

*A client is to me a mere unit,  
a factor in a problem.*

—Sir Arthur Conan Doyle

*They also serve who only  
stand and wait.*

—John Milton

*...if the simplest things of  
nature have a message that  
you understand, rejoice, for  
your soul is alive.*

—Eleanora Duse

*Protocol is everything*

—Francoise Giuliani

# Web Services

## OBJECTIVES

In this chapter you will learn:

- What a web service is.
- How to publish and consume Java web services in Netbeans.
- The elements that comprise web services, such as service descriptions and classes that implement web services.
- How to create client desktop and web applications that invoke web service methods.
- The important part that XML and the Simple Object Access Protocol (SOAP) play in enabling 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.
- How to build a REST-based web service in ASP.NET.

- 28.1 Introduction
- 28.2 Java Web Services Basics
- 28.3 Creating, Publishing, Testing and Describing a Web Service
  - 28.3.1 Creating a Web Application Project and Adding a Web Service Class in Netbeans
  - 28.3.2 Defining the HugeInteger Web Service in Netbeans
  - 28.3.3 Publishing the HugeInteger Web Service from Netbeans
  - 28.3.4 Testing the HugeInteger Web Service with Sun Java System Application Server's Tester Web Page
  - 28.3.5 Describing a Web Service with the Web Service Description Language (WSDL)
- 28.4 Consuming a Web Service
  - 28.4.1 Creating a Client in Netbeans to Consume the HugeInteger Web Service
  - 28.4.2 Consuming the HugeInteger Web Service
- 28.5 SOAP
- 28.6 Session Tracking in Web Services
  - 28.6.1 Creating a Blackjack Web Service
  - 28.6.2 Consuming the Blackjack Web Service
- 28.7 Consuming a Database-Driven Web Service from a Web Application
  - 28.7.1 Configuring Java DB in Netbeans and Creating the Reservation Database
  - 28.7.2 Creating a Web Application to Interact with the Reservation Web Service
- 28.8 Passing an Object of a User-Defined Type to a Web Service
- 28.9 REST-Based Web Services in ASP.NET
  - 28.9.1 REST-Based Web Service Functionality
  - 28.9.2 Creating an ASP.NET REST-Based Web Service
  - 28.9.3 Adding Data Components to a Web Service
- 28.10 Web Resources

Summary | Terminology | Self-Review Exercises | Exercises

## 28.1 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 via method calls by an application (or other software component) on another computer over a network. Web services communicate using such technologies as XML and HTTP. Several Java APIs facilitate web services. In this chapter, we'll be dealing with Java APIs that are 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. There are other web services technologies, such as Representational State Transfer (REST), which we cover in the context of ASP.NET web services in Section 28.9. For information on web services, see the web resources in Section 28.10 and visit our Web Services Resource Center at [www.deitel.com/WebServices](http://www.deitel.com/WebServices). The Web Services Resource Center

includes information on designing and implementing web services in many languages, and information 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. [Note: This chapter assumes that you know Java for Sections 28.2–28.8. To learn more about Java, check out *Java How to Program, Seventh Edition*, or visit our Java Resource Centers at [www.deitel.com/ResourceCenters.html](http://www.deitel.com/ResourceCenters.html). For Section 28.9, the chapter assumes you know Visual Basic and ASP.NET. To learn more about Visual Basic and ASP.NET, check out our book *Visual Basic 2005 How to Program, Third Edition* or visit our Visual Basic Resource Center ([www.deitel.com/visualbasic/](http://www.deitel.com/visualbasic/)) and our ASP.NET Resource Center ([www.deitel.com/aspdotnet/](http://www.deitel.com/aspdotnet/)).]

Web services have important implications for **business-to-business (B2B) transactions**. They enable businesses to conduct transactions via standardized, widely available web services rather than relying on proprietary applications. Web services and SOAP are platform and language independent, so companies can collaborate via web services without worrying about the compatibility of their hardware, software and communications technologies. Companies such as Amazon, Google, eBay, PayPal and many others are using web services to their advantage by making their server-side applications available to partners via web services.

By purchasing web services and using extensive free web services that are relevant to their businesses, companies can spend less time developing new applications and can create innovative new applications. E-businesses can use web services to provide their customers with enhanced shopping experiences. Consider an online music store. The store's website links to information about various CDs, enabling users to purchase the CDs, to learn about the artists, to find more titles by those artists, to find other artists' music they may enjoy, and more. Another company that sells concert tickets provides a web service that displays upcoming concert dates for various artists and allows 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 can “consume” web services. The resulting applications would call web service methods of objects running on servers that could be thousands of miles away. To learn more about Java web services read the Java Technology and Web Services Overview at [java.sun.com/webservices/overview.html](http://java.sun.com/webservices/overview.html).

### *Netbeans*

Netbeans—developed by Sun—is one of the many tools that enable programmers to “publish” and/or “consume” web services. We demonstrate how to use Netbeans to implement web services and 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 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 5.5.1 IDE, its Visual Web Pack and the Sun Java System Application Server (SJSAS), see Section 26.1.

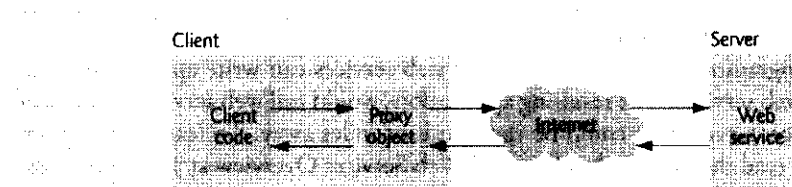
## 28.2 Java Web Services Basics

A web service normally resides on a **server**. The application (i.e., the client) that accesses the web service sends a method call over a network to the remote machine, which processes the call and returns a response over the network to the application. This kind of distributed computing is beneficial in many applications. For example, a client application without direct access to a database on a remote server 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. In previous chapters, all the pieces of an application resided on one machine. 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**; using a web service from a client application is known as **consuming a web service**. An application that consumes a web service consists of two parts—an object of a **proxy class** for interacting with the web service and a client application that consumes the web service by invoking methods on the object of the proxy class. The client code invokes methods on the proxy object, which handles the details of communicating with the web service (such as passing method arguments to the web service 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 proxy object, which then returns the results to the client code. Figure 28.1 depicts the interactions among the client code, the proxy class and the web service. As you'll soon see, Netbeans creates these proxy classes for you in your client applications.

Requests to and responses from web services created with **JAX-WS 2.0** (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 discuss SOAP in Section 28.5.



**Fig. 28.1** | Interaction between a web service client and a web service.

## 28.3 Creating, Publishing, Testing and Describing a Web Service

The following subsections demonstrate how to create, publish and test a `HugeInteger` web service that performs calculations with positive integers up to 100 digits long (maintained as arrays of digits). Such integers are much larger than Java's integral primitive types can represent. The `HugeInteger` web service provides methods that take two "huge integers"



(represented as `Strings`) and determine their sum, their difference, which is larger, which is smaller or whether the two numbers are equal. These methods will be services available to other applications via the web—hence the term web services.

### 28.3.1 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. 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 **Web** from the dialog's **Categories** list, then select **Web Application** from the **Projects** list. Click **Next >**.
3. Specify the name of your project (`HugeInteger`) 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.
4. Select **Sun Java System Application Server 9** from the **Server** drop-down list.
5. Select **Java EE 5** from the **J2EE Version** drop-down list.
6. Click **Finish** to dismiss the **New Project** dialog.

This creates a web application that will run in a web browser, similar to the **Visual Web Application** projects used in Chapters 26 and 27. Netbeans generates additional files to support the web application. This chapter discusses only the web-service-specific files.

#### *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 `HugeInteger` project's node and select **New > Web Service...** to open the **New Web Service** dialog.
2. Specify `HugeInteger` in the **Web Service Name** field.
3. Specify `com.deitel.iw3htp4.ch28.hugeinteger` in the **Package** field.
4. Click **Finish** to dismiss the **New Web Service** dialog.

The IDE generates a sample web service class with the name you specified in *Step 2*. You can find this class in the **Projects** tab under the **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 (which we'll discuss shortly) for your web service.

### 28.3.2 Defining the HugeInteger Web Service in Netbeans

Figure 28.2 contains the `HugeInteger` web service's code. You can implement this code yourself in the `HugeInteger.java` file created in Section 28.3.1, or you can simply replace the code in `HugeInteger.java` with a copy of our code from this example's folder. You

can find this file in the project's `src\java\com\deitel\iw3http4\ch28\hugeinteger` folder. The book's examples can be downloaded from [www.deitel.com/books/iw3http4/](http://www.deitel.com/books/iw3http4/).

```

1 // Fig. 28.2: HugeInteger.java
2 // HugeInteger web service that performs operations on large integers.
3 package com.deitel.iw3http4.ch28.hugeinteger;
4
5 import javax.xml.ws.WebService; // program uses the annotation @WebService
6 import javax.xml.ws.WebMethod; // program uses the annotation @WebMethod
7 import javax.xml.ws.WebParam; // program uses the annotation @WebParam
8
9 @WebService( // annotates the class as a web service
10     name = "HugeInteger", // sets class name
11     serviceName = "HugeIntegerService" ) // sets the service name
12 public class HugeInteger
13 {
14     private final static int MAXIMUM = 100; // maximum number of digits
15     public int[] number = new int[ MAXIMUM ]; // stores the huge integer
16
17     // returns a String representation of a HugeInteger
18     public String toString()
19     {
20         String value = "";
21
22         // convert HugeInteger to a String
23         for ( int digit : number )
24             value = digit + value; // places next digit at beginning of value
25
26         // locate position of first non-zero digit
27         int length = value.length();
28         int position = -1;
29
30         for ( int i = 0; i < length; i++ )
31         {
32             if ( value.charAt( i ) != '0' )
33             {
34                 position = i; // first non-zero digit
35                 break;
36             }
37         } // end for
38
39         return ( position != -1 ? value.substring( position ) : "0" );
40     } // end method toString
41
42     // creates a HugeInteger from a String
43     public static HugeInteger parseHugeInteger( String s )
44     {
45         HugeInteger temp = new HugeInteger();
46         int size = s.length();
47
48         for ( int i = 0; i < size; i++ )
49             temp.number[ i ] = s.charAt( size - i - 1 ) - '0';

```

**Fig. 28.2** | HugeInteger web service that performs operations on large integers. (Part I of 3.)

```

50     return temp;
51 } // end method parseHugeInteger
52
53 // WebMethod that adds huge integers represented by String arguments
54 @WebMethod( operationName = "add" )
55 public String add( @WebParam( name = "first" ) String first,
56                  @WebParam( name = "second" ) String second )
57 {
58     int carry = 0; // the value to be carried
59     HugeInteger operand1 = HugeInteger.parseHugeInteger( first );
60     HugeInteger operand2 = HugeInteger.parseHugeInteger( second );
61     HugeInteger result = new HugeInteger(); // stores addition result
62
63     // perform addition on each digit
64     for ( int i = 0; i < MAXIMUM; i++ )
65     {
66         // add corresponding digits in each number and the carried value
67         // store result in the corresponding column of HugeInteger result
68         result.number[ i ] =
69             ( operand1.number[ i ] + operand2.number[ i ] + carry ) % 10;
70
71         // set carry for next column
72         carry =
73             ( operand1.number[ i ] + operand2.number[ i ] + carry ) / 10;
74     } // end for
75
76     return result.toString();
77 } // end WebMethod add
78
79 // WebMethod that subtracts integers represented by String arguments
80 @WebMethod( operationName = "subtract" )
81 public String subtract( @WebParam( name = "first" ) String first,
82                       @WebParam( name = "second" ) String second )
83 {
84     HugeInteger operand1 = HugeInteger.parseHugeInteger( first );
85     HugeInteger operand2 = HugeInteger.parseHugeInteger( second );
86     HugeInteger result = new HugeInteger(); // stores difference
87
88     // subtract bottom digit from top digit
89     for ( int i = 0; i < MAXIMUM; i++ )
90     {
91         // if the digit in operand1 is smaller than the corresponding
92         // digit in operand2, borrow from the next digit
93         if ( operand1.number[ i ] < operand2.number[ i ] )
94             operand1.borrow( i );
95
96         // subtract digits
97         result.number[ i ] = operand1.number[ i ] - operand2.number[ i ];
98     } // end for
99
100     return result.toString();
101 } // end WebMethod subtract
102

```

Fig. 28.2 | HugeInteger web service that performs operations on large integers. (Part 2 of 3.)

```

188 // borrow 1 from next digit
189 private void borrow( int place )
190 {
191     if ( place >= MAXIMUM )
192         throw new IndexOutOfBoundsException();
193     else if ( number[ place + 1 ] == 0 ) // if next digit is zero
194         borrow( place + 1 ); // borrow from next digit
195     number[ place ] += 10; // add 10 to the borrowing digit
196     number[ place + 1 ]--; // subtract one from the next digit
197 } // end method borrow
198
199 // webMethod that returns true if first integer is greater
200 @WebMethod( operationName = "bigger" )
201 public boolean bigger( @WebParam( name = "first" ) String first,
202                      @WebParam( name = "second" ) String second )
203 {
204     try // try subtracting first from second
205     {
206         String difference = subtract( first, second );
207         return (difference.matches( "[0]+") );
208     } // end try
209     catch ( IndexOutOfBoundsException e ) // first is less than
210     {
211         return false;
212     } // end catch
213 } // end webMethod bigger
214
215 // webMethod that returns true if the first integer is less
216 @WebMethod( operationName = "smaller" )
217 public boolean smaller( @WebParam( name = "first" ) String first,
218                      @WebParam( name = "second" ) String second )
219 {
220     return bigger( second, first );
221 } // end webMethod smaller
222
223 // webMethod that returns true if the first integer
224 @WebMethod( operationName = "equals" )
225 public boolean equals( @WebParam( name = "first" ) String first,
226                     @WebParam( name = "second" ) String second )
227 {
228     return ( ( bigger( first, second ) ) || smaller( first, second ) );
229 } // end webMethod equals
230 } // end class HugeInteger

```

**Fig. 28.2** | HugeInteger web service that performs operations on large integers. (Part 3 of 3.)

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), meaning that—unlike prior Java web service APIs—you do not need to extend a class or implement an interface to create a web service. When you compile a class that uses these JAX-WS 2.0 annotations, the compiler creates all the server-side artifacts that support the web

service—that is, the compiled code framework that allows the web service to wait for client requests and respond to those requests once the service is deployed on an application server. Popular application servers that support Java web services include the Sun Java System Application Server ([www.sun.com/software/products/appsrvr/index.xml](http://www.sun.com/software/products/appsrvr/index.xml)), GlassFish ([glassfish.dev.java.net](http://glassfish.dev.java.net)), Apache Tomcat ([tomcat.apache.org](http://tomcat.apache.org)), BEA Weblogic Server ([www.bea.com](http://www.bea.com)) and JBoss Application Server ([www.jboss.org/products/jbossas](http://www.jboss.org/products/jbossas)). We use Sun Java System Application Server in this chapter.

Lines 9–11 contain a `@WebService` annotation (imported at line 5) with properties `name` and `serviceName`. The `@WebService` annotation indicates that class `HugeInteger` implements a web service. The annotation is followed by a set of parentheses containing optional elements. The annotation's `name` element (line 10) specifies the name of the proxy class that will be generated for the client. The annotation's `serviceName` element (line 11) specifies the name of the class that the client uses to obtain an object of the proxy class. [Note: If the `serviceName` element is not specified, the web service's name is assumed to be the 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.

Line 14 declares the constant `MAXIMUM` that specifies the maximum number of digits for a `HugeInteger` (i.e., 100 in this example). Line 15 creates the array that stores the digits in a huge integer. Lines 18–40 declare method `toString`, which returns a `String` representation of a `HugeInteger` without any leading 0s. Lines 43–52 declare static method `parseHugeInteger`, which converts a `String` into a `HugeInteger`. The web service's methods `add`, `subtract`, `bigger`, `smaller` and `equals` use `parseHugeInteger` to convert their `String` arguments to `HugeIntegers` for processing.

`HugeInteger` methods `add`, `subtract`, `bigger`, `smaller` and `equals` are tagged with the `@WebMethod` annotation (lines 55, 81, 117, 133 and 141) to indicate that they 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. Note that the `@WebMethod` annotations each use the `operationName` element to specify the method name that is exposed to the web service's client.



### Common Programming Error 28.1

*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.*



### Common Programming Error 28.2

*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.*

Each web method in class `HugeInteger` specifies parameters that are annotated with the `@WebParam` annotation (e.g., lines 56–57 of method `add`). The optional `@WebParam` element `name` indicates the parameter name that is exposed to the web service's clients.

Lines 55–78 and 81–102 declare `HugeInteger` web methods `add` and `subtract`. We assume for simplicity that `add` does not result in overflow (i.e., the result will be 100 digits or fewer) and that `subtract`'s first argument will always be larger than the second. The `subtract` method calls method `borrow` (lines 105–114) when it is necessary to borrow 1 from the next digit to the left in the first argument—that is, when a particular digit in the

left operand is smaller than the corresponding digit in the right operand. Method `borrow` adds 10 to the appropriate digit and subtracts 1 from the next digit to the left. This utility method is not intended to be called remotely, so it is not tagged with `@WebMethod`.

Lines 117–130 declare `HugeInteger` web method `bigger`. Line 123 invokes method `subtract` to calculate the difference between the numbers. If the first number is less than the second, this results in an exception. In this case, `bigger` returns `false`. If `subtract` does not throw an exception, then line 124 returns the result of the expression

```
!difference.matches( "[0]+" )
```

This expression calls `String` method `matches` to determine whether the `String` `difference` matches the regular expression `"[0]+"`, which determines if the `String` consists only of one or more 0s. The symbols `^` and `$` indicate that `matches` should return `true` only if the entire `String` `difference` matches the regular expression. We then use the logical negation operator (`!`) to return the opposite `boolean` value. Thus, if the numbers are equal (i.e., their difference is 0), the preceding expression returns `false`—the first number is not greater than the second. Otherwise, the expression returns `true`.

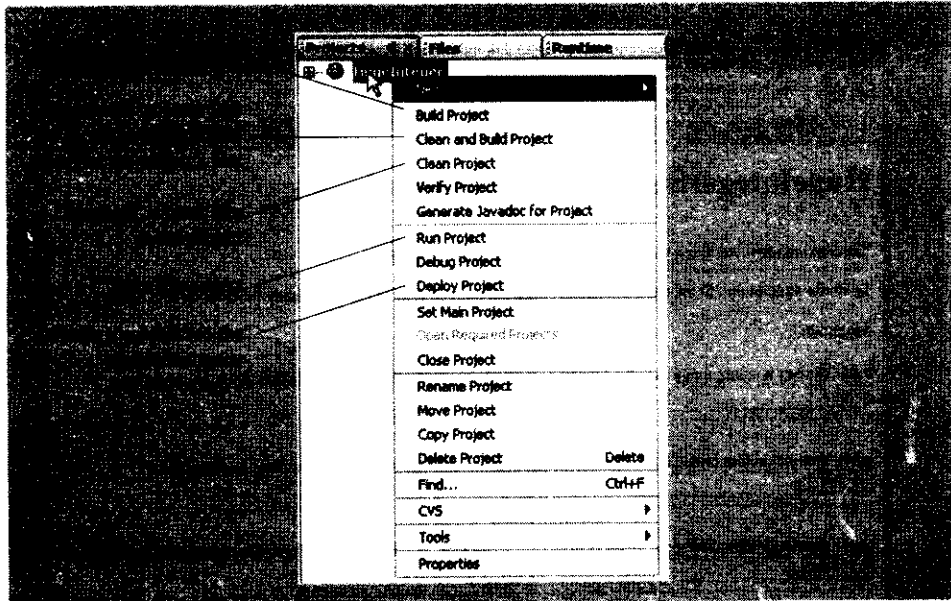
Lines 133–146 declare methods `smaller` and `equals`. Method `smaller` returns the result of invoking method `bigger` (line 137) with the arguments reversed—if `first` is less than `second`, then `second` is greater than `first`. Method `equals` invokes methods `bigger` and `smaller` (line 145). If either `bigger` or `smaller` returns `true`, line 145 returns `false`, because the numbers are not equal. If both methods return `false`, the numbers are equal and line 145 returns `true`.

