // reference to proxy
16     private String seatType; // type of seat to reserve
17     private String classType; // class of seat to reserve
18     private String result; // result of reservation attempt
19
20     // no-argument constructor
21     public ReservationBean()
22     {
23        // get service endpoint interface
24        ReservationService reservationService = new ReservationService();
25        reservationServiceProxy = reservationService.getReservationPort();
26     }
```

Fig. 32.22 | Page bean for seat reservation client. (Part 1 of 2.)

```java

   // return classType
   public String getClassType()
   {
      return classType;
   }

   // set classType
   public void setClassType(String classType)
   {
      this.classType = classType;
   }

   // return seatType
   public String getSeatType()
   {
      return seatType;
   }

   // set seatType
   public void setSeatType(String seatType)
   {
      this.seatType = seatType;
   }

   // return result
   public String getResult()
   {
      return result;
   }

   // invoke the web service when the user clicks Reserve button
   public void reserveSeat()
   {
      try
      {
         boolean reserved = reservationServiceProxy.reserve(
            getSeatType(), getClassType());

         if (reserved)
            result = "Your reservation has been made. Thank you!";
         else
            result = "This type of seat is not available. " +
               "Please modify your request and try again.";
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }
}
```

Fig. 32.22 | Page bean for seat reservation client. (Part 2 of 2.)

32.11 Equation Generator: Returning User-Defined Types

Most of the web services we've demonstrated received and returned primitive-type instances. It's also possible to process instances of class types in a web service. These types can be passed to or returned from web service methods.

This section presents a RESTful `EquationGenerator` web service that generates random arithmetic equations of type `Equation`. The client is a math-tutoring application that accepts information about the mathematical question that the user wishes to attempt (addition, subtraction or multiplication) and the skill level of the user (1 specifies equations using numbers from 1 through 9, 2 specifies equations involving numbers from 10 through 99, and 3 specifies equations containing numbers from 100 through 999). Each web service then generates an equation consisting of random numbers in the proper range. The client application receives the `Equation` and displays the sample question to the user. We present the web service and client twice—once using XML and once using JSON.

Defining Class `Equation`

We define class `Equation` in Fig. 32.23. All the programs in this section have a copy of this class in their corresponding package. Except for the package name, the class is identical in each project, so we show it only once. Like the `TextMessage` class used earlier, the server-side and client-side copies of class `Equation` are unrelated to each other. The only requirement for *serialization* and *deserialization* to work with the JAXB and Gson classes is that class `Equation` must have the same `public` properties on both the server and the client. Such properties can be `public` instance variables or `private` instance variables that have corresponding, appropriately named *set* and *get* methods.

```java
// Fig. 32.23: Equation.java
// Equation class that contains information about an equation.
package com.deitel.equationgeneratorxml;

public class Equation
{
   private int leftOperand;
   private int rightOperand;
   private int result;
   private String operationType;

   // required no-argument constructor
   public Equation()
   {
      this(0, 0, "add");
   }

   // constructor that receives the operands and operation type
   public Equation(int leftValue, int rightValue, String type)
   {
      leftOperand = leftValue;
      rightOperand = rightValue;

```

Fig. 32.23 | `Equation` class that contains information about an equation. (Part 1 of 3.)

```
         // determine result
         if (type.equals("add")) // addition
         {
            result = leftOperand + rightOperand;
            operationType = "+";
         }
         else if (type.equals("subtract")) // subtraction
         {
            result = leftOperand - rightOperand;
            operationType = "-";
         }
         else // multiplication
         {
            result = leftOperand * rightOperand;
            operationType = "*";
         }
      }

      // gets the leftOperand
      public int getLeftOperand()
      {
         return leftOperand;
      }

      // required setter
      public void setLeftOperand(int value)
      {
         leftOperand = value;
      }

      // gets the rightOperand
      public int getRightOperand()
      {
         return rightOperand;
      }

      // required setter
      public void setRightOperand(int value)
      {
         rightOperand = value;
      }

      // gets the resultValue
      public int getResult()
      {
         return result;
      }

      // required setter
      public void setResult(int value)
      {
         result = value;
      }
```

Fig. 32.23 | `Equation` class that contains information about an equation. (Part 2 of 3.)

```
77
78     // gets the operationType
79     public String getOperationType()
80     {
81        return operationType;
82     }
83
84     // required setter
85     public void setOperationType(String value)
86     {
87        operationType = value;
88     }
89
90     // returns the left hand side of the equation as a String
91     public String getLeftHandSide()
92     {
93        return leftOperand + " " + operationType + " " + rightOperand;
94     }
95
96     // returns the right hand side of the equation as a String
97     public String getRightHandSide()
98     {
99        return "" + result;
100    }
101
102    // returns a String representation of an Equation
103    public String toString()
104    {
105       return getLeftHandSide() + " = " + getRightHandSide();
106    }
107 }
```

Fig. 32.23 | `Equation` class that contains information about an equation. (Part 3 of 3.)

Lines 19–40 define a constructor that takes two `int`s representing the left and right operands, and a `String` representing the arithmetic operation. The constructor stores this information, then calculates the result. The parameterless constructor (lines 13–16) calls the three-argument constructor (lines 19–40) and passes default values.

Class `Equation` defines *get* and *set* methods for example variables `leftOperand` (lines 43–52), `rightOperand` (lines 55–64), `result` (line 67–76) and `operationType` (lines 79–88). It also provides *get* methods for the left-hand and right-hand sides of the equation and a `toString` method that returns the entire equation as a `String`. An instance variable can be serialized only if it has both a *get* and a *set* method. Because the different sides of the equation and the result of `toString` can be generated from the other instance variables, there's no need to send them across the wire. The client in this case study does not use the `getRightHandSide` method, but we included it in case future clients choose to use it.

32.11.1 Creating the `EquationGeneratorXML` Web Service

Figure 32.24 presents the `EquationGeneratorXML` web service's class for creating randomly generated `Equation`s. Method `getXml` (lines 19–38) takes two parameters—a `String` representing the mathematical operation (`"add"`, `"subtract"` or `"multiply"`) and an `int` representing the difficulty level. JAX-RS automatically converts the arguments to the correct type and will return a “not found” error to the client if the argument cannot be converted from a `String` to the destination type. Supported types for conversion include integer types, floating-point types, `boolean` and the corresponding type-wrapper classes.