### 28.3.3 Publishing the `HugeInteger` Web Service from Netbeans

Now that we've created the `HugeInteger` web service class, we'll use Netbeans to build and publish (i.e., 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 (`HugeInteger`) in the Netbeans **Projects** tab to display the pop-up menu shown in Fig. 28.3. To determine if there are any compilation errors in your project, select the **Build Project** option. When the project compiles successfully, you can select **Deploy Project** to deploy the project to the server you selected when you set up the web application in Section 28.3.1. If the code in the project has changed since the last build, selecting **Deploy Project** also builds the project. Selecting **Run Project** executes the web application. If the web application was not previously built or deployed, this option performs these tasks first. Note that both the **Deploy Project** and **Run Project** options also start the application server (in our case Sun Java System Application Server) if it is not already running. To ensure that all source-code files in a project are recompiled during the next build operation, you can use the **Clean Project** or **Clean and Build Project** options. If you have not already done so, select **Deploy Project** now.

### 28.3.4 Testing the `HugeInteger` Web Service with Sun Java System Application Server's Tester Web page

The next step is to test the `HugeInteger` web service. We previously selected the Sun Java System Application Server to execute this web application. This server can dynamically



**Fig. 28.3** | Pop-up menu that appears when you right click a project name in the Netbeans **Projects** tab.

create a web page for testing a web service's methods from a web browser. To enable this capability:

1. Right click the project name (HugeInteger) in the Netbeans **Projects** tab and select **Properties** from the pop-up menu to display the **Project Properties** dialog.
2. Click **Run** under **Categories** to display the options for running the project.
3. In the **Relative URL** field, type `/HugeIntegerService?Tester`.
4. Click **OK** to dismiss the **Project Properties** dialog.

The **Relative URL** field specifies what should happen when the web application executes. If this field is empty, then the web application's default JSP displays when you run the project. When you specify `/HugeIntegerService?Tester` in this field, then run the project, Sun Java System Application Server builds the **Tester** web page and loads it into your web browser. Figure 28.4 shows the **Tester** web page for the **HugeInteger** web service. Once you've deployed the web service, you can also type the URL

`http://localhost:8080/HugeInteger/HugeIntegerService?Tester`

in your web browser to view the **Tester** web page. Note that **HugeIntegerService** is the name (specified in line 11 of Fig. 28.2) that clients, including the **Tester** web page, use to access the web service.

To test **HugeInteger**'s web methods, type two positive integers into the text fields to the right of a particular method's button, then click the button to invoke the web method and see the result. Figure 28.5 shows the results of invoking **HugeInteger**'s **add** method with the values 9999999999999999 and 1. Note that the number 9999999999999999 is larger than primitive type **long** can represent.



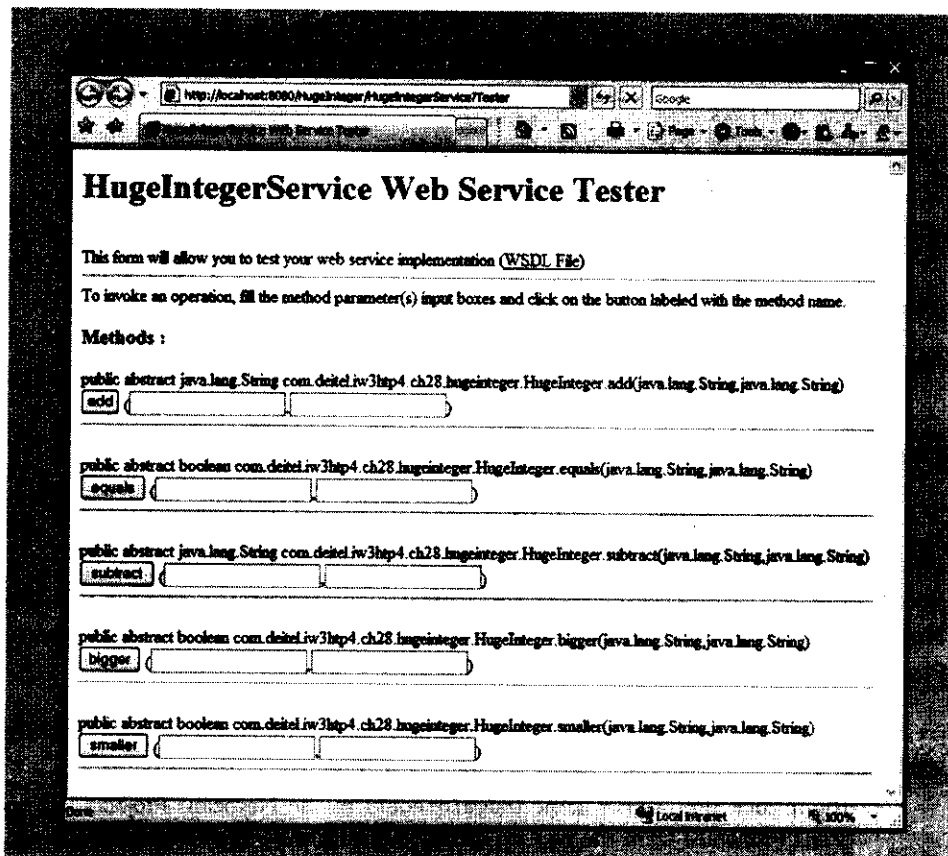


Fig. 28.4 | Tester web page created by Sun Java System Application Server for the HugelInteger web service.

Note that you can access the web service only when the application server is running. If Netbeans launches the application server for you, it will automatically shut it down when you close Netbeans. To keep the application server up and running, you can launch it independently of Netbeans before you deploy or run web applications in Netbeans. For Sun Java System Application Server running on Windows, you can do this by selecting

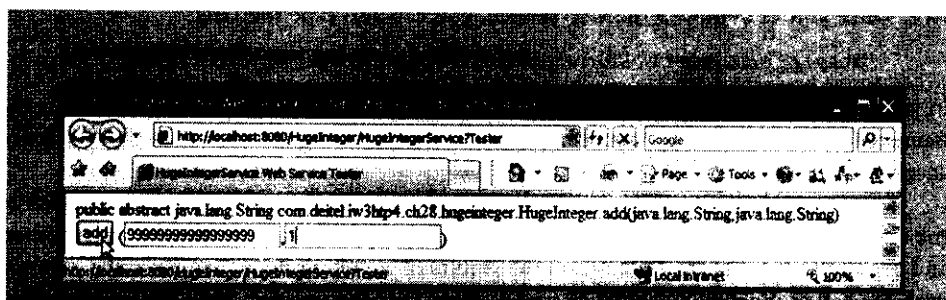
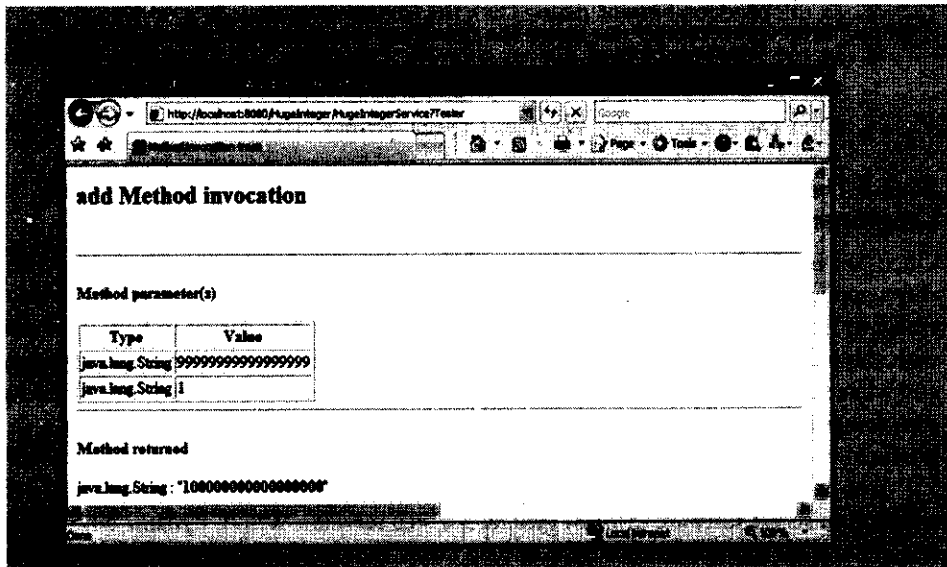


Fig. 28.5 | Testing HugelInteger's add method. (Part 1 of 2.)



**Fig. 28.5** | Testing HugeInteger's add method. (Part 2 of 2.)

Start > All Programs > Sun Microsystems > Application Server PE 9 > Start Default Server. To shut down the application server, you can select the Stop Default Server option from the same location.

#### *Testing the HugeInteger 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/HugeInteger/HugeIntegerService?Tester`

#### *Note to Windows XP Service Pack 2 and Windows Vista Users*

For security reasons, computers running Windows XP Service Pack 2 or Windows Vista do not allow HTTP requests from other computers by default. If you wish to allow other computers to connect to your computer using HTTP, perform the following steps on Windows XP SP2:

1. Select Start > Control Panel to open your system's Control Panel window, then double click Windows Firewall to view the Windows Firewall settings dialog.
2. In the Windows Firewall dialog, click the Exceptions tab, then click Add Port... and add port 8080 with the name SJSAS.
3. Click OK to dismiss the Windows Firewall settings dialog.

To allow other computers to connect to your Windows Vista computer using HTTP, perform the following steps:

1. Open the Control Panel, switch to Classic View and double click Windows Firewall to open the Windows Firewall dialog.

2. In the **Windows Firewall** dialog click the **Change Settings...** link.
3. In the **Windows Firewall** dialog, click the **Exceptions** tab, then click **Add Port...** and add port 8080 with the name **SJSAS**.
4. Click **OK** to dismiss the **Windows Firewall** settings dialog.

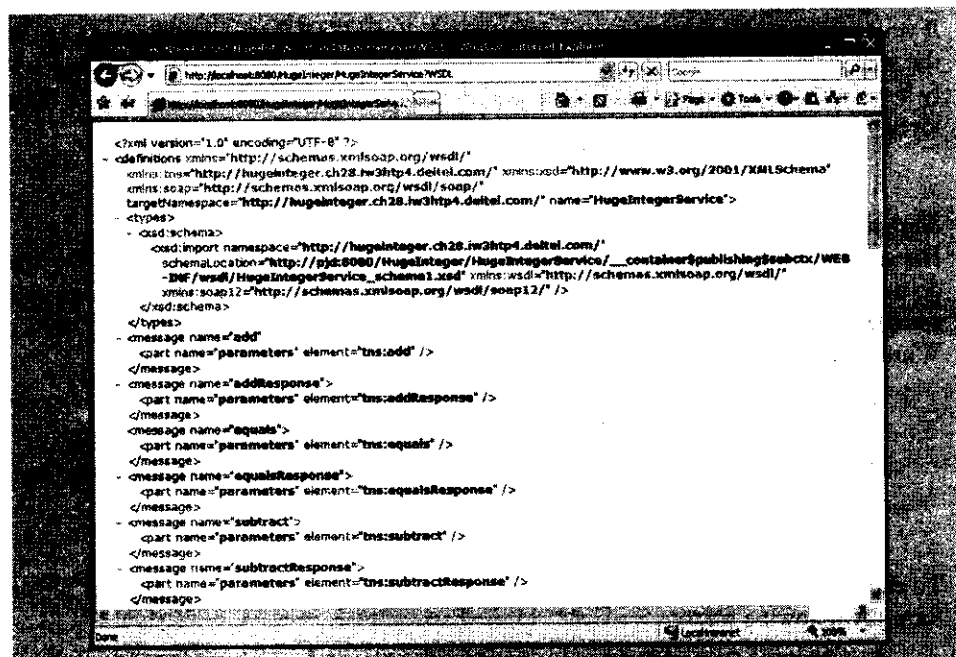
### 28.3.5 Describing a Web Service with the Web Service Description Language (WSDL)

Once you implement a web service, compile it and deploy it on an application server, a client application can consume the web service. To do so, however, the client must know where to find the web service and must be provided with a description of how to interact with the web service—that is, what methods are available, what parameters they expect and what each method returns. For this purpose, JAX-WS uses the **Web Service Description Language (WSDL)**—a standard XML vocabulary for describing web services in a platform-independent manner.

You do not need to understand the details of WSDL to take advantage of it—the application server software (SJSAS) 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. Since the WSDL is created dynamically, clients always receive a deployed web service's most up-to-date description. To view the WSDL for the **HugeInteger** web service (Fig. 28.6), enter the following URL in your browser:

`http://localhost:8080/HugeInteger/HugeIntegerService?WSDL`

or click the **WSDL File** link in the **Tester** web page (shown in Fig. 28.4).



**Fig. 28.6** | A portion of the .wsdl file for the **HugeInteger** web service.

*Accessing the HugeInteger Web Service's WSDL from Another Computer*

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

```
http://host:8080/HugeInteger/HugeIntegerService?WSDL
```

where *host* is the hostname or IP address of the computer on which the web service is deployed. As we discussed in Section 28.3.4, this will work only if your computer allows HTTP connections from other computers—as is the case for publicly accessible web and application servers.

## 28.4 Consuming a Web Service

Now that we've defined and deployed our web service, we can consume it 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. This process defines the proxy class that allows the client to access the web service.

### 28.4.1 Creating a Client in Netbeans to Consume the HugeInteger Web Service

In this section, you'll 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 web service reference, the IDE creates and compiles the client-side artifacts—the framework of Java code that supports the client-side proxy class. The client then calls methods on an object of the proxy 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 HugeInteger web service has been deployed and that the Sun Java System Application Server is running (see Section 28.3.3). 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 **General** from the **Categories** list and **Java Application** from the **Projects** list, then click **Next >**.
3. Specify the name `UsingHugeInteger` in the **Project Name** field and uncheck the **Create Main Class** checkbox. In a moment, you'll add a subclass of `JFrame` that contains a `main` method.
4. Click **Finish** to create the project.

#### *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 HugeInteger web service. To add a web service reference, perform the following steps.

1. Right click the project name (`UsingHugeInteger`) in the Netbeans **Projects** tab.
2. Select **New > Web Service Client...** from the pop-up menu to display the **New Web Service Client** dialog (Fig. 28.7).

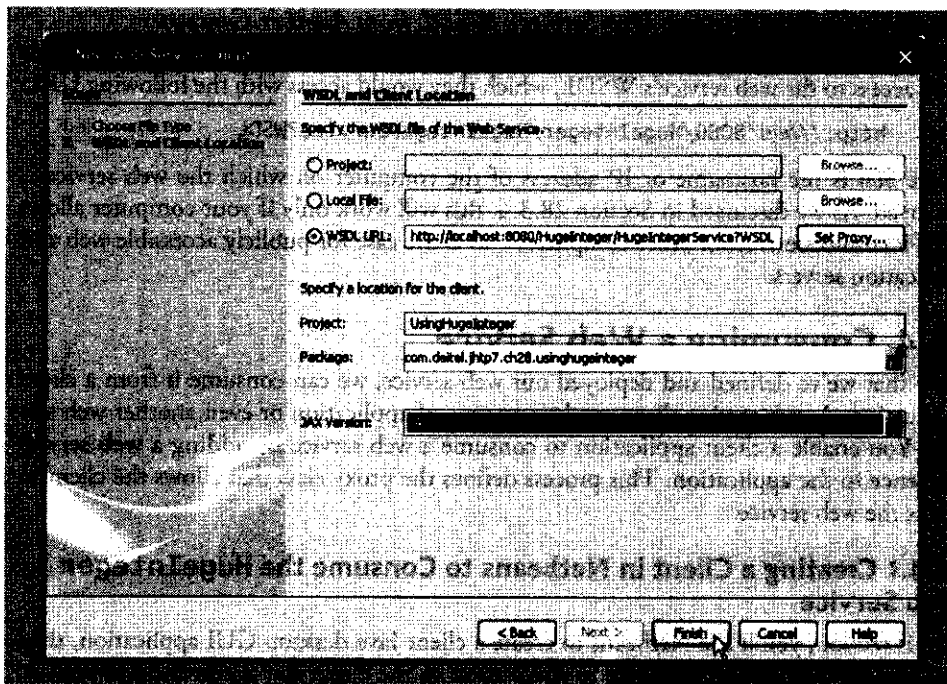
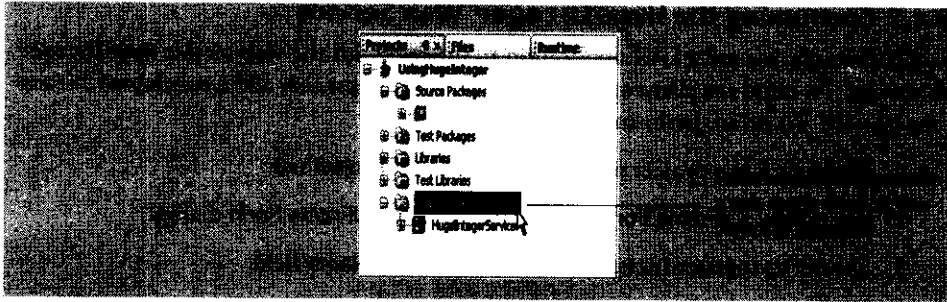


Fig. 28.7 | New Web Service Client dialog.

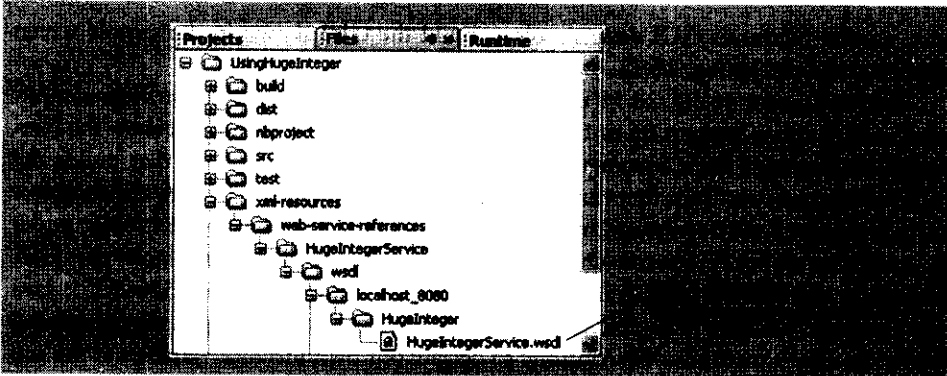
3. In the **WSDL URL** field, specify the URL `http://localhost:8080/HugeInteger/HugeIntegerService?WSDL` (Fig. 28.7). This URL tells the IDE where to find the web service's WSDL description. [Note: If the Sun Java System Application Server is located on a different computer, replace `localhost` with 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 proxy. Note that the **New Web Service Client** dialog enables you to search for web services in several locations. Many companies simply distribute the exact WSDL URLs for their web services, which you can place in the **WSDL URL** field.
4. In the **Package** field, specify `com.deitel.iw3http4.ch28.usinghugeinteger` as the package name.
5. Click **Finish** to dismiss the **New Web Service Client** dialog.

In the NetBeans **Projects** tab, the `UsingHugeInteger` project now contains a **Web Service References** folder with the `HugeInteger` web service's proxy (Fig. 28.8). Note that the proxy's name is listed as `HugeIntegerService`, as we specified in line 11 of Fig. 28.2.

When you specify the web service you want to consume, NetBeans accesses the web service's WSDL information and copies it into a file in your project (named `HugeIntegerService.wsdl` in this example). You can view this file from the NetBeans **Files** tab by expanding the nodes in the `UsingHugeInteger` project's `xml-resources` folder as shown in Fig. 28.9. 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 `HugeIntegerService` node shown in Fig. 28.8 and selecting **Refresh Client**.