```java
 1  // Fig. 32.24: EquationGeneratorXMLResource.java
 2  // RESTful equation generator that returns XML.
 3  package com.deitel.equationgeneratorxml;
 4
 5  import java.io.StringWriter;
 6  import java.security.SecureRandom;
 7  import javax.ws.rs.PathParam;
 8  import javax.ws.rs.Path;
 9  import javax.ws.rs.GET;
10  import javax.ws.rs.Produces;
11  import javax.xml.bind.JAXB; // utility class for common JAXB operations
12
13  @Path("equation")
14  public class EquationGeneratorXMLResource
15  {
16     private static SecureRandom randomObject = new SecureRandom();
17
18     // retrieve an equation formatted as XML
19     @GET
20     @Path("{operation}/{level}")
21     @Produces("application/xml")
22     public String getXml(@PathParam("operation") String operation,
23        @PathParam("level") int level)
24     {
25        // compute minimum and maximum values for the numbers
26        int minimum = (int) Math.pow(10, level - 1);
27        int maximum = (int) Math.pow(10, level);
28
29        // create the numbers on the left-hand side of the equation
30        int first = randomObject.nextInt(maximum - minimum) + minimum;
31        int second = randomObject.nextInt(maximum - minimum) + minimum;
32
33        // create Equation object and marshal it into XML
34        Equation equation = new Equation(first, second, operation);
35        StringWriter writer = new StringWriter(); // XML output here
36        JAXB.marshal(equation, writer); // write Equation to StringWriter
37        return writer.toString(); // return XML string
38     }
39  }
```

Fig. 32.24 | RESTful equation generator that returns XML.

The `getXml` method first determines the minimum (inclusive) and maximum (exclusive) values for the numbers in the equation it will return (lines 26–27). It then uses a `Random` object (created at line 16) to generate two random numbers in that range (lines 30–31). Line 34 creates an `Equation` object, passing these two numbers and the requested operation to the constructor. The `getXml` method then uses JAXB to convert the `Equation` object to XML (line 36), which is output to the `StringWriter` created on line 35. Finally, it retrieves the data that was written to the `StringWriter` and returns it to the client. For example, if you invoke the web service with

```
http://localhost:8080/EquationGeneratorXML/webresources/equation/
   add/1
```

the response will have the format

```
<equation>
   <leftOperand>6</leftOperand>
   <operationType>+</operationType>
   <result>11</result>
   <rightOperand>5</rightOperand>
</equation>
```

[*Note:* We'll reimplement this web service with JSON in Section 32.11.3.]

32.11.2 Consuming the EquationGeneratorXML Web Service

The `EquationGeneratorXMLClient` application (Fig. 32.25) retrieves an XML-formatted `Equation` object from the `EquationGeneratorXML` web service. The application then displays the `Equation`'s left-hand side and waits for user to submit an answer.

```java
// Fig. 32.25: EquationGeneratorXMLClientJFrame.java
// Math-tutoring program using REST and XML to generate equations.
package com.deitel.equationgeneratorxmlclient;

import javax.swing.JOptionPane;
import javax.xml.bind.JAXB; // utility class for common JAXB operations

public class EquationGeneratorXMLClientJFrame extends javax.swing.JFrame
{
   private String operation = "add"; // operation user is tested on
   private int difficulty = 1; // 1, 2, or 3 digits in each number
   private int answer; // correct answer to the question

   // no-argument constructor
   public EquationGeneratorXMLClientJFrame()
   {
      initComponents();
   }

   // The initComponents method is autogenerated by NetBeans and is called
   // from the constructor to initialize the GUI. This method is not shown
   // here to save space. Open EquationGeneratorXMLClientJFrame.java in
   // this example's folder to view the complete generated code.

   // determine if the user answered correctly
   private void checkAnswerJButtonActionPerformed(
      java.awt.event.ActionEvent evt)
   {
      if (answerJTextField.getText().equals(""))
      {
```

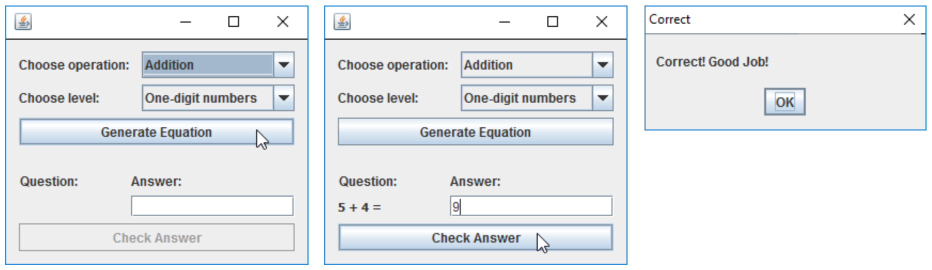
Fig. 32.25 | Math-tutoring program using REST and XML to generate equations. (Part 1 of 3.)

```
            JOptionPane.showMessageDialog(
               this, "Please enter your answer.");
         }

         int userAnswer = Integer.parseInt(answerJTextField.getText());

         if (userAnswer == answer)
         {
            equationJLabel.setText(""); // clear label
            answerJTextField.setText(""); // clear text field
            checkAnswerJButton.setEnabled(false);
            JOptionPane.showMessageDialog(this, "Correct! Good Job!",
               "Correct", JOptionPane.PLAIN_MESSAGE);
         }
         else
         {
            JOptionPane.showMessageDialog(this, "Incorrect. Try again.",
               "Incorrect", JOptionPane.PLAIN_MESSAGE);
         }
      }

      // retrieve equation from web service and display left side to user
      private void generateJButtonActionPerformed(
         java.awt.event.ActionEvent evt)
      {
         try
         {
            String url = String.format("http://localhost:8080/" +
               "EquationGeneratorXML/webresources/equation/%s/%d",
               operation, difficulty);

            // convert XML back to an Equation object
            Equation equation = JAXB.unmarshal(url, Equation.class);

            answer = equation.getResult();
            equationJLabel.setText(equation.getLeftHandSide() + " =");
            checkAnswerJButton.setEnabled(true);
         }
         catch (Exception exception)
         {
            exception.printStackTrace();
         }
      }

      // obtains the mathematical operation selected by the user
      private void operationJComboBoxItemStateChanged(
         java.awt.event.ItemEvent evt)
      {
         String item = (String) operationJComboBox.getSelectedItem();

         if (item.equals("Addition"))
            operation = "add"; // user selected addition
```

Fig. 32.25 | Math-tutoring program using REST and XML to generate equations. (Part 2 of 3.)

```java
      else if (item.equals("Subtraction"))
         operation = "subtract"; // user selected subtraction
      else
         operation = "multiply"; // user selected multiplication
   }

   // obtains the difficulty level selected by the user
   private void levelJComboBoxItemStateChanged(
      java.awt.event.ItemEvent evt)
   {
      // indices start at 0, so add 1 to get the difficulty level
      difficulty = levelJComboBox.getSelectedIndex() + 1;
   }

   // main method begins execution
   public static void main(String args[])
   {
      java.awt.EventQueue.invokeLater(
         new Runnable()
         {
            public void run()
            {
               new EquationGeneratorXMLClientJFrame().setVisible(true);
            }
         }
      );
   }

   // Variables declaration - do not modify
   private javax.swing.JLabel answerJLabel;
   private javax.swing.JTextField answerJTextField;
   private javax.swing.JButton checkAnswerJButton;
   private javax.swing.JLabel equationJLabel;
   private javax.swing.JButton generateJButton;
   private javax.swing.JComboBox levelJComboBox;
   private javax.swing.JLabel levelJLabel;
   private javax.swing.JComboBox operationJComboBox;
   private javax.swing.JLabel operationJLabel;
   private javax.swing.JLabel questionJLabel;
   // End of variables declaration
}
```

a) Generating a simple equation. b) Sumbitting the answer. c) Dialog indicating correct answer.



a)
Choose operation: Addition
Choose level: One-digit numbers
Generate Equation
Question: Answer:
Check Answer

b)
Choose operation: Addition
Choose level: One-digit numbers
Generate Equation
Question: Answer:
5 + 4 = 9
Check Answer

c)
Correct
Correct! Good Job!
OK

Fig. 32.25 | Math-tutoring program using REST and XML to generate equations. (Part 3 of 3.)

The default setting for the difficulty level is `1`, but the user can change this by choosing a level from the **Choose level** `JComboBox`. Changing the selected value invokes the `levelJComboBoxItemStateChanged` event handler (lines 212–217), which sets the `difficulty` instance variable to the level selected by the user. Although the default setting for the question type is **Addition**, the user also can change this by choosing from the **Choose operation** `JComboBox`. This invokes the `operationJComboBoxItemStateChanged` event handler in lines 198–209, which assigns to instance variable `operation` the `String` corresponding to the user's selection.

The event handler for `generateJButton` (lines 175–195) constructs the URL to invoke the web service, then passes this URL to the `unmarshal` method, along with an instance of `Class<Equation>`, so that JAXB can convert the XML into an `Equation` object (line 185). Once the XML has been converted back into an `Equation`, lines 183–184 retrieve the correct answer and display the left-hand side of the equation. The **Check Answer** button is then enabled (line 189), and the user must solve the problem and enter the answer.

When the user enters a value and clicks **Check Answer**, the `checkAnswerJButtonActionPerformed` event handler (lines 148–172) retrieves the user's answer from the dialog box (line 157) and compares it to the correct answer that was stored earlier (line 159). If they match, lines 161–165 reset the GUI elements so the user can generate another equation and tell the user that the answer was correct. If they do not match, a message box asking the user to try again is displayed (lines 169–170).

32.11.3 Creating the `EquationGeneratorJSON` Web Service

As you saw in Section 32.8, RESTful web services can return data formatted as JSON as well. Figure 32.26 is a reimplementation of the `EquationGeneratorXML` service that returns an `Equation` in JSON format. The logic implemented here is the same as the XML version except for the last line (line 34), which uses Gson to convert the `Equation` object into JSON instead of using JAXB to convert it into XML. The `@Produces` annotation (line 20) has also changed to reflect the JSON data format.