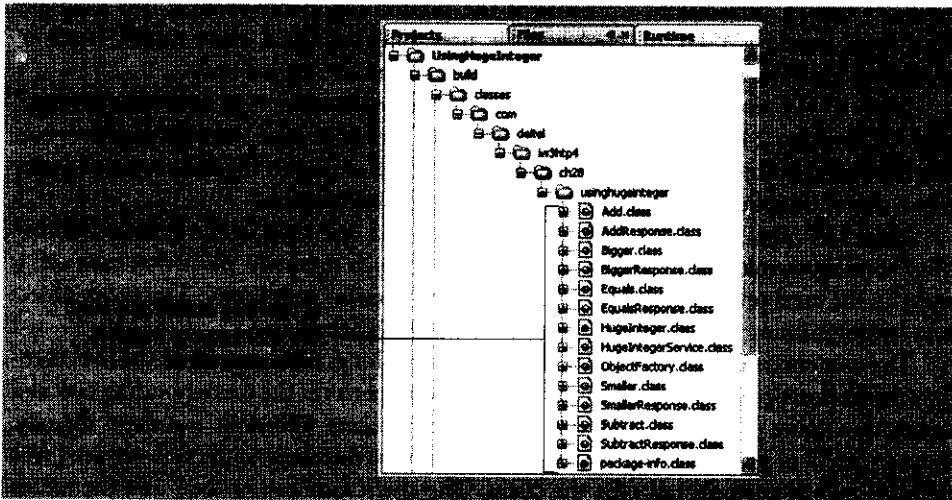


**Fig. 28.8** | Netbeans Project tab after adding a web service reference to the project.



**Fig. 28.9** | Locating the HugeIntegerService.wsdl file in the Netbeans Files tab.

You can view the IDE-generated client-side artifacts by selecting the Netbeans Files tab and expanding the UsingHugeInteger project's build folder as shown in Fig. 28.10.



**Fig. 28.10** | Viewing the HugeInteger web service's client-side artifacts generated by Netbeans.

### 28.4.2 Consuming the HugeInteger Web Service

For this example, we use a GUI application to interact with the web service `HugeInteger` web service. To build the client application's GUI, you must first add a subclass of `JFrame` to the project. To do so, perform the following steps:

1. Right click the project name in the Netbeans **Project** tab.
2. Select **New > JFrame Form...** to display the **New JFrame Form** dialog.
3. Specify `UsingHugeIntegerJFrame` in the **Class Name** field.
4. Specify `com.deitel.iw3htp4.ch28.hugeintegerclient` in the **Package** field.
5. 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. 28.11.

The application in Fig. 28.11 uses the `HugeInteger` web service to perform computations with positive integers up to 100 digits long. To save space, we do not show the Netbeans autogenerated  `initComponents`  method, which contains the code that builds the GUI components, positions them and registers their event handlers. To view the complete source code, open the `UsingHugeIntegerJFrame.java` file in this example's folder under `src\java\com\deitel\iw3htp4\ch28\hugeintegerclient`. Netbeans places the GUI component instance-variable declarations at the end of the class (lines 326–335). Java allows instance variables to be declared anywhere in a class's body as long as they are placed outside the class's methods. We continue to declare our own instance variables at the top of the class.

Lines 6–7 import the classes `HugeInteger` and `HugeIntegerService` that enable the client application to interact with the web service. We include these import declarations only for documentation purposes here. These classes are in the same package as `UsingHugeIntegerJFrame`, so these import declarations are not necessary. 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 11), so import declarations are unnecessary.

Lines 13–14 declare the variables of type `HugeIntegerService` and `HugeInteger`, respectively. Line 24 in the constructor creates an object of type `HugeIntegerService`. Line 25 uses this object's `getHugeIntegerPort` method to obtain the `HugeInteger` proxy object that the application uses to invoke the web service's method.

Lines 165–166, 189–190, 213–214, 240–241 and 267–268 in the various `JButton` event handlers invoke the `HugeInteger` web service's web methods. Note that each call is made on the local proxy object that is referenced by `hugeIntegerProxy`. The proxy object then communicates with the web service on the client's behalf.

The user enters two integers, each up to 100 digits long. Clicking any of the five `JButtons` causes the application to invoke a web method to perform the corresponding task and return the result. Our client application cannot process 100-digit numbers directly. Instead the client passes `String` representations of these numbers to the web service's web methods, which perform tasks for the client. The client application then uses the return value of each operation to display an appropriate message.



```

1 // File: UsingHugeIntegerServiceJFrame.java
2 // Client desktop application for the HugeInteger web service.
3 // Generated by NetBeans IDE 7.0.1
4
5 // Import classes for accessing HugeInteger web service's proxy
6 import com.deitel.iw3http4.ch28.hugeintegerclient.HugeInteger;
7 import com.deitel.iw3http4.ch28.hugeintegerclient.HugeIntegerService;
8
9 // The main method
10 public static void main(String[] args) {
11     // Create the client application
12     // Create the HugeIntegerService proxy
13     private HugeIntegerService hugeIntegerService; // used to obtain proxy
14     private HugeInteger hugeIntegerProxy; // used to access the web service
15
16     // Create the client application
17     UsingHugeIntegerJFrame frame = new UsingHugeIntegerJFrame();
18     frame.setVisible(true);
19 }
20
21 // The initComponents method is autogenerated by NetBeans and is called
22 // from the constructor to initialize the GUI. This method is not shown
23 // here to save space. Open UsingHugeIntegerJFrame.java in this
24 // example's folder to view the complete generated code (lines 37-152).
25
26 // The service class is used to access the web service
27 private HugeIntegerService hugeIntegerService = new HugeIntegerService();
28
29 // The proxy class is used to access the web service
30 private HugeInteger hugeIntegerProxy = hugeIntegerService.getHugeIntegerPort();
31
32 // The first number
33 private int firstNumber = 10;
34
35 // The second number
36 private int secondNumber = 20;
37
38 // The results text area
39 private JTextArea resultsJTextArea = new JTextArea();
40
41 // The add method
42 private void add() {
43     // Add the numbers
44     hugeIntegerProxy.add( firstNumber, secondNumber );
45 }
46
47 // End of file
48

```

**Fig. 28.11** | Client desktop application for the HugeInteger web service. (Part 1 of 6.)

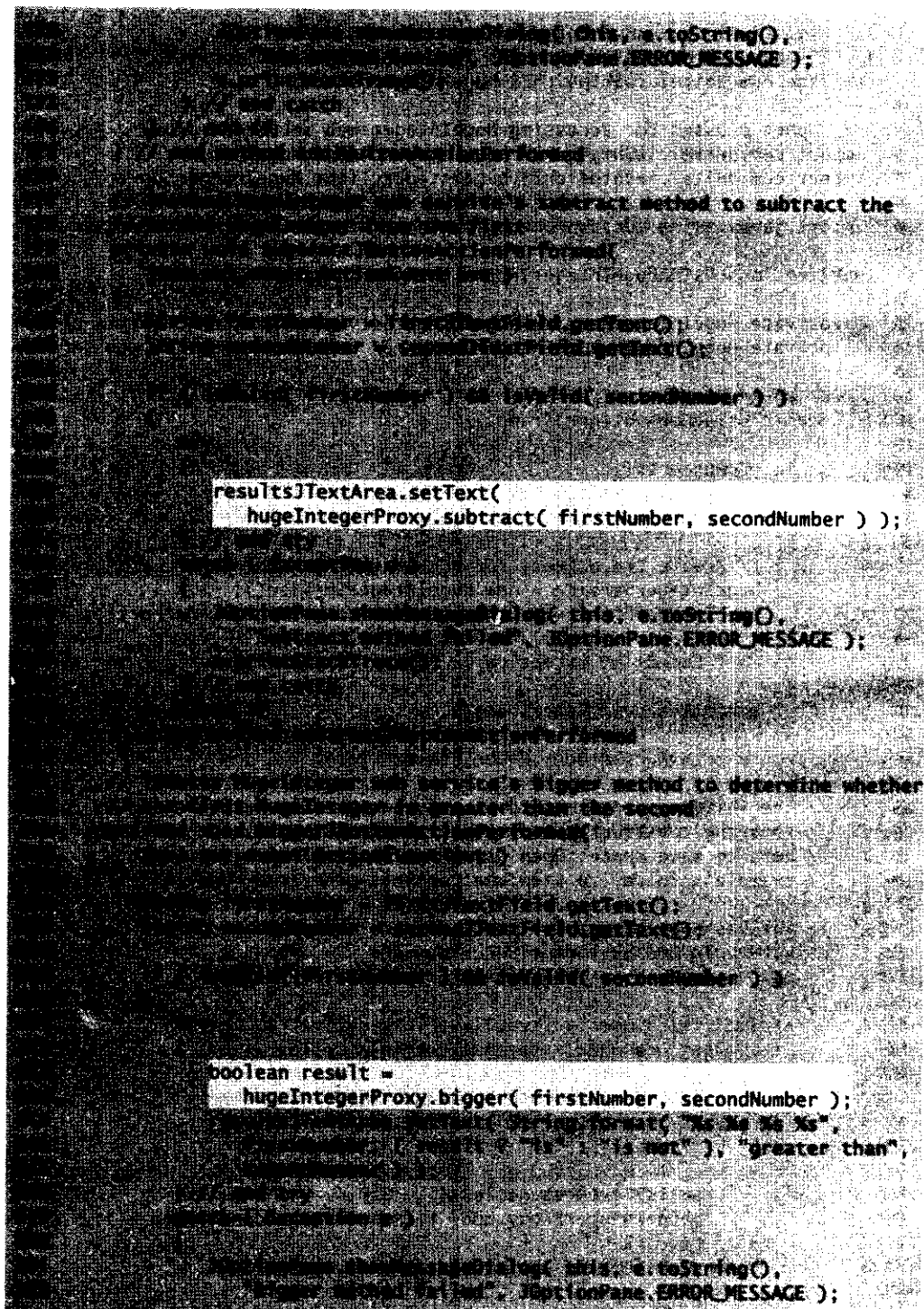


Fig. 28.11 | Client desktop application for the HugeInteger web service. (Part 2 of 6.)

```

303         e.printStackTrace();
304     } // end catch
305 } // end if
306 } // end method biggerButtonActionPerformed
307
308 // invoke HugeInteger web service's smaller method to determine
309 // whether the first HugeInteger is less than the second
310 private void smallerButtonActionPerformed(
311     javax.swing.ActionEvent evt )
312 {
313     String firstNumber = firstTextField.getText();
314     String secondNumber = secondTextField.getText();
315
316     if ( !isValid( firstNumber ) || !isValid( secondNumber ) )
317     {
318         // do nothing
319     }
320     else
321     {
322         boolean result =
323             hugeIntegerProxy.smaller( firstNumber, secondNumber );
324         resultsJTextArea.setText( String.format( "%s < %s",
325             firstNumber, ( result ? "is" : "is not" ), "less than",
326             secondNumber ) );
327     } // end try
328     catch ( Exception e )
329     {
330         JOptionPane.showMessageDialog( this, e.toString(),
331             "Button method failed", JOptionPane.ERROR_MESSAGE );
332         e.printStackTrace();
333     } // end catch
334 } // end if
335 } // end method smallerButtonActionPerformed
336
337 // invoke HugeInteger web service's equals method to determine
338 // whether the first HugeInteger is equal to the second
339 private void equalsButtonActionPerformed(
340     javax.swing.ActionEvent evt )
341 {
342     String firstNumber = firstTextField.getText();
343     String secondNumber = secondTextField.getText();
344
345     if ( !isValid( firstNumber ) || !isValid( secondNumber ) )
346     {
347         // do nothing
348     }
349     else
350     {
351         boolean result =
352             hugeIntegerProxy.equals( firstNumber, secondNumber );
353         resultsJTextArea.setText( String.format( "%s == %s",
354             firstNumber, ( result ? "is" : "is not" ), "equal to",
355             secondNumber ) );
356     } // end try
357     catch ( Exception e )
358     {
359         // do nothing
360     }
361 }

```

Fig. 28.11 | Client desktop application for the HugeInteger web service. (Part 3 of 6.)



```

209         JOptionPane.showMessageDialog( this, e.getMessage(),
210             "Error: method failed", JOptionPane.ERROR_MESSAGE );
211         e.printStackTrace();
212         // end catch
213     }
214     // end if
215     // end method equalsNonConsecutiveFormed
216     // checks the size of a string to ensure that it is not too big
217     // to use as a BigInteger - numbers only digits in decimal
218     // form - number (partial string number )
219     // check String's length
220     if ( number.length() > 100 )
221     {
222         JOptionPane.showMessageDialog( this,
223             "BigIntegers must be <= 100 digits.", "BigInteger Overflow",
224             JOptionPane.ERROR_MESSAGE );
225         return false;
226     }
227     // and if
228     // look for nondigit characters in String
229     for ( char c : number.toCharArray() )
230     {
231         if ( !Character.isDigit( c ) )
232         {
233             JOptionPane.showMessageDialog( this,
234                 "There are nondigits in the String",
235                 "BigInteger Contains Nondigit Characters",
236                 JOptionPane.ERROR_MESSAGE );
237             return false;
238         }
239         // end if
240     }
241     // end for
242     return true; // number can be used as a BigInteger
243     // end method validate
244     // main method begins execution
245     public static void main( String args[] )
246     {
247         Java.awt.EventQueue.invokeLater(
248             new Runnable()
249             {
250                 public void run()
251                 {
252                     new UsingBigIntegerJFrame().setVisible( true );
253                 } // end method run
254             } // end anonymous inner class
255         ); // end call to java.awt.EventQueue.invokeLater
256     } // end method main
257     // Variables declaration - do not modify
258     private javax.swing.JButton addJButton;

```

Fig. 28.11 | Client desktop application for the HugeInteger web service. (Part 4 of 6.)

```

327 private javax.swing.JButton biggerButton;
328 private javax.swing.JLabel directionsLabel;
329 private javax.swing.JButton equalsButton;
330 private javax.swing.JTextField firstTextField;
331 private javax.swing.JScrollPane resultsScrollPane;
332 private javax.swing.JTextArea resultsJTextArea;
333 private javax.swing.JTextField secondTextField;
334 private javax.swing.JButton smallerButton;
335 private javax.swing.JButton subtractButton;
336 // End of variables declaration
337 } // end class UsingHugeIntegerJFrame
    
```

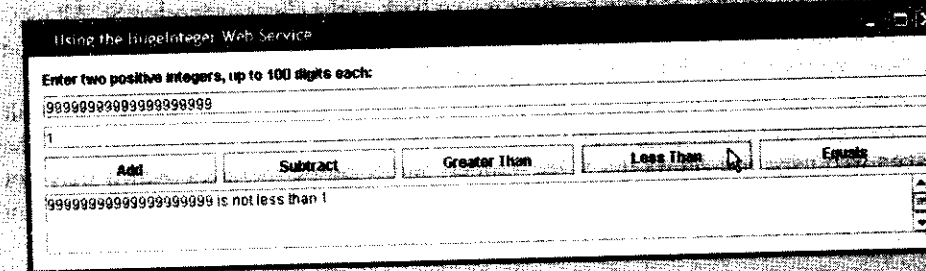
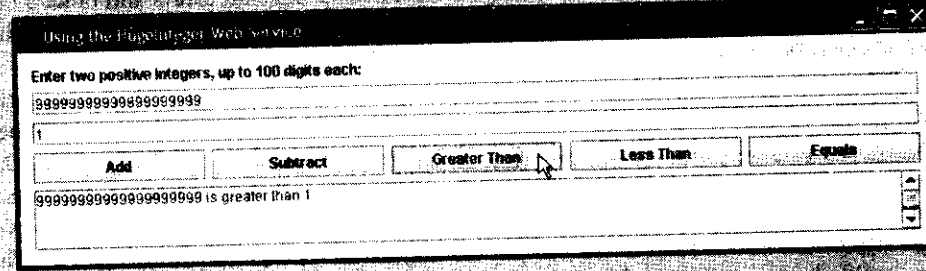
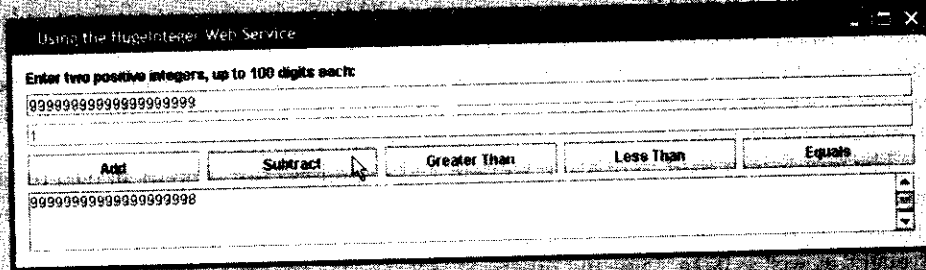
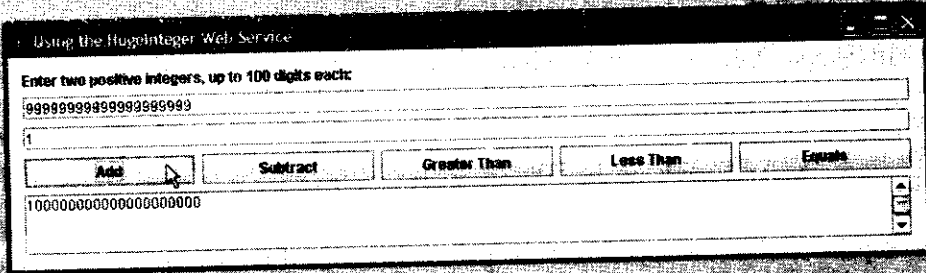
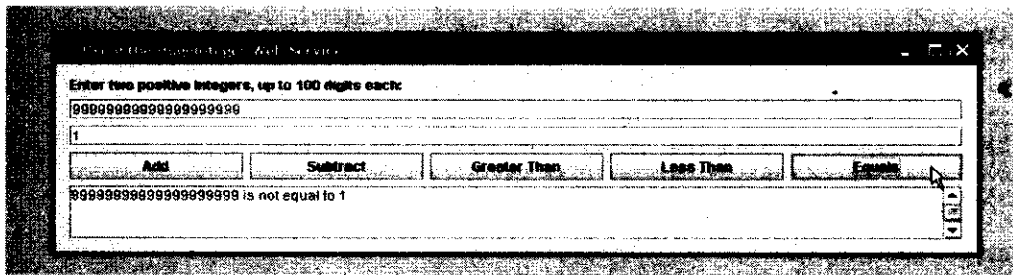


Fig. 28.11 | Client desktop application for the HugeInteger web service. (Part 5 of 6.)



**Fig. 28.11** | Client desktop application for the HugeInteger web service. (Part 6 of 6.)

## 28.5 SOAP

SOAP (Simple Object Access Protocol) is a platform-independent protocol that uses XML to facilitate remote procedure calls, typically over HTTP. SOAP is one common protocol for passing information between web service clients and web services. The protocol that transmits request-and-response messages is also known as the web service's **wire format** or **wire protocol**, because it defines how information is sent “along the wire.”

Each request and response is packaged in a **SOAP message** (also known as a **SOAP envelope**)—an XML “wrapper” containing the information that a web service requires to process the message. SOAP messages are written in XML so that they are platform independent. Many **firewalls**—security barriers that restrict communication among networks—are configured to allow HTTP traffic to pass through so that clients can browse websites on web servers behind firewalls. Thus, XML and HTTP enable computers on different platforms to send and receive SOAP messages with few limitations.

The wire format used to transmit requests and responses must support all data types passed between the applications. Web services also use SOAP for the many data types it supports. SOAP supports primitive types (e.g., `int`) and their wrapper types (e.g., `Integer`), as well as `Date`, `Time` and others. SOAP can also transmit arrays and objects of user-defined types (as you'll see in Section 28.8). For more SOAP information, visit [www.w3.org/TR/soap/](http://www.w3.org/TR/soap/).

When a program invokes a web method, the request and all relevant information are packaged in a SOAP message and sent to the server on which the web service resides. The web service processes the SOAP message's contents (contained in a SOAP envelope), which specify the method that the client wishes to invoke and the method's arguments. This process of interpreting a SOAP message's contents is known as **parsing a SOAP message**. After the web service receives and parses a request, the proper method is called with any specified arguments, and the response is sent back to the client in another SOAP message. The client-side proxy parses the response, which contains the result of the method call, and returns the result to the client application.

Figure 28.5 used the HugeInteger web service's Tester web page to show the result of invoking HugeInteger's `add` method with the values 999999999999999999 and 1. The Tester web page also shows the SOAP request and response messages (which were not previously shown). Figure 28.12 shows the SOAP messages in the Tester web page from Fig. 28.5 after the calculation. In the request message from Fig. 28.12, the text

```
<ns1:add>
<first>999999999999999999</first>
```

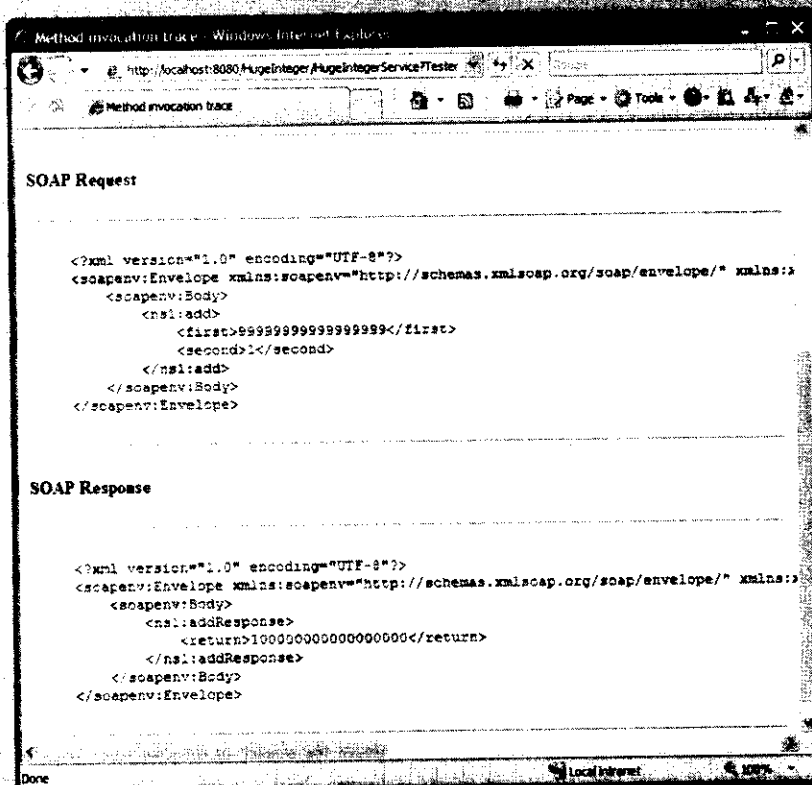
```
<second>1</second>
</ns1:add>
```

specifies the method to call (add), the method's arguments (first and second) and the arguments' values (9999999999999999 and 1). Similarly, the text

```
<ns1:addResponse>
<return>10000000000000000</return>
</ns1:addResponse>
```

from the response message in Fig. 28.12 specifies the return value of method add.

As with the WSDL for a web service, the SOAP messages are generated for you automatically, so you don't need to understand the details of SOAP or XML to take advantage of it when publishing and consuming web services.



**Fig. 28.12** | SOAP messages for the HugeInteger web service's add method as shown by the Sun Java System Application Server's Tester web page.

## 28.6 Session Tracking in Web Services

Section 26.5 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.

### 28.6.1 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. 28.13) 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. 28.14). 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 will 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. Note that 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. 28.13) 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 is represented as "2 0". To create and deploy this web service, follow the steps presented in Sections 28.3.2–28.3.3 for the `HugeInteger` service.

```

1 // Fig. 28.13: Blackjack.java
2 // Blackjack web service that deals cards and evaluates hands
3 package com.deitel.iw3http4.ch28.blackjack;
4
5 import java.util.ArrayList;
6 import java.util.Random;
7 import javax.annotation.Resource;
8 import javax.jws.WebService;
9 import javax.jws.WebMethod;
10 import javax.jws.WebParam;
11 import javax.servlet.http.HttpSession;
12 import javax.servlet.http.HttpServletRequest;
13 import javax.xml.ws.WebServiceContext;
14 import javax.xml.ws.handler.MessageContext;
15
16 @WebService( name = "Blackjack", serviceName = "BlackjackService" )
17 public class Blackjack
18 {
19     // use @Resource to create a WebServiceContext for session tracking
20     private @Resource WebServiceContext webServiceContext;
21     private MessageContext messageContext; // used in session tracking
22     private HttpSession session; // stores attributes of the session
23
24     // deal one card
25     @WebMethod( operationName = "dealCard" )
26     public String dealCard()
27     {
28         String card = "";
29
30         ArrayList< String > deck =
31             ( ArrayList< String > ) session.getAttribute( "deck" );
32
33         card = deck.get( 0 ); // get top card of deck
34         deck.remove( 0 ); // remove top card of deck
35
36         return card;
37     } // end WebMethod dealCard
38
39     // shuffle the deck
40     @WebMethod( operationName = "shuffle" )
41     public void shuffle()
42     {
43         // obtain the HttpSession object to store deck for current client
44         messageContext = webServiceContext.getMessageContext();
45         session = ( ( HttpServletRequest ) messageContext.get(
46             MessageContext.SERVLET_REQUEST ) ).getSession();
47
48         // populate deck of cards
49         ArrayList< String > deck = new ArrayList< String >();
50
51         for ( int face = 1; face <= 13; face++ ) // loop through faces
52             for ( int suit = 0; suit <= 3; suit++ ) // loop through suits
53                 deck.add( face + " " + suit ); // add each card to deck

```

Fig. 28.13 | Blackjack web service that deals cards and evaluates hands. (Part 1 of 3.)

```

54
55 String tempCard; // holds card temporarily during swapping
56 Random randomObject = new Random(); // generates random numbers
57 int index; // index of randomly selected card
58
59 for ( int i = 0; i < deck.size(); i++ ) // shuffle
60 {
61     index = randomObject.nextInt( deck.size() - 1 );
62
63     // swap card at position i with randomly selected card
64     tempCard = deck.get( i );
65     deck.set( i, deck.get( index ) );
66     deck.set( index, tempCard );
67 } // end for
68
69 // add this deck to user's session
70 session.setAttribute( "deck", deck );
71 } // end WebMethod shuffle
72
73 // determine a hand's value
74 @WebMethod( operationName = "getHandValue" )
75 public int getHandValue( @WebParam( name = "hand" ) String hand )
76 {
77     // split hand into cards
78     String[] cards = hand.split( "\\t" );
79     int total = 0; // total value of cards in hand
80     int face; // face of current card
81     int aceCount = 0; // number of aces in hand
82
83     for ( int i = 0; i < cards.length; i++ )
84     {
85         // parse string and get first int in String
86         face = Integer.parseInt(
87             cards[ i ].substring( 0, cards[ i ].indexOf( " " ) ) );
88
89         switch ( face )
90         {
91             case 1: // if ace, increment aceCount
92                 ++aceCount;
93                 break;
94             case 11: // jack
95             case 12: // queen
96             case 13: // king
97                 total += 10;
98                 break;
99             default: // otherwise, add face
100                 total += face;
101                 break;
102         } // end switch
103     } // end for
104 } // end for

```

Fig. 28.13 | Blackjack web service that deals cards and evaluates hands. (Part 2 of 3.)

```

105 // calculate optimal use of aces
106 if ( aceCount > 0 )
107 {
108 // if possible, count one ace as 11
109 if ( total + 11 + aceCount - 1 <= 21 )
110 total += 11 + aceCount - 1;
111 else // otherwise, count all aces as 1
112 total += aceCount;
113 } // end if
114
115 return total;
116 } //end WebMethod getHandValue
117 } // end class Blackjack

```

**Fig. 28.13** | Blackjack web service that deals cards and evaluates hands. (Part 3 of 3.)

### Session Tracking in Web Services

The Blackjack web service client first calls method `shuffle` (lines 40–71) to shuffle the deck of cards. This method also places the deck of cards into an `HttpSession` object that is specific to the client that called `shuffle`. To use session tracking in a Web service, you must include code for the resources that maintain the session state information. In the past, you had to write the sometimes tedious code to create these resources. JAX-WS, however, handles this for you via the `@Resource` annotation. This annotation enables tools like Netbeans to “inject” complex support code into your class, thus allowing you to focus on your business logic rather than the support code. The concept of using annotations to add code that supports your classes is known as **dependency injection**. Annotations like `@WebService`, `@WebMethod` and `@WebParam` also perform dependency injection.

Line 20 injects a `WebServiceContext` object into your class. A `WebServiceContext` object enables a web service to access and maintain information for a specific request, such as session state. As you look through the code in Fig. 28.13, you’ll notice that we never create the `WebServiceContext` object. All of the code necessary to create it is injected into the class by the `@Resource` annotation. Line 21 declares a variable of interface type `MessageContext` that the web service will use to obtain an `HttpSession` object for the current client. Line 22 declares the `HttpSession` variable that the web service will use to manipulate the session state information.

Line 44 in method `shuffle` uses the `WebServiceContext` object that was injected in line 20 to obtain a `MessageContext` object. Lines 45–46 then use the `MessageContext` object’s `get` method to obtain the `HttpSession` object for the current client. Method `get` receives a constant indicating what to get from the `MessageContext`. In this case, the constant `MessageContext.SERVLET_REQUEST` indicates that we’d like to get the `HttpServletRequest` object for the current client. We then call method `getSession` to get the `HttpSession` object from the `HttpServletRequest` object.

Lines 49–70 generate an `ArrayList` representing a deck of cards, shuffle the deck and store the deck in the client’s session object. Lines 51–53 use nested loops to generate Strings in the form “*face suit*” to represent each possible card in the deck. Lines 59–67 shuffle the deck by swapping each card with another card selected at random. Line 70 inserts the `ArrayList` in the session object to maintain the deck between method calls from a particular client.

Lines 25–37 define method `dealCard` as a web method. Lines 30–31 use the session object to obtain the "deck" session attribute that was stored in line 70 of method `shuffle`. Method `getAttribute` takes as a parameter a `String` that identifies the Object to obtain from the session state. The `HttpSession` can store many Objects, provided that each has a unique identifier. Note that method `shuffle` must be called before method `dealCard` is called the first time for a client—otherwise, an exception occurs at line 33 because `getAttribute` returns `null` at lines 30–31. After obtaining the user's deck, `dealCard` gets the top card from the deck (line 33), removes it from the deck (line 34) and returns the card's value as a `String` (line 36). 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 74–116) 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 78 tokenizes 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 83–103 count the value of each card. Lines 86–87 retrieve the first integer—the face—and use that value in the `switch` statement (lines 89–102). 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 97). If the card is anything else, the method increases the total by that value (line 100).

Because an ace can have either of two values, additional logic is required to process aces. Lines 106–113 of method `getHandValue` 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 109 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 110 adjusts the total accordingly. Otherwise, line 112 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.

### 28.6.2 Consuming the Blackjack Web Service

The blackjack application in Fig. 28.14 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 proxy object (lines 46–47). In the GUI, each player has 11 `JLabels`—the maximum number of cards that can be dealt without automatically exceeding 21 (i.e., four aces, four twos and three threes). These `JLabels` are placed in an `ArrayList` of `JLabels`, (lines 59–82), so we can index the `ArrayList` during the game to determine the `JLabel` that will display a particular card image.

With JAX-WS 2.0, 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 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, lines 50–51 invoke the `BindingProvider`'s `getRequestContext` method to obtain the `RequestContext` object. Then the `RequestContext`'s `put` method is called 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.

```

1 // Fig. 28.14: BlackjackGameJFrame.java
2 // Blackjack game that uses the Blackjack Web Service
3 package com.deitel.iw3http4.ch28.blackjackclient;
4
5 import javax.swing.Color;
6 import java.util.ArrayList;
7 import javax.swing.ImageIcon;
8 import javax.swing.JLabel;
9 import javax.swing.JOptionPane;
10 import javax.xml.ws.BindingProvider;
11 import com.deitel.iw3http4.ch28.blackjackclient.Blackjack;
12 import com.deitel.iw3http4.ch28.blackjackclient.BlackjackService;
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     // enumeration 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     } // end enum GameStatus
32
33     // no-argument constructor
34     public BlackjackGameJFrame()
35     {
36         initComponents();
37

```

Fig. 28.14 | Blackjack game that uses the Blackjack web service. (Part 1 of 10.)



```

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();
48
49     // enable session tracking
50     ( ( BindingProvider ) blackjackProxy ).getRequestContext().put(
51         BindingProvider.SESSION_MAINTAIN_PROPERTY, true );
52 } // end try
53 catch ( Exception e )
54 {
55     e.printStackTrace();
56 } // end catch
57
58 // add JLabels to cardBoxes ArrayList for programmatic manipulation
59 cardboxes = new ArrayList< JLabel >();
60
61 cardboxes.add( 0, dealerCard1JLabel );
62 cardboxes.add( dealerCard2JLabel );
63 cardboxes.add( dealerCard3JLabel );
64 cardboxes.add( dealerCard4JLabel );
65 cardboxes.add( dealerCard5JLabel );
66 cardboxes.add( dealerCard6JLabel );
67 cardboxes.add( dealerCard7JLabel );
68 cardboxes.add( dealerCard8JLabel );
69 cardboxes.add( dealerCard9JLabel );
70 cardboxes.add( dealerCard10JLabel );
71 cardboxes.add( dealerCard11JLabel );
72 cardboxes.add( playerCard1JLabel );
73 cardboxes.add( playerCard2JLabel );
74 cardboxes.add( playerCard3JLabel );
75 cardboxes.add( playerCard4JLabel );
76 cardboxes.add( playerCard5JLabel );
77 cardboxes.add( playerCard6JLabel );
78 cardboxes.add( playerCard7JLabel );
79 cardboxes.add( playerCard8JLabel );
80 cardboxes.add( playerCard9JLabel );
81 cardboxes.add( playerCard10JLabel );
82 cardboxes.add( playerCard11JLabel );
83 } // end no-argument constructor
84
85 // play the dealer's hand
86 private void dealerPlay()
87 {
88     try
89     {

```

Fig. 28.14 | Blackjack game that uses the B1ackjack web service. (Part 2 of 10.)



```

90 // while the value of the dealer's hand is below 17
91 // the dealer must continue to take cards
92 String[] cards = dealerCards.split( "\t" );
93
94 // display dealer's cards
95 for ( int i = 0; i < cards.length; i++ )
96     displayCard( i, cards[ i ] );
97
98 while ( blackjackProxy.getHandValue( dealerCards ) < 17 )
99 {
100     String newCard = blackjackProxy.dealCard();
101     dealerCards += "\t" + newCard; // deal new card
102     displayCard( currentDealerCard, newCard );
103     ++currentDealerCard;
104     JOptionPane.showMessageDialog( this, "Dealer takes a card"
105         "Dealer's turn", JOptionPane.PLAIN_MESSAGE );
106 } // end while
107
108 int dealersTotal = blackjackProxy.getHandValue( dealerCards );
109 int playersTotal = blackjackProxy.getHandValue( playerCards );
110
111 // if dealer busted, player wins
112 if ( dealersTotal > 21 )
113 {
114     gameOver( GameState.WIN );
115     return;
116 } // end if
117
118 // if dealer and player are below 21
119 // higher score wins, equal scores is a push
120 if ( dealersTotal > playersTotal )
121     gameOver( GameState.LOSE );
122 else if ( dealersTotal < playersTotal )
123     gameOver( GameState.WIN );
124 else
125     gameOver( GameState.PUSH );
126 } // end try
127 catch ( Exception e )
128 {
129     e.printStackTrace();
130 } // end catch
131 } // end method dealerPlay
132
133 // displays the card represented by cardValue in specified JLabel
134 public void displayCard( int card, String cardValue )
135 {
136     try
137     {
138         // retrieve correct JLabel from cardBoxes
139         JLabel displayLabel = cardBoxes.get( card );
140

```

Fig. 28.14 | Blackjack game that uses the BBlackjack web service. (Part 3 of 10.)

```

141 // if string representing card is empty, display back of card
142 if ( cardValue.equals( "" ) )
143 {
144     displayLabel.setIcon( new ImageIcon( getClass().getResource(
145         "/com/deitel/iw3http4/ch28/blackjackclient/" +
146         "blackjack_images/cardback.png" ) ) );
147     return;
148 } // end if
149
150 // retrieve the face value of the card
151 String face = cardValue.substring( 0, cardValue.indexOf( " " ) );
152
153 // retrieve the suit of the card
154 String suit =
155     cardValue.substring( cardValue.indexOf( " " ) + 1 );
156
157 char suitLetter; // suit letter used to form image file
158
159 switch ( Integer.parseInt( suit ) )
160 {
161     case 0: // hearts
162         suitLetter = 'h';
163         break;
164     case 1: // diamonds
165         suitLetter = 'd';
166         break;
167     case 2: // clubs
168         suitLetter = 'c';
169         break;
170     default: // spades
171         suitLetter = 's';
172         break;
173 } // end switch
174
175 // set image for displayLabel
176 displayLabel.setIcon( new ImageIcon( getClass().getResource(
177     "/com/deitel/iw3http4/ch28/blackjackclient/blackjack_images/" +
178     face + suitLetter + ".png" ) ) );
179 } // end try
180 catch ( Exception e )
181 {
182     e.printStackTrace();
183 } // end catch
184 } // end method displayCard
185
186 // displays all player cards and shows appropriate message
187 public void gameOver( GameStatus winner )
188 {
189     String[] cards = dealerCards.split( "\t" );
190
191     // display blackjackProxy's cards
192     for ( int i = 0; i < cards.length; i++ )
193         displayCard( i, cards[ i ] );

```

Fig. 28.14 | Blackjack game that uses the BBlackjack web service. (Part 4 of 10.)

```

196 // display appropriate status image
197 if ( winner == GameStatus.WIN )
198     statusJLabel.setText( "You win!" );
199 else if ( winner == GameStatus.LOSE )
200     statusJLabel.setText( "You lose." );
201 else if ( winner == GameStatus.PUSH )
202     statusJLabel.setText( "It's a push." );
203 else // blackjack
204     statusJLabel.setText( "Blackjack!" );
205
206 // display final scores
207 int dealersTotal = blackjackProxy.getHandValue( dealerCards );
208 int playersTotal = blackjackProxy.getHandValue( playerCards );
209 dealerTotalJLabel.setText( "Dealer: " + dealersTotal );
210 playerTotalJLabel.setText( "Player: " + playersTotal );
211
212 // reset for new game
213 standJButton.setEnabled( false );
214 hitJButton.setEnabled( false );
215 dealJButton.setEnabled( true );
216 // end method gameOver
217
218 // The initComponents method is autogenerated by Netbeans and is called
219 // from the constructor to initialize the GUI. This method is not shown
220 // here to save space. Open BlackjackGameJFrame.java in this
221 // example's folder to view the complete generated code (lines 221-531)
222
223 // handles standJButton click
224 private void standJButtonActionPerformed(
225     java.awt.event.ActionEvent evt )
226 {
227     standJButton.setEnabled( false );
228     hitJButton.setEnabled( false );
229     dealJButton.setEnabled( true );
230     dealerPlay();
231 } // end method standJButtonActionPerformed
232
233 // handles hitJButton click
234 private void hitJButtonActionPerformed(
235     java.awt.event.ActionEvent evt )
236 {
237     // get player another card
238     String card = blackjackProxy.dealCard(); // deal new card
239     playerCards += " " + card; // add card to hand
240
241     // update GUI to display new card
242     displayCard( currentPlayerCard, card );
243     currentPlayerCard = card;
244
245     // determine new value of player's hand
246     int total = blackjackProxy.getHandValue( playerCards );