```java
 1  // Fig. 32.26: EquationGeneratorJSONResource.java
 2  // RESTful equation generator that returns JSON.
 3  package com.deitel.equationgeneratorjson;
 4
 5  import com.google.gson.Gson; // converts POJO to JSON and back again
 6  import java.util.Random;
 7  import javax.ws.rs.GET;
 8  import javax.ws.rs.Path;
 9  import javax.ws.rs.PathParam;
10  import javax.ws.rs.Produces;
11
12  @Path("equation")
13  public class EquationGeneratorJSONResource
14  {
15     static Random randomObject = new Random(); // random number generator
16
```

Fig. 32.26 | RESTful equation generator that returns JSON. (Part 1 of 2.)

```java
   // retrieve an equation formatted as JSON
   @GET
   @Path("{operation}/{level}")
   @Produces("application/json")
   public String getJson(@PathParam("operation") String operation,
      @PathParam("level") int level)
   {
      // compute minimum and maximum values for the numbers
      int minimum = (int) Math.pow(10, level - 1);
      int maximum = (int) Math.pow(10, level);

      // create the numbers on the left-hand side of the equation
      int first = randomObject.nextInt(maximum - minimum) + minimum;
      int second = randomObject.nextInt(maximum - minimum) + minimum;

      // create Equation object and return result
      Equation equation = new Equation(first, second, operation);
      return new Gson().toJson(equation); // convert to JSON and return
   }
}
```

Fig. 32.26 | RESTful equation generator that returns JSON. (Part 2 of 2.)

32.11.4 Consuming the EquationGeneratorJSON Web Service

The program in Fig. 32.27 consumes the `EquationGeneratorJSON` service and performs the same function as `EquationGeneratorXMLClient`—the only difference is in how the `Equation` object is retrieved from the web service. Lines 181–183 construct the URL that is used to invoke the `EquationGeneratorJSON` service. As in the `WelcomeRESTJSONClient` example, we use the `URL` class and an `InputStreamReader` to invoke the web service and read the response (lines 186–187). The retrieved JSON is *deserialized* using Gson (line 191) and converted back into an `Equation` object. As before, we use the `getResult` method (line 194) of the deserialized object to obtain the answer and the `getLeftHandSide` method (line 195) to display the left side of the equation.

```java
 1  // Fig. 32.27: EquationGeneratorJSONClientJFrame.java
 2  // Math-tutoring program using REST and JSON to generate equations.
 3  package com.deitel.equationgeneratorjsonclient;
 4
 5  import com.google.gson.Gson; // converts POJO to JSON and back again
 6  import java.io.InputStreamReader;
 7  import java.net.URL;
 8  import javax.swing.JOptionPane;
 9
10  public class EquationGeneratorJSONClientJFrame extends javax.swing.JFrame
11  {
12     private String operation = "add"; // operation user is tested on
13     private int difficulty = 1; // 1, 2, or 3 digits in each number
14     private int answer; // correct answer to the question
15
```

Fig. 32.27 | Math-tutoring program using REST and JSON to generate equations. (Part 1 of 4.)

```
   // no-argument constructor
   public EquationGeneratorJSONClientJFrame()
   {
      initComponents();
   }

   // The initComponents method is autogenerated by NetBeans and is called
   // from the constructor to initialize the GUI. This method is not shown
   // here to save space. Open EquationGeneratorJSONClientJFrame.java in
   // this example's folder to view the complete generated code.

   // determine if the user answered correctly
   private void checkAnswerJButtonActionPerformed(
      java.awt.event.ActionEvent evt)
   {
      if (answerJTextField.getText().equals(""))
      {
         JOptionPane.showMessageDialog(
            this, "Please enter your answer.");
      }

      int userAnswer = Integer.parseInt(answerJTextField.getText());

      if (userAnswer == answer)
      {
         equationJLabel.setText(""); // clear label
         answerJTextField.setText(""); // clear text field
         checkAnswerJButton.setEnabled(false);
         JOptionPane.showMessageDialog(this, "Correct! Good Job!",
            "Correct", JOptionPane.PLAIN_MESSAGE);
      }
      else
      {
         JOptionPane.showMessageDialog(this, "Incorrect. Try again.",
            "Incorrect", JOptionPane.PLAIN_MESSAGE);
      }
   }

   // retrieve equation from web service and display left side to user
   private void generateJButtonActionPerformed(
      java.awt.event.ActionEvent evt)
   {
      try
      {
         // URL of the EquationGeneratorJSON service, with parameters
         String url = String.format("http://localhost:8080/" +
            "EquationGeneratorJSON/webresources/equation/%s/%d",
            operation, difficulty);

         // open URL and create a Reader to read the data
         InputStreamReader reader =
            new InputStreamReader(new URL(url).openStream());

```

Fig. 32.27 | Math-tutoring program using REST and JSON to generate equations. (Part 2 of 4.)

```
         // convert the JSON back into an Equation object
         Equation equation =
            new Gson().fromJson(reader, Equation.class);

         // update the internal state and GUI to reflect the equation
         answer = equation.getResult();
         equationJLabel.setText(equation.getLeftHandSide() + " =");
         checkAnswerJButton.setEnabled(true);
      }
      catch (Exception exception)
      {
         exception.printStackTrace();
      }
   }

   // obtains the mathematical operation selected by the user
   private void operationJComboBoxItemStateChanged(
      java.awt.event.ItemEvent evt)
   {
      String item = (String) operationJComboBox.getSelectedItem();

      if (item.equals("Addition"))
         operation = "add"; // user selected addition
      else if (item.equals("Subtraction"))
         operation = "subtract"; // user selected subtraction
      else
         operation = "multiply"; // user selected multiplication
   }

   // obtains the difficulty level selected by the user
   private void levelJComboBoxItemStateChanged(
      java.awt.event.ItemEvent evt)
   {
      // indices start at 0, so add 1 to get the difficulty level
      difficulty = levelJComboBox.getSelectedIndex() + 1;
   }

   // main method begins execution
   public static void main(String args[])
   {
      java.awt.EventQueue.invokeLater(
         new Runnable()
         {
            public void run()
            {
               new EquationGeneratorJSONClientJFrame().setVisible(true);
            }
         }
      );
   }

   // Variables declaration - do not modify
   private javax.swing.JLabel answerJLabel;
```

Fig. 32.27 | Math-tutoring program using REST and JSON to generate equations. (Part 3 of 4.)