```

Fig. 28.14 | Blackjack game that uses the Blackjack web service. (Part 5 of 10.)

```

597     if ( total > 21 ) // player busts
598         gameOver( GameState.LOSE );
599     if ( total == 21 ) // player cannot take any more cards
600     {
601         hitJButton.setEnabled( false );
602         dealerPlay();
603     } // end if
604 } // end method hitJButtonActionPerformed
605
606 // handles dealJButton click
607 private void dealJButtonActionPerformed(
608     java.awt.event.ActionEvent evt )
609 {
610     String card; // stores a card temporarily until it's added to
611
612     // clear card images
613     for ( int i = 0; i < cardboxes.size(); i++ )
614         cardboxes.get( i ).setIcon( null );
615
616     statusJLabel.setText( "" );
617     dealerTotalJLabel.setText( "" );
618     playerTotalJLabel.setText( "" );
619
620     // create a new, shuffled deck on remote machine
621     blackjackProxy.shuffle();
622
623     // deal two cards to player
624     playerCards = blackjackProxy.dealCard(); // add first card to hand
625     displayCard( 11, playerCards ); // display first card
626     card = blackjackProxy.dealCard(); // deal second card
627     displayCard( 12, card ); // display second card
628     playerCards += "\t" + card; // add second card to hand
629
630     // deal two cards to blackjackProxy, but only show first
631     dealerCards = blackjackProxy.dealCard(); // add first card to hand
632     displayCard( 0, dealerCards ); // display first card
633     card = blackjackProxy.dealCard(); // deal second card
634     displayCard( 1, "" ); // display back of card
635     dealerCards += "\t" + card; // add second card to hand
636
637     standJButton.setEnabled( true );
638     hitJButton.setEnabled( true );
639     dealJButton.setEnabled( false );
640
641     // determine the value of the two hands
642     int dealersTotal = blackjackProxy.getHandValue( dealerCards );
643     int playersTotal = blackjackProxy.getHandValue( playerCards );
644
645     // if hands both equal 21, it is a push
646     if ( playersTotal == dealersTotal && playersTotal == 21 )
647         gameOver( GameState.PUSH );
648     else if ( dealersTotal == 21 ) // blackjackProxy has blackjack
649         gameOver( GameState.LOSE );

```

Fig. 28.14 | Blackjack game that uses the Blackjack web service. (Part 6 of 10.)



```

610     else if ( playersTotal == 21 ) // blackjack
611         gameOver( GameStatus.BLACKJACK );
612
613     // next card for blackjackProxy has index 2
614     currentDealerCard = 2;
615
616     // next card for player has index 13
617     currentPlayerCard = 13;
618 } // end method dealNextActionPerformed
619
620 // begins application execution
621 public static void main( String args[] )
622 {
623     java.awt.EventQueue.invokeLater(
624         new Runnable()
625         {
626             public void run()
627             {
628                 new BlackjackGameFrame().setVisible(true);
629             }
630         }
631     ); // end call to java.awt.EventQueue.invokeLater
632 } // end method main
633
634 // Variables declaration - do not modify
635 private javax.swing.JButton dealButton;
636 private javax.swing.JLabel dealerCard0Label;
637 private javax.swing.JLabel dealerCard1Label;
638 private javax.swing.JLabel dealerCard2Label;
639 private javax.swing.JLabel dealerCard3Label;
640 private javax.swing.JLabel dealerCard4Label;
641 private javax.swing.JLabel dealerCard5Label;
642 private javax.swing.JLabel dealerCard6Label;
643 private javax.swing.JLabel dealerCard7Label;
644 private javax.swing.JLabel dealerCard8Label;
645 private javax.swing.JLabel dealerCard9Label;
646 private javax.swing.JLabel dealerLabel;
647 private javax.swing.JLabel dealerTotalLabel;
648 private javax.swing.JButton hitButton;
649 private javax.swing.JLabel playerCard0Label;
650 private javax.swing.JLabel playerCard1Label;
651 private javax.swing.JLabel playerCard2Label;
652 private javax.swing.JLabel playerCard3Label;
653 private javax.swing.JLabel playerCard4Label;
654 private javax.swing.JLabel playerCard5Label;
655 private javax.swing.JLabel playerCard6Label;
656 private javax.swing.JLabel playerCard7Label;
657 private javax.swing.JLabel playerCard8Label;
658 private javax.swing.JLabel playerCard9Label;
659 private javax.swing.JLabel playerLabel;
660 private javax.swing.JLabel playerTotalLabel;
661

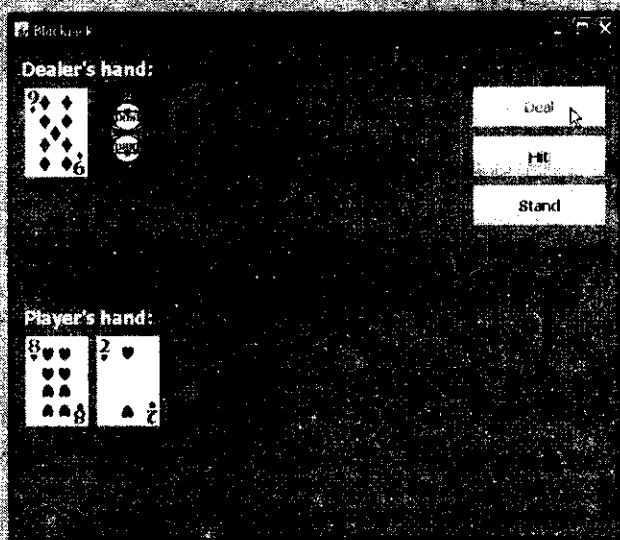
```

Fig. 28.14 | Blackjack game that uses the Blackjack web service. (Part 7 of 10.)

```

683 private javax.swing.JButton standButton;
684 private javax.swing.JLabel statusLabel;
685 // End of variables declaration
686 } // end class BlackjackGameFrame
    
```

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



b) Dealer and player hands after the user clicks Hit twice, then clicks Stand. In this case, the player wins.

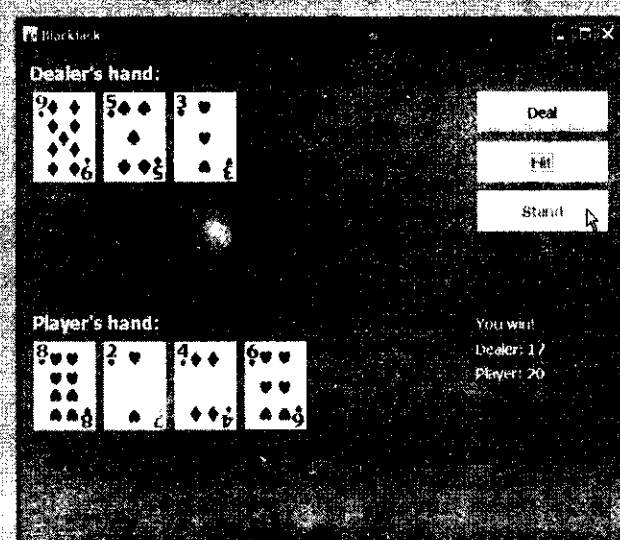
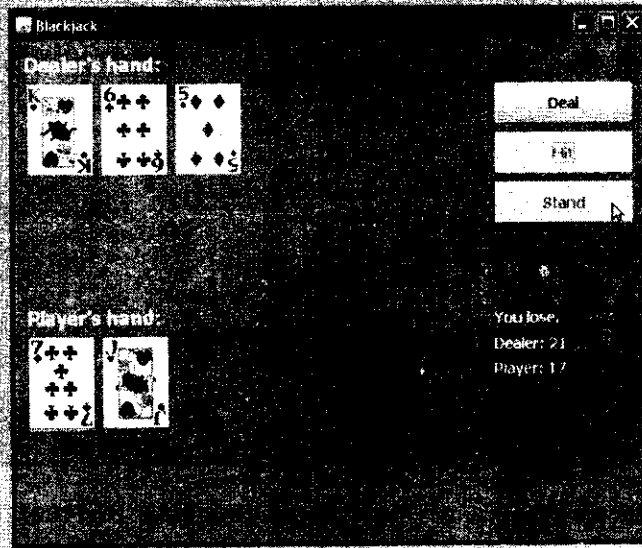


Fig. 28.14 | Blackjack game that uses the Blackjack web service. (Part 8 of 10.)



c) Dealer and player hands after the user clicks Hit based on the deal hand. In this case, the player loses.



d) Dealer and player hands after the user is dealt blackjack.

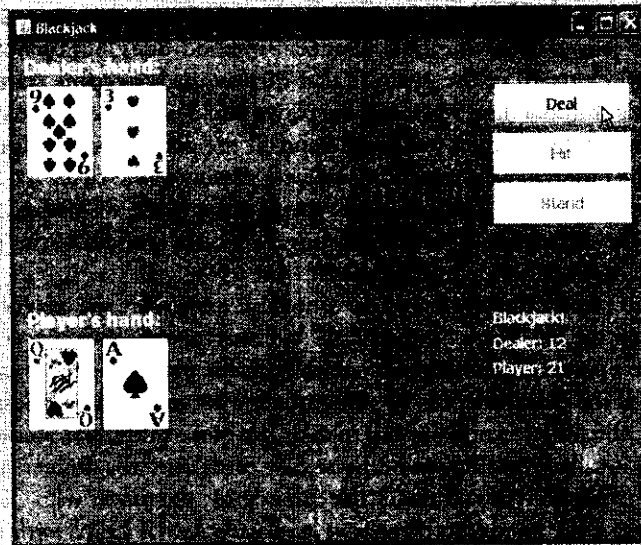
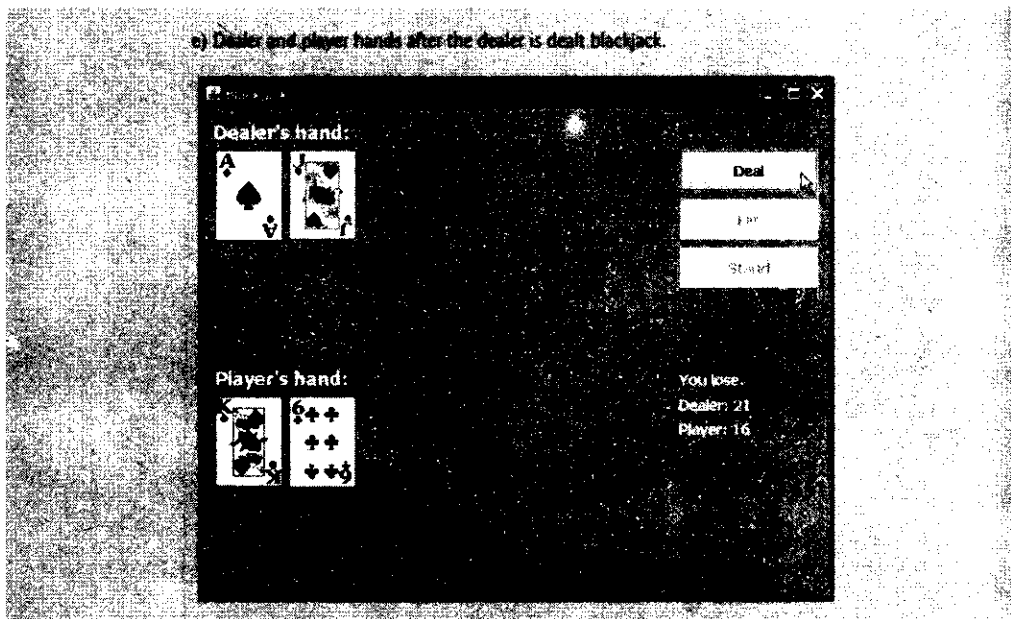


Fig. 28.14 | Blackjack game that uses the B1ackjack web service. (Part 9 of 10.)



**Fig. 28.14** | Blackjack game that uses the `Blackjack` web service. (Part 10 of 10.)

Method `gameOver` (lines 187–215) 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` enumeration (defined in lines 25–31). The enumeration represents whether the player tied, lost or won the game; its four members are `PUSH`, `LOSE`, `WIN` and `BLACKJACK`.

When the player clicks the `Deal` `JButton`, method `dealJButtonActionPerformed` (lines 567–618) clears all of the `JLabels` that display cards or game status information. Next, the deck is shuffled (line 581), and the player and dealer receive two cards each (lines 584–595). Lines 602–603 then total each hand. If the player and the dealer both obtain scores of 21, the program calls method `gameOver`, passing `GameStatus.PUSH` (line 607). If only the dealer has 21, the program passes `GameStatus.LOSE` to method `gameOver` (line 609). If only the player has 21 after the first two cards are dealt, the program passes `GameStatus.BLACKJACK` to method `gameOver` (line 611).

If `dealJButtonActionPerformed` does not call `gameOver`, the player can take more cards by clicking the `Hit` `JButton`, which calls `hitJButtonActionPerformed` in lines 543–564. Each time a player clicks `Hit`, the program deals the player one more card and displays it in the GUI. If the player exceeds 21, the game is over and the player loses. If the player has exactly 21, the player is not allowed to take any more cards, and method `dealerPlay` (lines 86–131) is called, causing the dealer to take cards until the dealer's hand has a value of 17 or more (lines 98–106). If the dealer exceeds 21, the player wins (line 114); otherwise, the values of the hands are compared, and `gameOver` is called with the appropriate argument (lines 120–125).

Clicking the `Stand` `JButton` indicates that a player does not want to be dealt another card. Method `standJButtonActionPerformed` (lines 533–540) disables the `Hit` and `Stand` buttons, enables the `Deal` button, then calls method `dealerPlay`.

Method `displayCard` (lines 134–184) 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 159–173) converts the number representing the suit to an integer and assigns the appropriate character to `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 176–178).

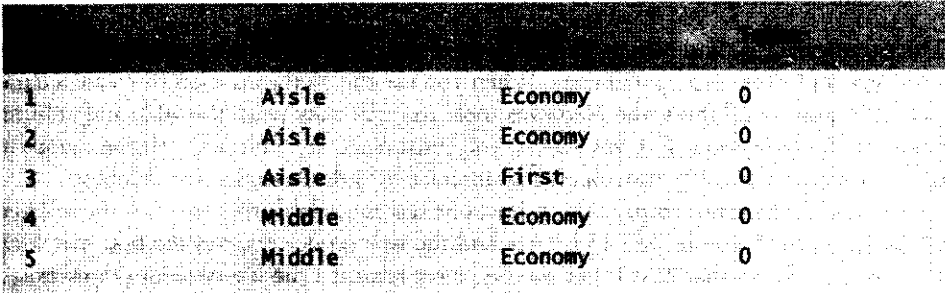
In this example, you learned how to set up a web service to support session handling so that you could keep track of each client's session state. You also learned how to indicate from a desktop client application that it wishes to take part in session tracking. You'll now learn how to access a database from a web service and how to consume a web service from a client web application.

## 28.7 Consuming a Database-Driven Web Service from a Web Application

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 prevalent, it is 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.

### 28.7.1 Configuring Java DB in Netbeans and Creating the Reservation Database

In this example, our web service uses a `Reservation` database containing a single table named `Seats` to locate a seat matching a client's request. To build the `Reservation` database, review the steps presented in Section 27.2.1 for building the `AddressBook` database. This chapter's examples directory contains a SQL script to build the `Seats` table and populate it with sample data. The sample data is shown in Fig. 28.15.



1	Aisle	Economy	0
2	Aisle	Economy	0
3	Aisle	First	0
4	Middle	Economy	0
5	Middle	Economy	0

**Fig. 28.15** | `Seats` table's data. (Part 1 of 2.)

6	Middle	First	0
7	Window	Economy	0
8	Window	Economy	0
9	Window	First	0
10	Window	First	0

**Fig. 28.15** | Seats table's data. (Part 2 of 2.)

### Creating the Reservation Web Service

You can now create a web service that uses the Reservation database (Fig. 28.16). The airline reservation web service has a single web method—`reserve` (lines 26–78)—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`. Note that the statements at lines 34–39 and lines 44–48 that query and update the database use objects of JDBC types `ResultSet` and `PreparedStatement`.



### Software Engineering Observation 28.1

Using `PreparedStatement`s to create SQL statements is highly recommended to secure against so-called SQL injection attacks in which executable code is inserted SQL code. The site [www.owasp.org/index.php/Preventing\\_SQL\\_Injection\\_in\\_Java](http://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 34–39 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. When you use the Netbeans tools to create a database table and its columns, the Netbeans tools automatically place the table and column names in double quotes. For this reason, you must place the table and column names in double quotes in the SQL statements that interact with the Reservation database.

If `resultSet` is not empty (i.e., at least one seat is available that matches the selected criteria), the condition in line 42 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 44) by accessing

resultSet's first column (i.e., resultSet.getInt(1)—the first column in the row). Then lines 45–48 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 49) to indicate that the reservation was successful. If there are no matching seats, or if an exception occurred, method reserve returns false (lines 52, 57, 62 and 75) to indicate that no seats matched the user's request.

```

1 // Fig. 28.16: Airline reservation web service
2 // Airline reservation web service
3 package com.deltasoft;
4
5 import java.sql.Connection;
6 import java.sql.PreparedStatement;
7 import java.sql.DriverManager;
8 import java.sql.ResultSet;
9 import java.sql.SQLException;
10 import java.util.ArrayList;
11 import javax.jws.WebService;
12 import javax.jws.WebParam;
13
14 @WebService(name = "AirlineReservation")
15 public class Reservation {
16     {
17         private static final String DATABASE_URL =
18             "jdbc:derby://localhost:1527/Reservation";
19         private static final String USERNAME = "jw3http4";
20         private static final String PASSWORD = "jw3http4";
21         private Connection connection;
22         private PreparedStatement lookupSeat;
23         private PreparedStatement reserveSeat;
24
25         // * WebMethod annotation
26         @WebMethod
27         public boolean reserveSeat(
28             @WebParam(name = "seatType") String seatType,
29             @WebParam(name = "classType") String classType)
30         {
31             try
32             {
33                 connection = DriverManager.getConnection(
34                     DATABASE_URL, USERNAME, PASSWORD);
35                 lookupSeat = connection.prepareStatement(
36                     "SELECT \"NUMBER\" FROM \"SEATS\" WHERE (\"TAKEN\" = 0)
37                     AND (\"LOCATION\" = ?) AND (\"CLASS\" = ?)");
38                 lookupSeat.setString(1, seatType);
39                 lookupSeat.setString(2, classType);
40                 ResultSet resultSet = lookupSeat.executeQuery();
41
42                 if (resultSet.next())
43                 {
44                     return true;
45                 }
46                 else
47                 {
48                     return false;
49                 }
50             }
51             catch (SQLException e)
52             {
53                 return false;
54             }
55         }
56     }
57 }

```

Fig. 28.16 | Airline reservation web service. (Part 1 of 2.)

```

40
41 // if requested seat is available, reserve it
42 if ( resultSet.next() )
43 {
44     int seat = resultSet.getInt( 1 );
45     reserveSeat = connection.prepareStatement(
46         "UPDATE \"SEATS\" SET \"TAKEN\"=1 WHERE \"NUMBER\"=? );
47     reserveSeat.setInt( 1, seat );
48     reserveSeat.executeUpdate();
49     return true;
50 } // end if
51
52 return false;
53 } // end try
54 catch ( SQLException e )
55 {
56     e.printStackTrace();
57     return false;
58 } // end catch
59 catch ( Exception e )
60 {
61     e.printStackTrace();
62     return false;
63 } // end catch
64 finally
65 {
66     try
67     {
68         // ...
69     } // end try
70     catch ( SQLException e )
71     {
72         e.printStackTrace();
73         return false;
74     } // end catch
75 } // end finally
76 } // end reservation method
77 } // end class Reservation

```

Fig. 28.16 | Airline reservation web service. (Part 2 of 2.)

### 28.7.2 Creating a Web Application to Interact with the Reservation Web Service

This section presents a `ReservationClient` web application that consumes the Reservation web service. The application allows users to select seats based on class ("Economy" or "First") and location ("Aisle", "Middle" or "Window"), then submit their requests to the airline reservation 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 26–27. We assume that you've already



read those chapters, and thus know how to build a web application's GUI, create event handlers and add properties to a web application's session bean (Section 27.2.1).