```
   private javax.swing.JTextField answerJTextField;
   private javax.swing.JButton checkAnswerJButton;
   private javax.swing.JLabel equationJLabel;
   private javax.swing.JButton generateJButton;
   private javax.swing.JComboBox levelJComboBox;
   private javax.swing.JLabel levelJLabel;
   private javax.swing.JComboBox operationJComboBox;
   private javax.swing.JLabel operationJLabel;
   private javax.swing.JLabel questionJLabel;
   // End of variables declaration
}
```

Fig. 32.27 | Math-tutoring program using REST and JSON to generate equations. (Part 4 of 4.)

32.12 Wrap-Up

This chapter introduced web services—a set of technologies for building distributed systems in which system components communicate with one another over networks. In particular, we presented JAX-WS SOAP-based web services and JAX-RS REST-based web services. You learned that a web service is a class that allows client software to call the web service's methods remotely via common data formats and protocols, such as XML, JSON, HTTP, SOAP and REST. We also benefits of distributed computing with web services.

We explained how NetBeans and the JAX-WS and JAX-RS APIs facilitate publishing and consuming web services. You learned how to define web services and methods using both SOAP protocol and REST architecture, and how to return data in both XML and JSON formats. You consumed SOAP-based web services using proxy classes to call the web service's methods. You also consumed REST-based web services by using class `URL` to invoke the services and open `InputStreams` from which the clients could read the services' responses. You learned how to define web services and web methods, as well as how to consume them both from Java desktop applications and from web applications. After explaining the mechanics of web services through our `Welcome` examples, we demonstrated more sophisticated web services that use session tracking, database access and user-defined types. We also explained XML and JSON serialization and showed how to retrieve objects of user-defined types from web services.

Summary

Section 32.1 Introduction

- A web service (p. 2) is a software component stored on one computer that can be accessed by an application (or other software component) on another computer over a network.
- Web services communicate using such technologies as XML, JSON and HTTP.
- JAX-WS (p. 2) is based on the Simple Object Access Protocol (SOAP; p. 2)—an XML-based protocol that allows web services and clients to communicate.
- JAX-RS (p. 2) uses Representational State Transfer (REST; p. 2)—a network architecture that uses the web's traditional request/response mechanisms such as `GET` and `POST` requests.
- Web services enable businesses to conduct transactions via standardized, widely available web services rather than relying on proprietary applications.

- Web services are platform and language independent, so companies can collaborate via web services without hardware, software and communications compatibility issues.
- NetBeans is one of the many tools that enable you to publish and/or consume web services.

Section 32.2 Web Service Basics

- The machine on which a web service resides is referred to as a web service host.
- A client application that accesses the web service sends a method call over a network to the web service host, which processes the call and returns a response over the network to the application.
- In Java, a web service is implemented as a class. The class that represents the web service resides on a server—it's not part of the client application.
- Making a web service available to receive client requests is known as publishing a web service (p. 4); using a web service from a client application is known as consuming a web service (p. 4).

Section 32.3 Simple Object Access Protocol (SOAP)

- SOAP is a platform-independent protocol that uses XML to make remote procedure calls, typically over HTTP. Each request and response is packaged in a SOAP message (p. 4)—an XML message containing the information that a web service requires to process the message.
- SOAP messages are written in XML so that they're computer readable, human readable and platform independent.
- SOAP supports an extensive set of types—the primitive types, as well as `DateTime`, `XmlNode` and others. SOAP can also transmit arrays of these types.
- When a program invokes a method of a SOAP web service, the request and all relevant information are packaged in a SOAP message, enclosed in a SOAP envelope (p. 4) and sent to the server on which the web service resides.
- When a web service receives a SOAP message, it parses the XML representing the message, then processes the message's contents. The message specifies the method that the client wishes to execute and the arguments the client passed to that method.
- After a web service parses a SOAP message, it calls the appropriate method with the specified arguments (if any) and sends the response back to the client in another SOAP message. The client parses the response to retrieve the method's result.

Section 32.4 Representational State Transfer (REST)

- Representational State Transfer (REST) refers to an architectural style for implementing web services. Such web services are often called RESTful web services (p. 4). Though REST itself is not a standard, RESTful web services are implemented using web standards.
- Each operation in a RESTful web service is identified by a unique URL.
- REST can return data in many formats, including XML and JSON.

Section 32.5 JavaScript Object Notation (JSON)

- JavaScript Object Notation (JSON; p. 5) is an alternative to XML for representing data.
- JSON is a text-based data-interchange format used to represent objects in JavaScript as collections of name/value pairs represented as `String`s.
- JSON is a simple format that makes objects easy to read, create and parse and allows programs to transmit data efficiently across the Internet, because it's much less verbose than XML.
- Each value in a JSON array can be a string, a number, a JSON object, `true`, `false` or `null`.

Section 32.6.1 Creating a Web Application Project and Adding a Web Service Class in NetBeans

- When you create a web service in NetBeans, you focus on the web service's logic and let the IDE handle the web service's infrastructure.
- To create a web service in NetBeans, you first create a **Web Application** project (p. 5).

Section 31.6.2 Defining the `WelcomeSOAP` Web Service in NetBeans

- By default, each new web service class created with the JAX-WS APIs is a POJO (plain old Java object)—you do not need to extend a class or implement an interface to create a web service.
- When you deploy a web application containing a JAX-WS web service, the server creates the server-side artifacts that support the web service.
- The `@WebService` annotation (p. 7) indicates that a class represents a web service. The optional `name` attribute (p. 7) specifies the service endpoint interface (SEI; p. 7) class's name. The optional `serviceName` attribute (p. 7) specifies the name of the class that the client uses to obtain an SEI object.
- Methods that are tagged with the `@WebMethod` annotation (p. 7) can be called remotely.
- The `@WebMethod` annotation's optional `operationName` attribute (p. 7) specifies the method name that is exposed to the web service's clients.
- Web method parameters are annotated with the `@WebParam` annotation (p. 8). The optional `name` attribute (p. 8) indicates the parameter name that is exposed to the web service's clients.

Section 31.6.3 Publishing the `WelcomeSOAP` Web Service from NetBeans

- NetBeans handles all the details of building and deploying a web service for you. This includes creating the framework required to support the web service.

Section 31.6.4 Testing the `WelcomeSOAP` Web Service with GlassFish Application Server's `Tester` Web Page

- GlassFish can dynamically create a web page for testing a web service's methods from a web browser. To open the test page, expand the project's **Web Services** node in the NetBeans **Projects** tab, then right click the web service class name and select **Test Web Service**.
- A client can access a web service only when the application server is running. If NetBeans launches the application server for you, the server will shut down when you close NetBeans. To keep the application server up and running, you can launch it independently of NetBeans.

Section 32.6.5 Describing a Web Service with the Web Service Description Language (WSDL)

- To consume a web service, a client must know where to find it and must be provided with the web service's description.
- JAX-WS uses the Web Service Description Language (WSDL; p. 11)—a standard XML vocabulary for describing web services in a platform-independent manner.
- The server generates a web service's WSDL dynamically for you, and client tools can parse the WSDL to help create the client-side proxy class that a client uses to access the web service.

Section 31.6.6 Creating a Client to Consume the `WelcomeSOAP` Web Service

- A web service reference (p. 12) defines the service endpoint interface class so that a client can access the a service.

- An application that consumes a SOAP-based web service invokes methods on a service endpoint interface (SEI) object that interact with the web service on the client's behalf.
- The service endpoint interface object handles the details of passing method arguments to and receiving return values from the web service. This communication can occur over a local network, over the Internet or even with a web service on the same computer.
- NetBeans creates these service endpoint interface classes for you.
- When you add the web service reference, the IDE creates and compiles the client-side artifacts—the framework of Java code that supports the client-side service endpoint interface class. The service endpoint interface class uses the rest of the artifacts to interact with the web service.
- A web service reference is added by giving NetBeans the URL of the web service's WSDL file.

Section 31.6.7 Consuming the `WelcomeSOAP` Web Service