### Reserve.jsp

Reserve.jsp (Fig. 28.17) defines two DropDownLists and a Button. The seatTypeDropDown (lines 26–31) displays all the seat types from which users can select. The classTypeDropDownList (lines 32–37) provides choices for the class type. Users click the reserveButton (lines 38–42) to submit requests after making selections from the DropDownLists. The page also defines three Labels—instructionLabel (lines 22–25) to display instructions, errorLabel (lines 43–47) to display an appropriate message if no seat matching the user's selection is available and successLabel (lines 48–51) to indicate a successful reservation. The page bean file (Fig. 28.18) attaches event handlers to seatTypeDropDown, classTypeDropDown and reserveButton.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <!-- Fig. 28.17 Reserve.jsp -->
4  <!-- JSP that allows a user to select a seat -->
5  <jsp:root version="1.2"
6  xmlns:f="http://java.sun.com/jsp/core"
7  xmlns:h="http://java.sun.com/jsp/html"
8  xmlns:jsp="http://java.sun.com/JSP/Page"
9  xmlns:webuijsf="http://www.sun.com/webui/webuijsf">
10 <jsp:directive page contentType="text/html; charset=UTF-8"
11     pageEncoding="UTF-8"/>
12 <f:view>
13 <webuijsf:page binding="#{Reserve.page1}" id="page1">
14     <webuijsf:html binding="#{Reserve.html1}" id="html1">
15         <webuijsf:head binding="#{Reserve.head1}" id="head1">
16             <webuijsf:link binding="#{Reserve.link1}" id="link1"
17                 uri="/resources/stylesheets.css"/>
18         </webuijsf:head>
19         <webuijsf:body binding="#{Reserve.body1}" id="body1"
20             style="-rave-layout: grid">
21             <webuijsf:form binding="#{Reserve.form1}" id="form1">
22                 <webuijsf:label binding="#{Reserve.instructionLabel}"
23                     id="instructionLabel" style="left: 24px; top: 24px;
24                     position: absolute" text="Please select the seat type
25                     and class to reserve:"/>
26                 <webuijsf:dropdown binding="#{Reserve.seatTypeDropDown}"
27                     id="seatTypeDropDown" items=
28                     "#{Reserve.seatTypeDropDownDefaultOptions.options}"
29                     style="left: 24px; top: 24px; position: absolute"
30                     valueChangeListenerExpression=
31                     "#{Reserve.seatTypeDropDown_processValueChange}"/>
32                 <webuijsf:dropdown binding="#{Reserve.classTypeDropDown}"
33                     id="classTypeDropDown" items=
34                     "#{Reserve.classTypeDropDownDefaultOptions.options}"
35                     style="left: 365px; top: 24px; position: absolute"
36                     valueChangeListenerExpression=
37                     "#{Reserve.classTypeDropDown_processValueChange}"/>

```

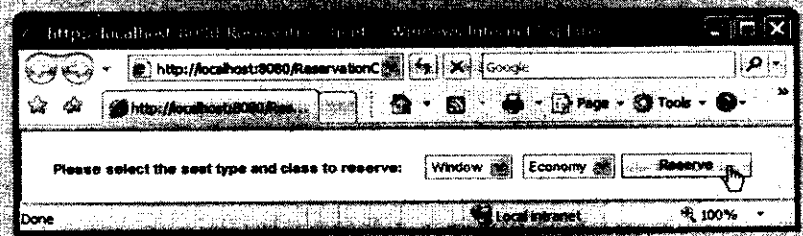
Fig. 28.17 | JSP that allows a user to select a seat. (Part I of 3.)

```

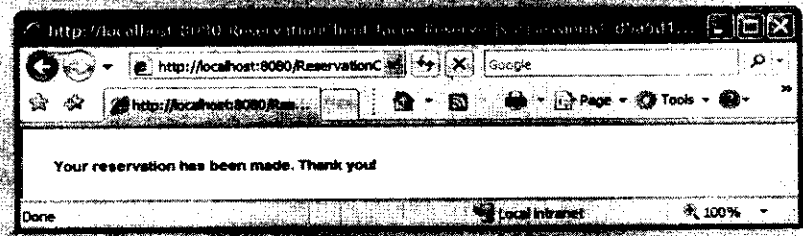
38 <webutil:action actionExpression=
39 <binding:binding id="reservebutton" binding=
40 <binding:binding id="reservebutton" style=
41 <binding:binding id="reservebutton" style=
42 <binding:binding id="reservebutton" style=
43 <binding:binding id="reservebutton" style=
44 <binding:binding id="reservebutton" style=
45 <binding:binding id="reservebutton" style=
46 <binding:binding id="reservebutton" style=
47 <binding:binding id="reservebutton" style=
48 <binding:binding id="reservebutton" style=
49 <binding:binding id="reservebutton" style=
50 <binding:binding id="reservebutton" style=
51 <binding:binding id="reservebutton" style=
52 <binding:binding id="reservebutton" style=
53 <binding:binding id="reservebutton" style=
54 <binding:binding id="reservebutton" style=
55 <binding:binding id="reservebutton" style=
56 <binding:binding id="reservebutton" style=
57 <binding:binding id="reservebutton" style=
58 <binding:binding id="reservebutton" style=
59 <binding:binding id="reservebutton" style=
60 <binding:binding id="reservebutton" style=
61 <binding:binding id="reservebutton" style=
62 <binding:binding id="reservebutton" style=
63 <binding:binding id="reservebutton" style=
64 <binding:binding id="reservebutton" style=
65 <binding:binding id="reservebutton" style=
66 <binding:binding id="reservebutton" style=
67 <binding:binding id="reservebutton" style=
68 <binding:binding id="reservebutton" style=
69 <binding:binding id="reservebutton" style=
70 <binding:binding id="reservebutton" style=
71 <binding:binding id="reservebutton" style=
72 <binding:binding id="reservebutton" style=
73 <binding:binding id="reservebutton" style=
74 <binding:binding id="reservebutton" style=
75 <binding:binding id="reservebutton" style=
76 <binding:binding id="reservebutton" style=
77 <binding:binding id="reservebutton" style=
78 <binding:binding id="reservebutton" style=
79 <binding:binding id="reservebutton" style=
80 <binding:binding id="reservebutton" style=
81 <binding:binding id="reservebutton" style=
82 <binding:binding id="reservebutton" style=
83 <binding:binding id="reservebutton" style=
84 <binding:binding id="reservebutton" style=
85 <binding:binding id="reservebutton" style=
86 <binding:binding id="reservebutton" style=
87 <binding:binding id="reservebutton" style=
88 <binding:binding id="reservebutton" style=
89 <binding:binding id="reservebutton" style=
90 <binding:binding id="reservebutton" style=
91 <binding:binding id="reservebutton" style=
92 <binding:binding id="reservebutton" style=
93 <binding:binding id="reservebutton" style=
94 <binding:binding id="reservebutton" style=
95 <binding:binding id="reservebutton" style=
96 <binding:binding id="reservebutton" style=
97 <binding:binding id="reservebutton" style=
98 <binding:binding id="reservebutton" style=
99 <binding:binding id="reservebutton" style=
100 <binding:binding id="reservebutton" style=

```

a) Selecting a seat



b) Seat reserved successfully



c) Attempting to reserve a seat that is no longer available

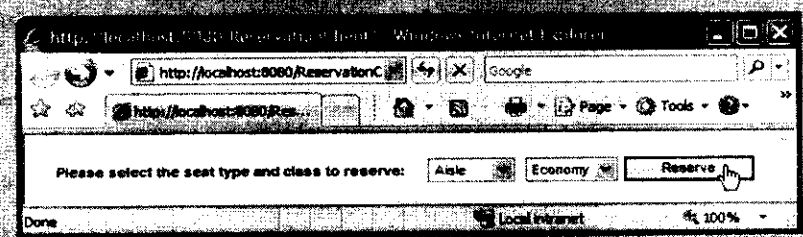


Fig. 28.17 | JSP that allows a user to select a seat. (Part 2 of 3.)

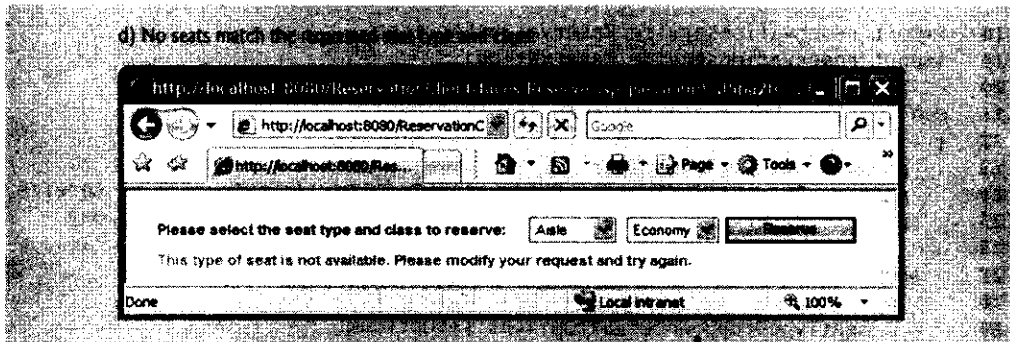


Fig. 28.17 | JSP that allows a user to select a seat. (Part 3 of 3.)

### Reserve.java

Figure 28.18 contains the page bean code that provides the logic for `Reserve.jsp`. As discussed in Section 26.5.2, the class that represents the page's bean extends `AbstractPageBean`. When the user selects a value in one of the `DropDownLists`, the corresponding event handler—`classTypeDropDown_processValueChange` (lines 262–267) or `seatTypeDropDown_processValueChange` (lines 270–275)—is called to set the session properties `seatType` and `classType`, which we added to the web application's session bean. The values of these properties are used as the arguments in the call to the web service's `reserve` method. When the user clicks `Reserve` in the JSP, the event handler `reserveButton_action` (lines 278–311) executes. Lines 282–284 use the proxy object (created in lines 38–39) to invoke the web service's `reserve` method, passing the selected seat type and class type as arguments. If `reserve` returns `true`, lines 288–293 hide the GUI components in the JSP and display the `successLabel` (line 292) to thank the user for making a reservation; otherwise, lines 297–302 ensure that the GUI components remain displayed and display the `errorLabel` (line 302) to notify the user that the requested seat type is not available and instruct the user to try again.

```

1 // Fig. 28.18: Reserve.java
2 // Page scope backing bean class for seat reservation client.
3 package com.detroittechinc.lessons.reservationclient;
4
5 import com.sun.faces.application.ActionListenerImpl;
6 import com.sun.faces.application.ResourceHandlerImpl;
7 import com.sun.faces.application.ResourceHandlerImpl.ResourceHandlerImpl;
8 import com.sun.faces.application.ResourceHandlerImpl.ResourceHandlerImpl;
9 import com.sun.faces.application.ResourceHandlerImpl.ResourceHandlerImpl;
10 import com.sun.faces.application.ResourceHandlerImpl.ResourceHandlerImpl;
11 import com.sun.faces.application.ResourceHandlerImpl.ResourceHandlerImpl;
12 import com.sun.faces.application.ResourceHandlerImpl.ResourceHandlerImpl;
13 import com.sun.faces.application.ResourceHandlerImpl.ResourceHandlerImpl;
14 import com.sun.faces.application.ResourceHandlerImpl.ResourceHandlerImpl;
15 import com.sun.faces.application.ResourceHandlerImpl.ResourceHandlerImpl;
16 import com.sun.faces.application.ResourceHandlerImpl.ResourceHandlerImpl;
17 import com.sun.faces.application.ResourceHandlerImpl.ResourceHandlerImpl;
18 import javax.faces.action.Action;
19 import javax.faces.event.ActionEvent;

```

Fig. 28.18 | Page scope backing bean class for seat reservation client. (Part 1 of 3.)

```

18 import reservationservice.ReservationService;
19 import reservationservice.Reservation;
20
21 public class Reserve extends AbstractPageBean
22 {
23     private int __placeholder;
24     private ReservationService reservationService; // reference to service
25     private Reservation reservationServiceProxy; // reference to proxy
26
27     private void _init() throws Exception
28     {
29         seatTypeDropDownDefaultOptions.setOptions(
30             new com.sun.webui.jsf.model.Option[] {
31                 new com.sun.webui.jsf.model.Option( "Aisle", "Aisle" ),
32                 new com.sun.webui.jsf.model.Option( "Middle", "Middle" ),
33                 new com.sun.webui.jsf.model.Option( "Window", "Window" ) } );
34         classTypeDropDownDefaultOptions.setOptions(
35             new com.sun.webui.jsf.model.Option[] {
36                 new com.sun.webui.jsf.model.Option( "Economy", "Economy" ),
37                 new com.sun.webui.jsf.model.Option( "First", "First" ) } );
38         reservationService = new ReservationService();
39         reservationServiceProxy = reservationService.getReservationPort();
40     } // end method
41
42     // Lines 42-260 of the autogenerated code have been removed to save
43     // space. The complete code is available in this example's folder.
44
45     // store selected class in session bean
46     public void classTypeDropDown_processValueChange(
47         ValueChangeEvent event )
48     {
49         getSessionBean1().setClassType(
50             ( String ) classTypeDropDown.getSelected() );
51     } // end method classTypeDropDown_processValueChange
52
53     // store selected seat type in session bean
54     public void seatTypeDropDown_processValueChange(
55         ValueChangeEvent event )
56     {
57         getSessionBean1().setSeatType(
58             ( String ) seatTypeDropDown.getSelected() );
59     } // end method seatTypeDropDown_processValueChange
60
61     // invoke the web service when the user clicks Reserve button
62     public String reserveButton_action()
63     {
64         try
65         {
66             boolean reserved = reservationServiceProxy.reserve(
67                 getSessionBean1().getSeatType(),
68                 getSessionBean1().getClassType() );
69         }
70     }

```

Fig. 28.18 | Page scope backing bean class for seat reservation client. (Part 2 of 3.)

```

280     if ( reserved ) // display successLabel; hide all others
281     {
282         instructionLabel.setRendered( false );
283         seatTypeDropDown.setRendered( false );
284         classTypeDropDown.setRendered( false );
285         reservation.setRendered( false );
286         successLabel.setRendered( true );
287         errorLabel.setRendered( false );
288     } // end if
289     else // display all but successLabel
290     {
291         instructionLabel.setRendered( true );
292         seatTypeDropDown.setRendered( true );
293         classTypeDropDown.setRendered( true );
294         reservation.setRendered( true );
295         successLabel.setRendered( false );
296         errorLabel.setRendered( true );
297     } // end else
298 } // end try
299 catch ( Exception e )
300 {
301     e.printStackTrace();
302 } // end catch
303
304 return null;
305 } // and method reservationAction
306 } // end class Reserve

```

Fig. 28.18 | Page scope backing bean class for seat reservation client. (Part 3 of 3.)

## 28.8 Passing an Object of a User-Defined Type to a Web Service

The web methods we've demonstrated so far each receive and return only primitive values or Strings. Web services also can receive and return objects of user-defined types—known as **custom types**. This section presents an `EquationGenerator` web service that generates random arithmetic questions of type `Equation`. The client is a math-tutoring desktop application in which the user selects the type of mathematical question to attempt (addition, subtraction or multiplication) and the skill level of the user—level 1 uses one-digit numbers in each question, level 2 uses two-digit numbers and level 3 uses three-digit numbers. The client passes this information to the web service, which then generates an `Equation` consisting of random numbers with the proper number of digits. The client application receives the `Equation`, displays the sample question to the user in a Java application, allows the user to provide an answer and checks the answer to determine whether it is correct.

### *Serialization of User-Defined Types*

We mentioned earlier that all types passed to and from SOAP web services must be supported by SOAP. How, then, can SOAP support a type that is not even created yet? Custom types that are sent to or from a web service are serialized into XML format. This process is referred to as **XML serialization**. The process of serializing objects to XML and deserializing objects from XML is handled for you automatically.

**Requirements for User-Defined Types Used with Web Methods**

A class that is used to specify parameter or return types in web methods must meet several requirements:

1. It must provide a `public` default or no-argument constructor. When a web service or web service consumer receives an XML serialized object, the JAX-WS 2.0 Framework must be able to call this constructor when deserializing the object (i.e., converting it from XML back to a Java object).
2. Instance variables that should be serialized in XML format must have `public` `set` and `get` methods to access the `private` instance variables (recommended), or the instance variables must be declared `public` (not recommended).
3. Non-`public` instance variables that should be serialized must provide both `set` and `get` methods (even if they have empty bodies); otherwise, they are not serialized.

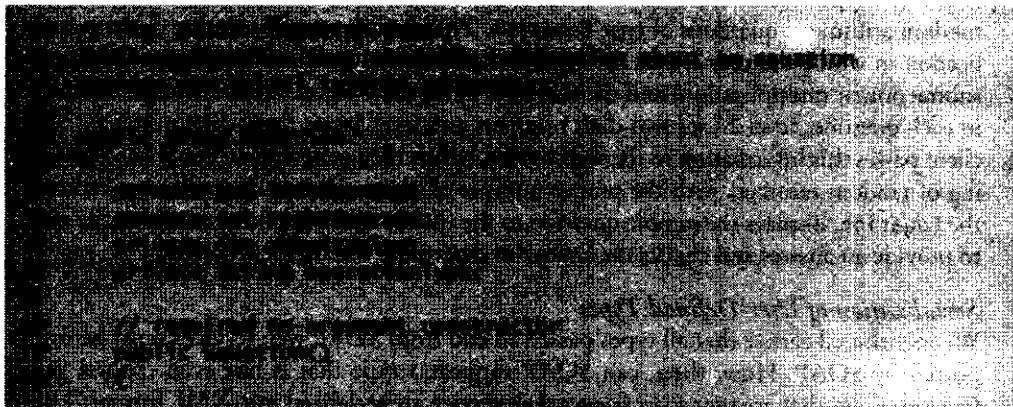
Any instance variable that is not serialized simply receives its default value (or the value provided by the no-argument constructor) when an object of the class is deserialized.

**Common Programming Error 28.3**

*A runtime error occurs if an attempt is made to deserialize an object of a class that does not have a default or no-argument constructor.*

**Defining Class Equation**

Figure 28.19 defines class `Equation`. Lines 18–31 define a constructor that takes three arguments—two `ints` representing the left and right operands and a `String` that represents the arithmetic operation to perform. The constructor sets the `leftOperand`, `rightOperand` and `operationType` instance variables, then calculates the appropriate result. The no-argument constructor (lines 13–16) calls the three-argument constructor (lines 18–31) and passes default values. We do not use the no-argument constructor explicitly, but the XML serialization mechanism uses it when objects of this class are deserialized. Because we provide a constructor with parameters, we must explicitly define the no-argument constructor in this class so that objects of the class can be passed to or returned from web methods.



**Fig. 28.19** | Class `Equation` that stores information about an equation. (Part 1 of 3.)



```

public class Equation {
    private int coefficient;
    private int constant;
    private int exponent;
    private String operator;
    private String operand;
    private String result;

    public Equation(int coefficient, int constant, int exponent, String operator, String operand) {
        this.coefficient = coefficient;
        this.constant = constant;
        this.exponent = exponent;
        this.operator = operator;
        this.operand = operand;
    }

    public Equation(int coefficient, int constant, int exponent) {
        this.coefficient = coefficient;
        this.constant = constant;
        this.exponent = exponent;
    }

    public Equation(int coefficient, int constant, int exponent, String operator) {
        this.coefficient = coefficient;
        this.constant = constant;
        this.exponent = exponent;
        this.operator = operator;
    }

    public Equation(int coefficient, int constant, int exponent, String operator, String operand, String result) {
        this.coefficient = coefficient;
        this.constant = constant;
        this.exponent = exponent;
        this.operator = operator;
        this.operand = operand;
        this.result = result;
    }

    public Equation(int coefficient, int constant, int exponent, String operator, String operand, String result, String description) {
        this.coefficient = coefficient;
        this.constant = constant;
        this.exponent = exponent;
        this.operator = operator;
        this.operand = operand;
        this.result = result;
        this.description = description;
    }

    public int getCoefficient() {
        return coefficient;
    }

    public int getConstant() {
        return constant;
    }

    public int getExponent() {
        return exponent;
    }

    public String getOperator() {
        return operator;
    }

    public String getOperand() {
        return operand;
    }

    public String getResult() {
        return result;
    }

    public String getDescription() {
        return description;
    }

    public void setCoefficient(int coefficient) {
        this.coefficient = coefficient;
    }

    public void setConstant(int constant) {
        this.constant = constant;
    }

    public void setExponent(int exponent) {
        this.exponent = exponent;
    }

    public void setOperator(String operator) {
        this.operator = operator;
    }

    public void setOperand(String operand) {
        this.operand = operand;
    }

    public void setResult(String result) {
        this.result = result;
    }

    public void setDescription(String description) {
        this.description = description;
    }
}

```

Fig. 28.19 | Class Equation that stores information about an equation. (Part 2 of 3.)



**Fig. 28.19** | Class Equation that stores information about an equation. (Part 3 of 3.)

Class Equation defines methods `getLeftHandSide` and `setLeftHandSide` (lines 41–44 and 77–80); `getRightHandSide` and `setRightHandSide` (lines 47–50 and 83–86); `getLeftOperand` and `setLeftOperand` (lines 53–56 and 89–92); `getRightOperand` and `setRightOperand` (lines 59–62 and 95–98); `getReturnValue` and `setReturnValue` (lines 65–68 and 101–104); and `getOperationType` and `setOperationType` (lines 71–74 and

107–110). The client of the web service does not need to modify the values of the instance variables. However, recall that a property can be serialized only if it has both a *get* and a *set* accessor, or if it is `public`. So we provided *set* methods with empty bodies for each of the class's instance variables. Method `getLeftHandSide` (lines 41–44) returns a `String` representing everything to the left of the equals (=) sign in the equation, and `getRightHandSide` (lines 47–50) returns a `String` representing everything to the right of the equals (=) sign. Method `getLeftOperand` (lines 53–56) returns the integer to the left of the operator, and `getRightOperand` (lines 59–62) returns the integer to the right of the operator. Method `getResultValue` (lines 65–68) returns the solution to the equation, and `getOperationType` (lines 71–74) returns the operator in the equation. The client in this example does not use the `rightHandSide` property, but we included it so future clients can use it.

### Creating the EquationGenerator Web Service

Figure 28.20 presents the EquationGenerator web service, which creates random, customized Equations. This web service contains only method `generateEquation` (lines 18–31), which takes two parameters—the mathematical operation (one of "+", "-", or "\*") and an `int` representing the difficulty level (1–3).

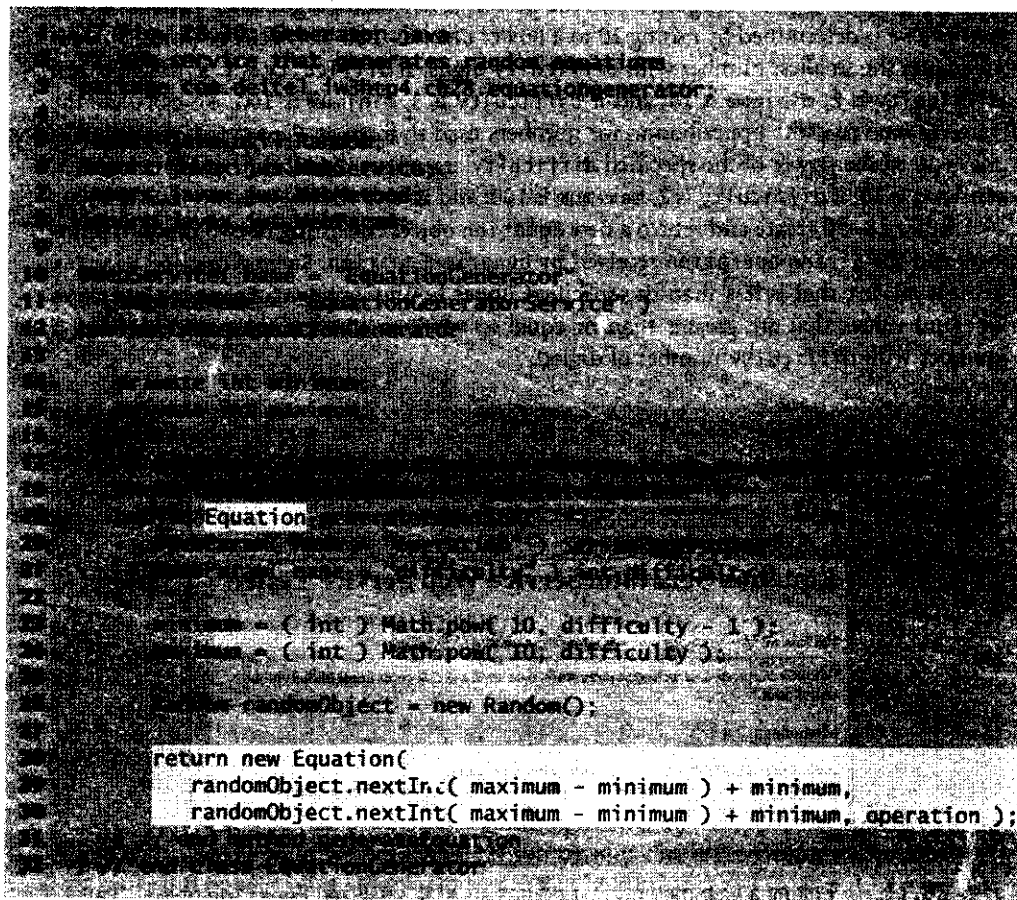


Fig. 28.20 | Web service that generates random equations.

*Testing the EquationGenerator Web Service*

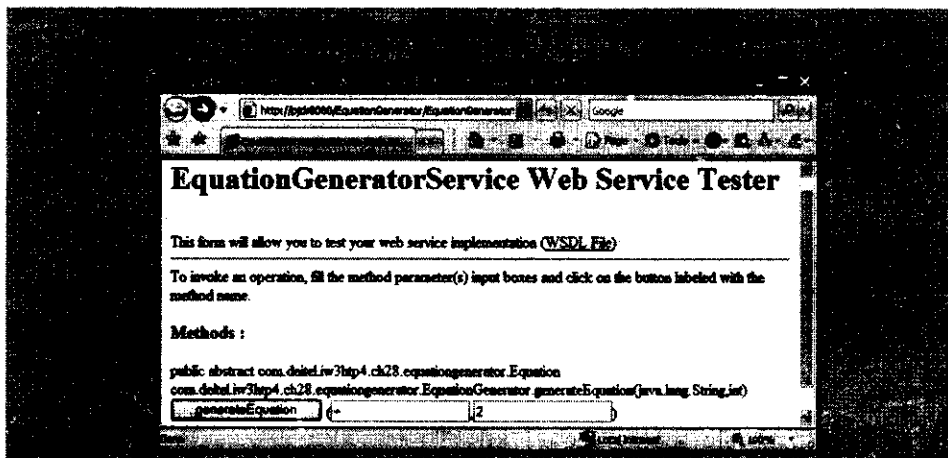
Figure 28.21 shows the result of testing the EquationGenerator service with the Tester web page. In *Part b* of the figure, note that the web method's return value is XML encoded. However, this example differs from previous ones in that the XML specifies the values for all the data of the returned XML serialized object returned. The proxy class receives this return value and deserializes it into an object of class Equation, then passes it to the client.

Note that an Equation object is *not* being passed between the web service and the client. Rather, the information in the object is being sent as XML-encoded data. Clients created using Java will take the information and create a new Equation object. Clients created on other platforms, however, may use the information differently. Readers creating clients on other platforms should check the web services documentation for the specific platform they are using, to see how their clients may process custom types.

*Details of the EquationGenerator Web Service*

Let's examine web method generateEquation more closely. Lines 23–24 of Fig. 28.20 define the upper and lower bounds of the random numbers that the method uses to generate an Equation. To set these limits, the program first calls static method pow of class Math—this method raises its first argument to the power of its second argument. Variable minimum's value is determined by raising 10 to a power one less than difficulty (line 23). This calculates the smallest number with difficulty digits. If difficulty is 1, minimum is 1; if difficulty is 2, minimum is 10; and if difficulty is 3, minimum is 100. To calculate the value of maximum (the upper bound for numbers used to form an Equation), the program raises 10 to the power of the specified difficulty argument (line 24). If difficulty is 1, maximum is 10; if difficulty is 2, maximum is 100; and if difficulty is 3, maximum is 1000.

Lines 28–30 create and return a new Equation object consisting of two random numbers and the String operation received by generateEquation. Random method nextInt returns an int that is less than the specified upper bound. generateEquation generates operand values that are greater than or equal to minimum but less than maximum (i.e., a number with difficulty number of digits).



**Fig. 28.21** | Testing a web method that returns an XML serialized Equation object. (Part 1 of 2.)

**generateEquation Method invocation**

---

**Method parameter(s)**

Type	Value
java.lang.String	+
int	2

---

**Method returned**

com.daiselw3http4.ch28.equationgenerator.Equation  
 'com.daiselw3http4.ch28.equationgenerator.Equation@d92923'

---

**SOAP Request**

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <nsl:generateEquation>
      <operation>+</operation>
      <difficulty>2</difficulty>
    </nsl:generateEquation>
  </soapenv:Body>
</soapenv:Envelope>
```

---

**SOAP Response**

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <nsl:generateEquationResponse>
      <return>
        <leftHandSide>23 + 27</leftHandSide>
        <leftOperand>23</leftOperand>
        <operationType>+</operationType>
        <returnValue>50</returnValue>
        <rightHandSide>50</rightHandSide>
        <rightOperand>27</rightOperand>
      </return>
    </nsl:generateEquationResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

**Fig. 28.21** | Testing a web method that returns an XML serialized Equation object. (Part 2 of 2.)

### *Consuming the EquationGenerator Web Service*

The Math Tutor application (Fig. 28.22) uses the EquationGenerator web service. The application calls the web service's generateEquation method to create an Equation object.

The tutor then displays the left-hand side of the Equation and waits for user input. Line 9 declares a GeneratorService instance variable that we use to obtain an EquationGenerator proxy object. Lines 10–11 declare instance variables of types EquationGenerator and Equation.

```
28.22: EquationGeneratorClientFrame.java
// generating program using web services to generate equations
// URL: http://www.ch28.com/equationgeneratorclient/
// @author: [author name]
// @version: 1.0
import javax.swing.*;
import java.io.*;

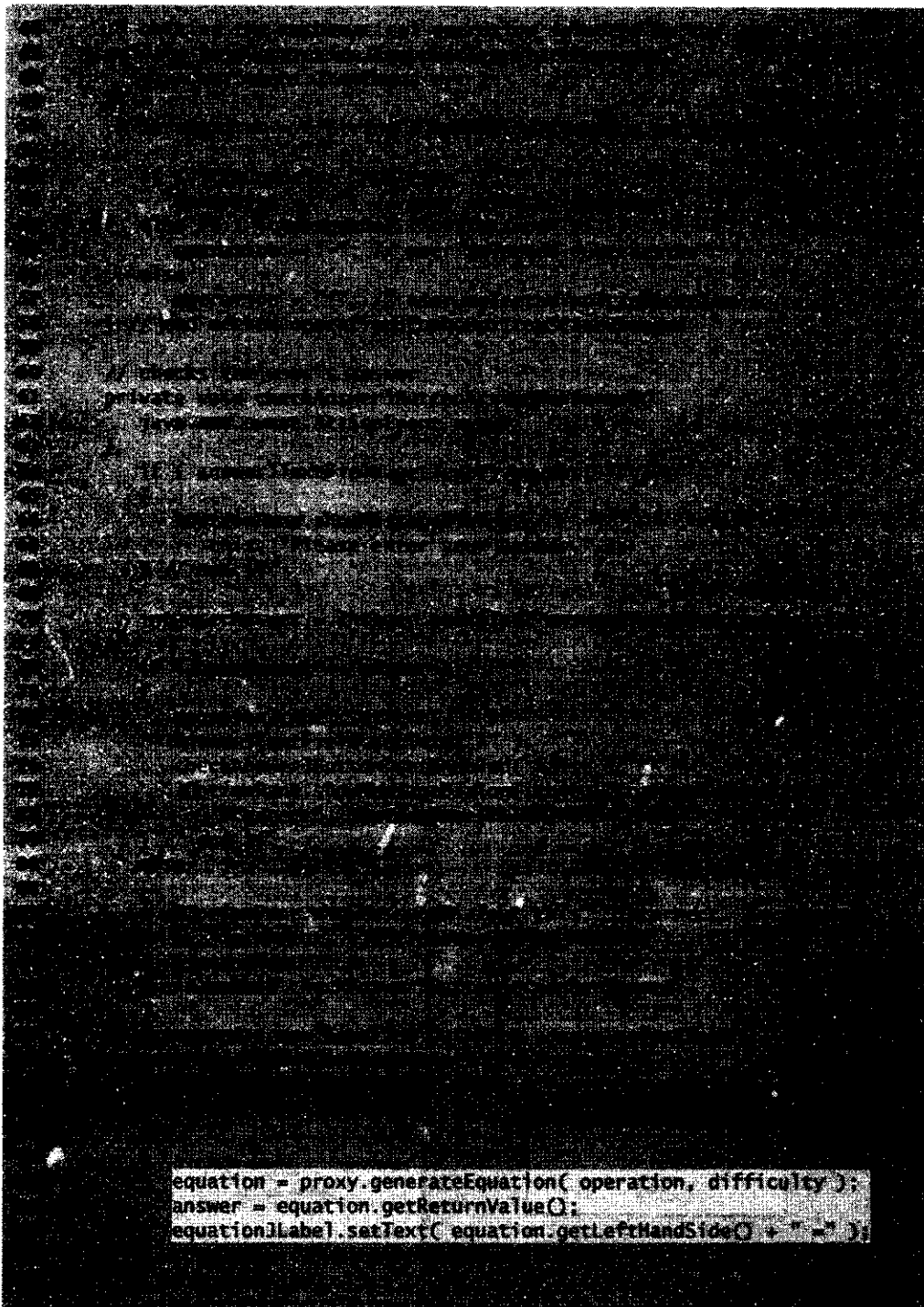
public class EquationGeneratorClientFrame extends JFrame {
    // declare EquationGeneratorService service: // used to obtain
    // private EquationGenerator proxy; // used to access the
    private EquationGenerator equation; // represents an equation
    private int answer; // the user's answer
    private String operator = "+"; // mathematical operation
    private int difficulty = 1; // 1, 2 or 3 digits in each number

    // no argument constructor
    public EquationGeneratorClientFrame() {
        initComponents();

        // create the objects for accessing the EquationGeneratorService
        service = new EquationGeneratorService();
        proxy = service.getEquationGeneratorPort();
        // and try
        try {
            // get an intStacktrace();
            // and catch
            // no argument constructors
        } catch (Exception ex) {
            // 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 EquationGeneratorClientFrame.java in this
            // example's folder to view the complete generated code (lines 37-156).
        }
    }
}
```

**Fig. 28.22** | Math tutoring application. (Part I of 4.)





**Fig. 28.22** | Math tutoring application. (Part 2 of 4.)

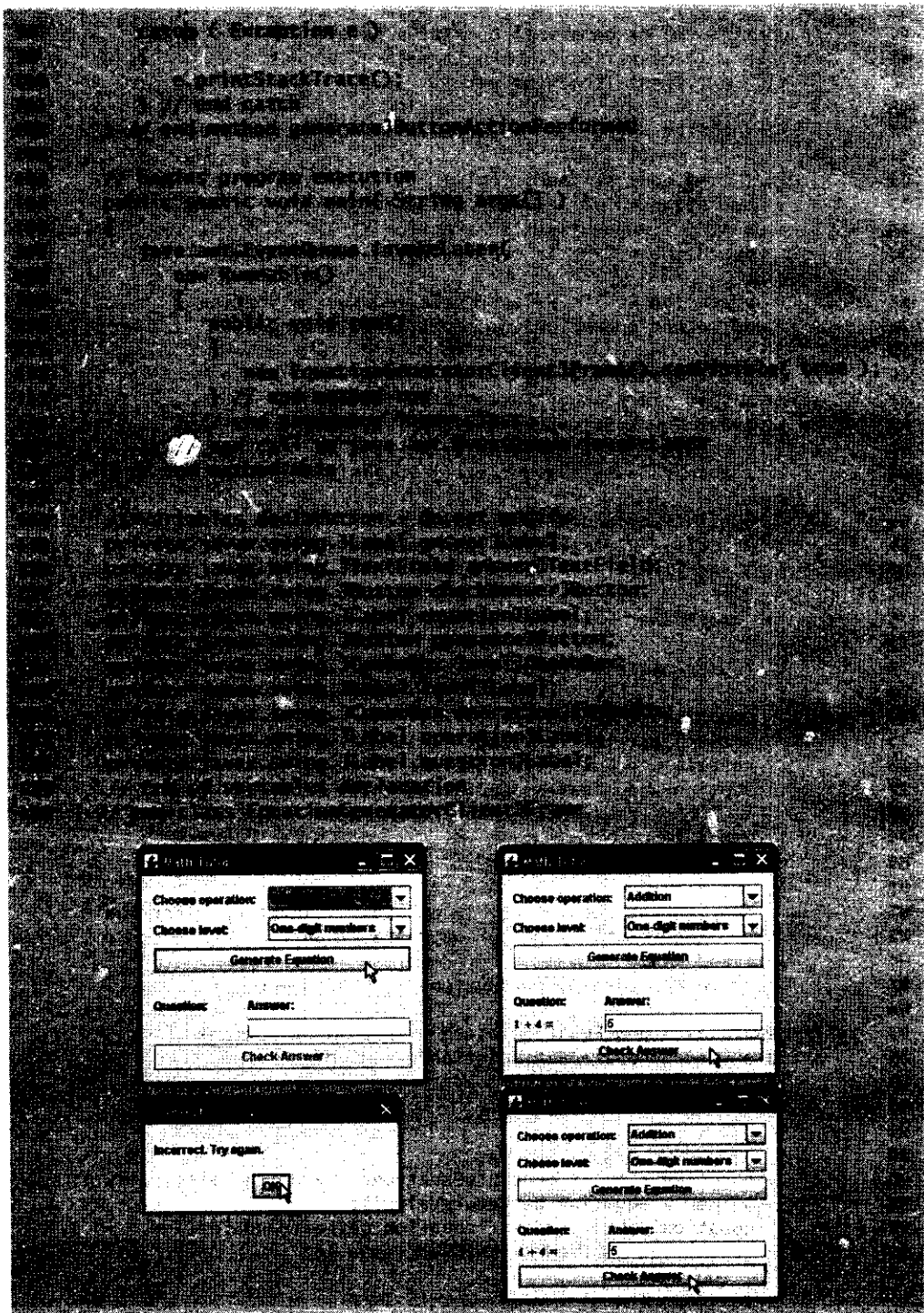
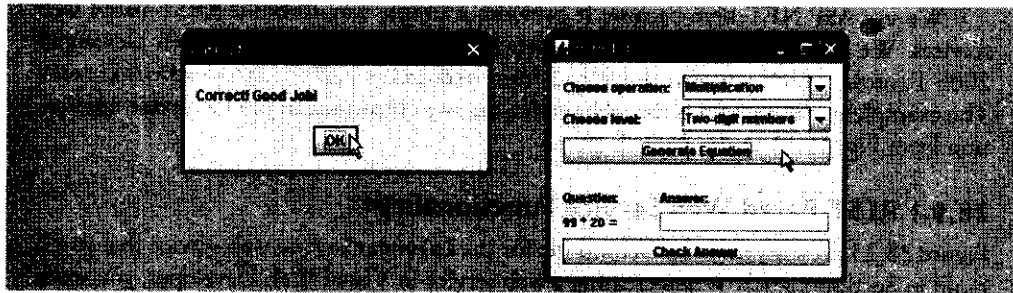


Fig. 28.22 | Math tutoring application. (Part 3 of 4.)



**Fig. 28.22** | Math tutoring application. (Part 4 of 4.)

After displaying an equation, the application waits for the user to enter an answer. The default setting for the difficulty level is **One-digit numbers**, but the user can change this by choosing a level from the **Choose level** JComboBox. Clicking any of the levels invokes `levelJComboBoxItemStateChanged` (lines 158–163), which sets the variable `difficulty` to the level selected by the user. Although the default setting for the question type is **Addition**, the user also can change this by selecting an operation from the **Choose operation** JComboBox. Doing so invokes `operationJComboBoxItemStateChanged` (lines 166–177), which sets the `String` operation to the appropriate mathematical symbol.

When the user clicks the **Generate Equation** JButton, method `generateButtonActionPerformed` (lines 207–221) invokes the `EquationGenerator` web service's `generateEquation` (line 212) method. After receiving an `Equation` object from the web service, the handler displays the left-hand side of the equation in `equationJLabel1` (line 214) and enables the `checkAnswerJButton` so that the user can submit an answer. When the user clicks the **Check Answer** JButton, method `checkAnswerJButtonActionPerformed` (lines 180–204) determines whether the user provided the correct answer.