- To consume a JAX-WS web service, you must obtain an SEI object. You then invoke the web service's methods through the SEI object.

Section 32.7.1 Creating a REST-Based XML Web Service

- The **RESTful Web Services** plug-in for NetBeans provides various templates for creating RESTful web services, including ones that can interact with databases on the client's behalf.
- The `@Path` annotation (p. 17) on a JAX-RS web service class indicates the URI for accessing the web service. This is appended to the web application project's URL to invoke the service. Methods of the class can also use the `@Path` annotation.
- Parts of the path specified in curly braces indicate parameters—they're placeholders for arguments that are passed to the web service as part of the path. The base path for the service is the project's `resources` directory.
- Arguments in a URL can be used as arguments to a web service method. To do so, you bind the parameters specified in the `@Path` specification to parameters of a web service method with the `@PathParam` annotation (p. 19). When the request is received, the server passes the argument(s) in the URL to the appropriate parameter(s) in the web service method.
- The `@GET` annotation (p. 19) denotes that a method is accessed via an HTTP `GET` request. Similar annotations exist for HTTP `PUT`, `POST`, `DELETE` and `HEAD` requests.
- The `@Produces` annotation (p. 19) denotes the content type returned to the client. It's possible to have multiple methods with the same HTTP method and path but different `@Produces` annotations, and JAX-RS will call the method matching the content type requested by the client.
- The `@Consumes` annotation (p. 19) restricts the content type that a web service accepts from a `PUT` request.
- JAXB (Java Architecture for XML Binding; p. 19) is a set of classes for converting POJOs to and from XML. Class `JAXB` (package `javax.xml.bind`) contains `static` methods for common operations.
- Class `JAXB`'s `static` method `marshal` (p. 19) converts a Java object to XML format.
- WADL (Web Application Description Language; p. 20) has similar design goals to WSDL, but describes RESTful services instead of SOAP services.

Section 32.7.2 Consuming a REST-Based XML Web Service

- Clients of RESTful web services do not require web service references.
- The `JAXB` class has a `static unmarshal` method that takes as arguments a filename or URL as a `String`, and a `Class<T>` object indicating the Java class to which the XML will be converted.

Section 32.8 Publishing and Consuming REST-Based JSON Web Services

- JSON components—objects, arrays, strings, numbers—can be easily mapped to constructs in Java and other programming languages.

Section 32.8.1 Creating a REST-Based JSON Web Service

- To add a JAR file as a library in NetBeans, right click your project's **Libraries** folder, select **Add JAR/Folder…**, locate the JAR file and click **Open**.
- For a web service method that returns JSON text, the argument to the `@Produces` attribute must be `"application/json"`.
- In JSON, all data must be encapsulated in a composite data type.
- Create a `Gson` object (from package `com.google.gson`) and call its `toJson` method to convert an object into its JSON `String` representation.

Section 32.8.2 Consuming a REST-Based JSON Web Service

- To read JSON data from a URL, create a `URL` object and call its `openStream` method (p. 26). This invokes the web service and returns an `InputStream` from which the client can read the response. Wrap the `InputStream` in an `InputStreamReader` so it can be passed as the first argument to the `Gson` class's `fromJson` method (p. 26).

Section 32.9 Session Tracking in a SOAP Web Service

- It can be beneficial for a web service to maintain client state information, thus eliminating the need to pass client information between the client and the web service multiple times. Storing session information also enables a web service to distinguish between clients.

Section 31.9.1 Creating a `Blackjack` Web Service

- In JAX-WS 2.2, to enable session tracking in a web service, you simply precede your web service class with the `@HttpSessionScope` annotation (p. 29) from package `com.sun.xml.ws.developer.servlet`. To use this package you must add the JAX-WS 2.2 library to your project.
- Once a web service is annotated with `@HttpSessionScope`, the server automatically maintains a separate instance of the class for each client session.

Section 31.9.2 Consuming the `Blackjack` Web Service

- In the JAX-WS framework, the client must indicate whether it wants to allow the web service to maintain session information. To do this, first cast the proxy object to interface type `BindingProvider`. A `BindingProvider` enables the client to manipulate the request information that will be sent to the server. This information is stored in an object that implements interface `RequestContext`. The `BindingProvider` and `RequestContext` are part of the framework that is created by the IDE when you add a web service client to the application.
- Next, invoke the `BindingProvider`'s `getRequestContext` method to obtain the `RequestContext` object. Then call the `RequestContext`'s `put` method to set the property `BindingProvider.SESSION_MAINTAIN_PROPERTY` to `true`, which enables session tracking from the client side so that the web service knows which client is invoking the service's web methods.

Section 32.11 Equation Generator: Returning User-Defined Types

- It's also possible to process instances of class types in a web service. These types can be passed to or returned from web service methods.
- An instance variable can be serialized only if it's `public` or has both a *get* and a *set* method.

- Properties that can be generated from the values of other properties should not be serialized to prevent redundancy.
- JAX-RS automatically converts arguments from an `@Path` annotation to the correct type, and it will return a "not found" error to the client if the argument cannot be converted from the `String` passed as part of the URL to the destination type. Supported types for conversion include integer types, floating-point types, `boolean` and the corresponding type-wrapper classes.

Self-Review Exercises

32.1 State whether each of the following is *true* or *false*. If *false*, explain why.

a) All methods of a web service class can be invoked by clients of that web service.
b) When consuming a web service in a client application created in NetBeans, you must create the proxy class that enables the client to communicate with the web service.
c) A proxy class communicating with a web service normally uses SOAP to send and receive messages.
d) Session tracking is automatically enabled in a client of a web service.
e) Web methods cannot be declared `static`.
f) A user-defined type used in a web service must define both *get* and *set* methods for any property that will be serialized.
g) Operations in a REST web service are defined by their own unique URLs.
h) A SOAP-based web service can return data in JSON format.

32.2 Fill in the blanks for each of the following statements:

a) A key difference between SOAP and REST is that SOAP messages have data wrapped in a(n) ________.
b) A web service in Java is a(n) ________—it does not need to implement any interfaces or extend any classes.
c) Web service requests are typically transported over the Internet via the ________ protocol.
d) To set the exposed name of a web method, use the ________ element of the `@WebMethod` annotation.
e) ________ transforms an object into a format that can be sent between a web service and a client.
f) To return data in JSON format from a method of a REST-based web service, the `@Produces` annotation is set to ________.
g) To return data in XML format from a method of a REST-based web service, the `@Produces` annotation is set to ________.

Answers to Self-Review Exercises

32.1 a) False. Only methods declared with the `@WebMethod` annotation can be invoked by a web service's clients. b) False. The proxy class is created by NetBeans when you add a web service client to the application. c) True. d) False. In the JAX-WS framework, the client must indicate whether it wants to allow the web service to maintain session information. First, you must cast the proxy object to interface type `BindingProvider`, then use the `BindingProvider`'s `getRequestContext` method to obtain the `RequestContext` object. Finally, you must use the `RequestContext`'s `put` method to set the property `BindingProvider.SESSION_MAINTAIN_PROPERTY` to `true`. e) True. f) True. g) True. h) False. A SOAP web service implicitly returns data in XML format.

32.2 a) SOAP message or SOAP envelope. b) POJO (plain old Java object) c) HTTP. d) `operationName`. e) serialization. f) `"application/json"`. g) `"application/xml"`.

Exercises

32.3 *(Phone Book Web Service)* Create a RESTful web service that stores phone book entries in the database `PhoneBookDB` and a web client application that consumes this service. The web service should output XML. Use the steps in Section 31.2.1 to create the `PhoneBook` database and a data source name for accessing it. The database contains one table—`PhoneBook`—with three columns—`LastName`, `FirstName` and `PhoneNumber`. The `LastName` and `FirstName` columns store up to 30 characters. The `PhoneNumber` column supports phone numbers of the form `(800) 555-1212` that contain 14 characters. Use the `PhoneBookDB.sql` script provided in the examples folder to create the `PhoneBook` table.

Give the client user the capability to enter a new contact (web method `addEntry`) and to find contacts by last name (web method `getEntries`). Pass only `String`s as arguments to the web service. The `getEntries` web method should return an array of `String`s that contains the matching phone book entries. Each `String` in the array should consist of the last name, first name and phone number for one phone book entry. These values should be separated by commas.

The `SELECT` query that will find a `PhoneBook` entry by last name should be:

```
SELECT LastName, FirstName, PhoneNumber
FROM PhoneBook
WHERE (LastName = LastName)
```

The `INSERT` statement that inserts a new entry into the `PhoneBook` database should be:

```
INSERT INTO PhoneBook (LastName, FirstName, PhoneNumber)
VALUES (LastName, FirstName, PhoneNumber)
```

32.4 *(Phone Book Web Service Modification)* Modify Exercise 32.3 so that it uses a class named `PhoneBookEntry` to represent a row in the database. The web service should return objects of type `PhoneBookEntry` in XML format for the `getEntries` method, and the client application should use the `JAXB` method `unmarshal` to retrieve the `PhoneBookEntry` objects.

32.5 *(Phone-Book Web Service with JSON)* Modify Exercise 32.4 so that the `PhoneBookEntry` class is passed to and from the web service as a JSON object. Use serialization to convert the JSON object into an object of type `PhoneBookEntry`.

32.6 *(`Blackjack` Web Service Modification)* Modify the `Blackjack` web service example in Section 32.9 to include class `Card`. Modify web method `dealCard` so that it returns an object of type `Card` and modify web method `getHandValue` so that it receives an array of `Card` objects from the client. Also modify the client application to keep track of what cards have been dealt by using `ArrayList`s of `Card` objects. The proxy class created by NetBeans will treat a web method's array parameter as a `List`, so you can pass these `ArrayList`s of `Card` objects directly to the `getHandValue` method. Your `Card` class should include *set* and *get* methods for the face and suit of the card.

32.7 *(Project: Airline Reservation Web-Service Modification)* Modify the airline reservation web service in Section 32.10 so that it contains two separate methods—one that allows users to view all available seats, and another that allows users to reserve a particular seat that is currently available. Use an object of type `Ticket` to pass information to and from the web service. The web service must be able to handle cases in which two users view available seats, one reserves a seat and the second user tries to reserve the same seat, not knowing that it's now taken. The names of the methods that execute should be `reserve` and `getAllAvailableSeats`.

32.8 *(Project: Morse Code Web Service)* In Exercise 14.22, you learned about Morse Code and wrote applications that could translate English phrases into Morse Code and vice versa. Create a SOAP-based web service that provides two methods—one that translates an English phrase into

Morse Code and one that translates Morse Code into English. Next, build a Morse Code translator GUI application that invokes the web service to perform these translations.

Making a Difference

32.9 *(Project: Spam Scanner Web Service)* In Exercise 14.27, you created a spam scanner application that scanned an e-mail and gave it a point rating based on the occurrence of certain words and phrases that commonly appear in spam e-mails and how many times the words and phrases occurred in the e-mail. Create a SOAP-based Spam scanner web service. Next, modify the GUI application you created in Exercise 14.27 to use the web service to scan an e-mail. Then display the point rating returned by the web service.

32.10 *(Project: SMS Web Service)* In Exercise 14.28, you created an SMS message-translator application. Create a SOAP-based web service with three methods:

a) one that receives an SMS abbreviation and returns the corresponding English word or phrase,
b) one that receives an entire SMS message and returns the corresponding English text, and
c) one that translates English text into an SMS message.

Use the web service from a GUI application that displays the web service's responses.

32.11 *(Project: Gender-Neutrality Web Service)* In Exercise 1.12, you researched eliminating sexism in all forms of communication. You then described the algorithm you'd use to read through a paragraph of text and replace gender-specific words with gender-neutral equivalents. Create a SOAP-based web service that receives a paragraph of text, then replaces gender-specific words with gender-neutral ones. Use the web service from a GUI application that displays the resulting gender-neutral text.