## 28.9 REST-Based Web Services in ASP.NET

[*Note:* This section assumes you already know ASP.NET (Chapter 25).] In this section, we discuss how to build ASP.NET REST-based web services. **Representational State Transfer (REST)** (originally proposed in Roy Thomas Fielding's doctoral dissertation<sup>1</sup>) refers to an architectural style for implementing web services. Though REST is not a standard, RESTful web services are implemented using web standards, such as HTTP, XML and JSON. Each operation in a RESTful web service is easily identified by a unique URL. So, when the server receives a request, it immediately knows what operation to perform. Such web services can be invoked from a program or directly from a web browser by entering the URL in the browser's address field. In some cases, the results of a particular operation may be cached locally by the browser. This can make subsequent requests for the same operation faster by loading the result directly from the browser's cache.<sup>2</sup> Many Web 2.0 web services provide RESTful interfaces.<sup>3</sup>

1. Fielding, R. T. "Architectural Styles and the Design of Network-based Software Architectures." <<http://www.ics.ucl.edu/~fielding/pubs/dissertation/top.htm>>.
2. Costello, R. "REST Tutorial." *xFront*, 26 June 2002 <<http://www.xfront.com/REST.html>>.
3. Richardson, L. and S. Ruby. *RESTful Web Services*. O'Reilly, 2007.

We use ASP.NET here because it provides a simple way to build REST-based web services. We take advantage of the tools provided in Microsoft's Visual Web Developer 2005 Express, which you can download from [msdn.microsoft.com/vstudio/express](http://msdn.microsoft.com/vstudio/express). The example in this section is the web service that we consumed in our Calendar application from Fig. 15.11 in the Ajax chapter.

### 28.9.1 REST-Based Web Service Functionality

Figure 28.23 presents the code-behind file for the CalendarService web service that you'll build in Section 28.9.2. When creating a web service in Visual Web Developer, you work almost exclusively in the code-behind file. This web service is designed to give the client access to a database of events. A client can access all events that occur on a specific day using the `getItemsByDate` method or request a specific event using the `getItemById` method. In addition, the client can modify an event by calling the `Save` method.

```

1  " Fig. 28.23 CalendarService.vb
2  A REST-based event web service
3  Imports System.Web
4  Imports System.Web.Services
5  Imports System.Web.Services.Description
6  Imports System.Data.SqlClient
7  Imports System.Web.Services.WebResource
8
9  <WebServiceBinding xmlns="" name="CalendarService" type="System.Web.Services.WebResource" >
10 <WebServiceBinding xmlns="" name="CalendarService" type="System.Web.Services.WebResource" >
11 <Global.Microsoft.VisualBasic.CompilerServices.DesignerGenerated()
12 Public Class CalendarService
13     Inherits System.Web.Services.WebResource
14
15     ' variables used to access the database
16     Private calendarDatabase As New CalendarDataSet()
17     Private eventTable As Table
18     New CalendarService()
19
20     <WebMethod>
21     <HttpGet>
22     Public Sub getItemsByDate(ByVal date As String)
23         ' get all events for the specified date
24         ' insert the code here to get the events
25
26         ' get the response as a list of events
27         Dim response As List(Of CalendarEvent) = calendarDatabase.Events
28
29         ' get the response as a list of events
30         Dim response As List(Of CalendarEvent) = calendarDatabase.Events
31
32         ' get the response as a list of events
33         Dim response As List(Of CalendarEvent) = calendarDatabase.Events
34         HttpContext.Current.Response.Write(response) ' send to client
35     End Sub ' getItemsByDate
36

```

Fig. 28.23 | REST-based event web service. (Part 1 of 2.)

```

37 retrieve the list of events that occur on a specific date
38 <webMethod>Description: Gets a list of events for a given date.>
39 Public Sub GetItemsByDate(ByVal eventId As String)
40     eventsAdapter = CType(calendarDataSet.Events, IEnumerable)
41     Dim identification As String, string used to store the ID
42     Dim description As String, string used to store the description
43     Dim eventRow As DataRow, used to iterate over the dataset
44     Dim length As Integer = calendarDataSet.Events.Rows.Count
45     Dim itemObject As New Item()
46     Dim count As Integer
47
48     ' Insert the data into an array of Item objects
49     For Each eventRow In calendarDataSet.Events.Rows
50         identification = eventRow.Item("ID")
51         description = eventRow.Item("Description")
52         itemObject(count) = New Item(identification, description)
53         count += 1
54     Next
55
56     ' Convert the data to JSON and send it back to the client
57     Dim serializer As New JavaScriptSerializer()
58     Dim response As String = serializer.Serialize(itemObject)
59     HttpContext.Current.Response.ContentType = "application/json"
60     End Sub ' GetItemsByDate
61
62 ' modify the description of the event with the given ID
63 <webMethod>Description: Updates an event's description.>
64 Public Sub Save(ByVal id As String, ByVal descr As String)
65     eventsAdapter.Update(eventRow)
66     End Sub ' Save
67 End Class ' CalendarService

```

Fig. 28.23 | REST-based event web service. (Part 2 of 2.)

Lines 3–7 import all the necessary libraries for b. Lines 3–5 are generated by Visual Web Developer for every web service. Line 6 enables us to use capabilities for interacting with databases. Line 7 imports the `System.Web.Script.Serialization` namespace, which provides tools to convert .NET objects into JSON strings.

Line 9 contains a **WebService** attribute. Attaching this attribute to a web service class indicates that the class implements a web service and allows you to specify the web service's namespace. We specify `http://www.deitel.com` as the web service's namespace using the **WebService** attribute's **Namespace** property.

Visual Web Developer places line 10 in all newly created web services. This line indicates that the web service conforms to the **Basic Profile 1.1 (BP 1.1)** developed by the **Web Services Interoperability Organization (WS-I)**, a group dedicated to promoting interoperability among web services developed on different platforms with different programming languages. BP 1.1 is a document that defines best practices for various aspects of web service creation and consumption ([www.ws-i.org](http://www.ws-i.org)). Setting the **WebServiceBinding** attribute's **ConformsTo** property to `WsiProfiles.BasicProfile1_1` instructs Visual Web Developer to perform its “behind-the-scenes” work, such as generating WSDL file and the ASMX file

(which provides access to the web service) in conformance with the guidelines laid out in BP 1.1. For more information on web services interoperability and the Basic Profile 1.1, visit the WS-I web site at [www.ws-i.org](http://www.ws-i.org).

By default, each new web service class created in Visual Web Developer inherits from class `System.Web.Services.WebService` (line 13). Although a web service need not do this, class `WebService` provides members that are useful in determining information about the client and the web service itself. All methods in class `CalendarService` are tagged with the `WebMethod` attribute (lines 21, 38 and 63), which exposes a method so that it can be called remotely (similar to Java's `@WebMethod` annotation that you learned earlier in this chapter).

### *Accessing the Database*

Lines 16–18 create the `calendarDataSet` and `eventsTableAdapter` objects that are used to access the database. The classes `CalendarDataSet` and `CalendarDataSetTableAdapter`.`EventsTableAdapter` are created for you when you use Visual Web Developer's **DataSet Designer** to add a `DataSet` to a project. Section 28.9.3 discusses the steps for this.

Our database has one table called `Events` containing three columns—the numeric `ID` of an event, the `Date` on which the event occurs and the event's `Description`. Line 24 calls the method `FillById`, which fills the `calendarDataSet` with results of the query

```
SELECT    ID, Description
FROM      Events
WHERE     (ID = @id)
```

The parameter `@id` is replaced with the `id` that was passed from the client, which we pass as an argument to the `FillById` method. Lines 27–29 store the results of the query in the variable of class `Item`, which will be defined shortly. An `Item` object stores the `id` and the description of an event. The `id` and `description` are obtained by accessing the `ID` and `Description` values of the first row of the `calendarDataSet`.

Line 40 calls the method `FillByDate` which fills the `CalendarDataSet` with results of the query

```
SELECT    ID, Description
FROM      Events
WHERE     (Date = @date)
```

The parameter `@date` is replaced with the `eventDate` that was passed from the client, which we pass as an argument to the `FillByDate` method. Lines 49–54 iterate over the rows in the `calendarDataSet` and store the `ID` and `Description` values in an array of `Items`.

Line 65 calls method `UpdateDescription` which modifies the database with the `UPDATE` statement.

```
UPDATE    Events
SET       Description = @descr
WHERE     (ID = @id)
```

The parameters `@descr` and `@id` are replaced with arguments passed from the client to the `updateDescription` method.



### Responses Formatted as JSON

The web service uses JSON (discussed in Section 15.7) to pass data to the client. To return JSON data, we must first create a class to define objects which will be converted into JSON format. Figure 28.24 defines a simple `Item` class that contains `Description` and `ID` members, and two constructors. The `Description` and `ID` are declared as `Public` members so that `Item` objects can be properly converted to JSON.



#### Common Programming Error 28.4

*Properties and instance variables that are not public will not be serialized as part of an object's JSON representation.*

After the `Item` objects have been created and initialized with data from the database, lines 33 and 58 in Fig. 28.23 use the `JavaScriptSerializer`'s `Serialize` method to

```

Fig. 28.24 Item.vb
A simple class designed to be converted into JSON format.
Imports Microsoft.VisualBasic
Public Class Item
    Private Description As String
    Private IdValue As Integer
    ' Default constructor
    Public Sub New()
        Id = 0
    End Sub ' New

    ' constructor that initializes ID and description
    Public Sub New(id As Integer, description As String)
        Id = id
        Description = description
    End Sub ' New

    ' property that encapsulates the description
    Public Property Description As String
        Get
            Return description
        End Get
        Set(ByVal value As String)
            description = value
        End Set
    End Property ' Description

    ' Property that encapsulates the ID
    Public Property Id As Integer
        Get
            Return IdValue
        End Get
        Set(ByVal value As Integer)
            IdValue = value
        End Set
    End Property ' Id
End Class ' Item

```

**Fig. 28.24** | A simple class to create objects to be converted into JSON format.

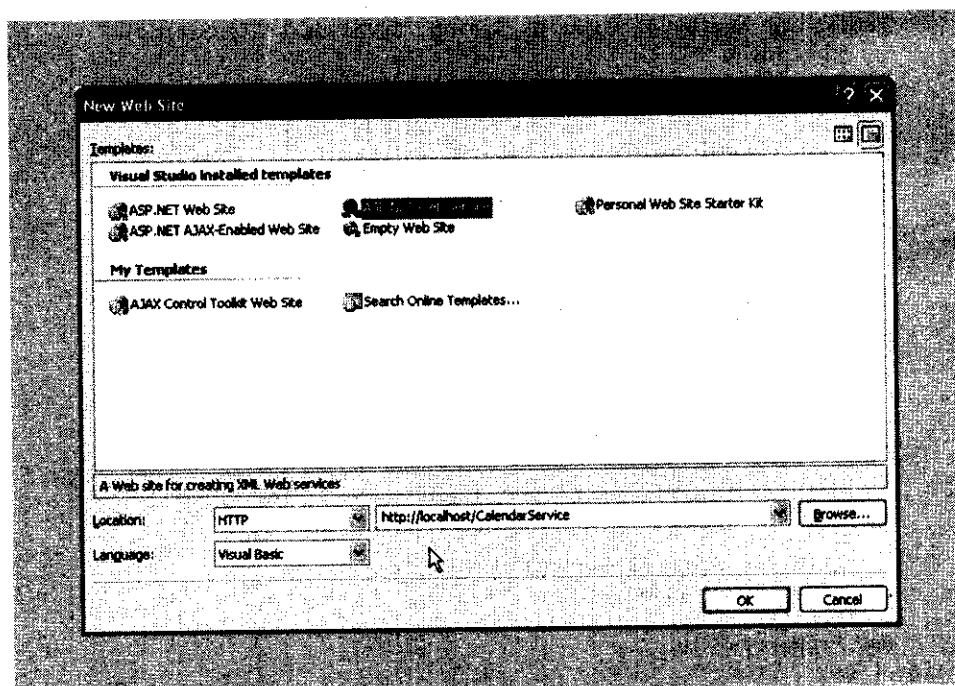
convert the objects into JSON strings. Then, lines 34 and 59 obtain a response object for the current client, using the `Current` property of the `HttpContext` object. Then we use this object to write the newly constructed JSON string as part of the response attribute, initiating the server response to the Ajax application in Fig. 15.11. To learn more about JSON visit our JSON Resource Center at [www.deitel.com/JSON](http://www.deitel.com/JSON).

### 28.9.2 Creating an ASP.NET REST-Based Web Service

We now show you how to create the `CalendarService` web service in Visual Web Developer. In the following steps, you'll create an **ASP.NET Web Service** project that executes on your computer's local IIS web server. Note that when you run this web service on your local computer the Ajax application from Fig. 15.11 can interact with the service only if it is served from your local computer. We discuss this at the end of this section. To create the `CalendarService` web service in Visual Web Developer, perform the following steps:

#### *Step 1: Creating the Project*

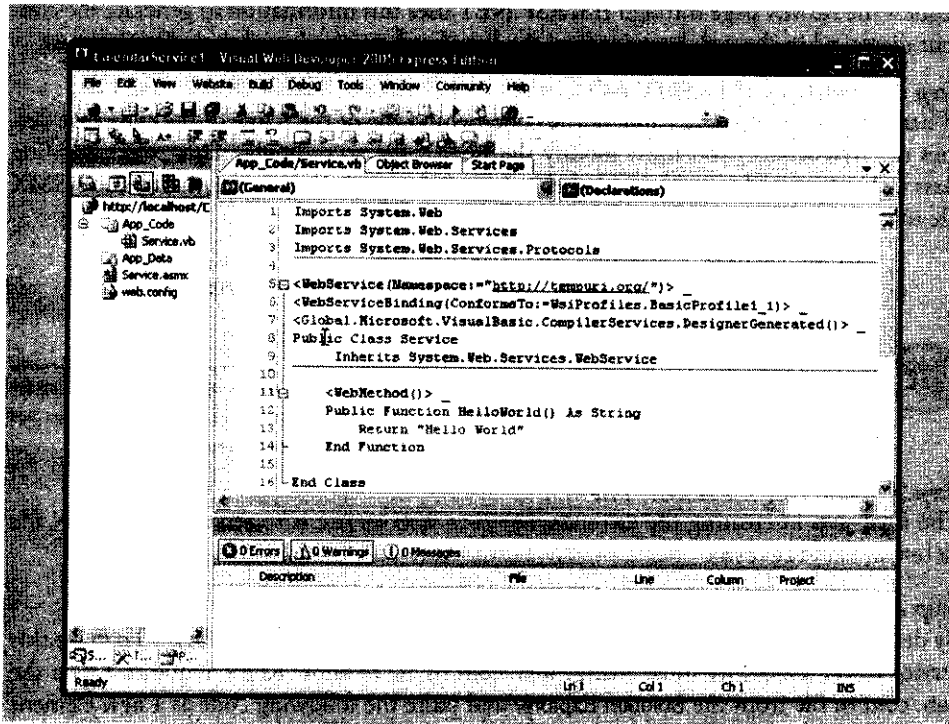
To begin, use Visual Web Developer create a project of type **ASP.NET Web Service**. Select **File > New Web Site...** to display the **New Web Site** dialog (Fig. 28.25). Select **ASP.NET Web Service** in the **Templates** pane. Select **HTTP** from the **Location** drop-down list to indicate that the files should be placed on a web server. By default, Visual Web Developer indicates that it will place the files on the local machine's IIS web server in a virtual directory named `WebSite` (`http://localhost/WebSite`). Replace the name `WebSite` with `CalendarService` for this example. Next, select **Visual Basic** from the **Language** drop-down list to indicate that you will use Visual Basic to build this web service.



**Fig. 28.25** | Creating an ASP.NET Web Service in Visual Web Developer

**Step 2: Examining the Newly Created Project**

After you create the project, you should see the code-behind file `Service.vb`, which contains code for a simple web service (Fig. 28.26). If the code-behind file is not open, it can be opened by double clicking the file in the **App\_Code** directory from the **Solution Explorer**. Visual Web Developer includes three `Imports` statements that are helpful for developing web services (lines 1–3). By default, a new code-behind file defines a class named `Service` that is marked with the `WebService` and `WebServiceBinding` attributes (lines 5–6). The class contains a sample web method named `HelloWorld` (lines 11–14). This method is a placeholder that you will replace with your own method(s).



**Fig. 28.26** | Code view of a web service.

**Step 3: Modifying and Renaming the Code-Behind File**

To create the `CalendarService` web service developed in this section, modify `Service.vb` by replacing the sample code provided by Visual Web Developer with the code from the `CalendarService` code-behind file (Fig. 28.23). Then rename the file `CalendarService.vb` (by right clicking the file in the **Solution Explorer** and choosing **Rename**). This code is provided in the examples directory for this chapter. You can download the examples from [www.deitel.com/books/iw3http4/](http://www.deitel.com/books/iw3http4/).

**Step 4 Creating an Item Class**

Select **File > New File...** to display the **Add New Item** dialog. Select **Class** in the **Templates** pane and change the name of the file to `Item.vb`. Then paste the code for `Item.vb` (Fig. 28.24) into the file.

**Step 5: Examining the ASMX File**

The **Solution Explorer** lists a `Service.asmx` file in addition to the code-behind file. A web service's ASMX page, when accessed through a web browser, displays information about the web service's methods and provides access to the web service's WSDL information. However, if you open the ASMX file on disk, you will see that it actually contains only

```
<%@ WebService Language="vb" CodeBehind="~/App_Code/Service.vb"
Class="Service" %>
```

to indicate the programming language in which the web service's code-behind file is written, the code-behind file's location and the class that defines the web service. When you request the ASMX page through IIS, ASP.NET uses this information to generate the content displayed in the web browser (i.e., the list of web methods and their descriptions).

**Step 6: Modifying the ASMX File**

Whenever you change the name of the code-behind file or the name of the class that defines the web service, you must modify the ASMX file accordingly. Thus, after defining class `CalendarService` in the code-behind file `CalendarService.vb`, modify the ASMX file to contain the lines

```
<%@ WebService Language="vb" CodeBehind=
"~/App_Code/CalendarService.vb" Class="CalendarService" %>
```

**Error-Prevention Tip 28.1**

*Update the web service's ASMX file appropriately whenever the name of a web service's code-behind file or the class name changes. Visual Web Developer creates the ASMX file, but does not automatically update it when you make changes to other files in the project.*

**Step 7: Renaming the ASMX File**

The final step in creating the `CalendarService` web service is to rename the ASMX file `CalendarService.asmx`.

**Step 8: Changing the Web.Config File to allow REST requests.**

By default ASP.NET web services communicate with the client using SOAP. To make this service REST-based, we must change `web.config` file to allow REST requests. Open the `web.config` file from the **Solution Explorer** and paste the following code as a new element in the `system.web` element.

```
<webServices>
  <protocols>
    <add name="HttpGet"/>
    <add name="HttpPost"/>
  </protocols>
</webServices>
```

**Step 9: Adding the System.Web.Extensions Reference**

The `JavaScriptSerializer` class that we use to generate JSON strings, is part of the Ajax Extensions package. You can find information on installing and downloading ASP.NET Ajax in Section 25.9. After you have installed ASP.NET Ajax, right click the project name in the solution explorer and select **Add Reference...** to display the **Add Reference** window. Select `System.Web.Extensions` from the `.NET` tab and click **OK**.

### 28.9.3 Adding Data Components to a Web Service

Next, you'll use Visual Web Developer's tools to configure a `DataSet` that allows our Web service to interact with the `Calendar.mdf` SQL Server 2005 Express database file. You can download `Calendar.mdf` with the rest of the code for this example at [www.deitel.com/books/iw3http4](http://www.deitel.com/books/iw3http4). You'll add a new `DataSet` to the project, then configure the `DataSet`'s `TableAdapter` using the **TableAdapter Configuration Wizard**. The wizard allows you to select the data source (`Calendar.mdf`) and to create the SQL statements necessary to support the database operations discussed in Fig. 28.23's description.

#### *Step 1: Adding a DataSet to the Project*

Add a `DataSet` named `CalendarDataSet` to the project. Right click the `App_Code` folder in the **Solution Explorer** and select **Add New Item...** from the pop-up menu. In the **Add New Item** dialog, select **DataSet**, specify `CalendarDataSet.xsd` in the **Name** field and click **Add**. This displays the `CalendarDataSet` in design view and opens the **TableAdapter Configuration Wizard**. When you add a `DataSet` to a project, the IDE creates appropriate `TableAdapter` classes for interacting with the database tables.

#### *Step 2: Selecting the Data Source and Creating a Connection*

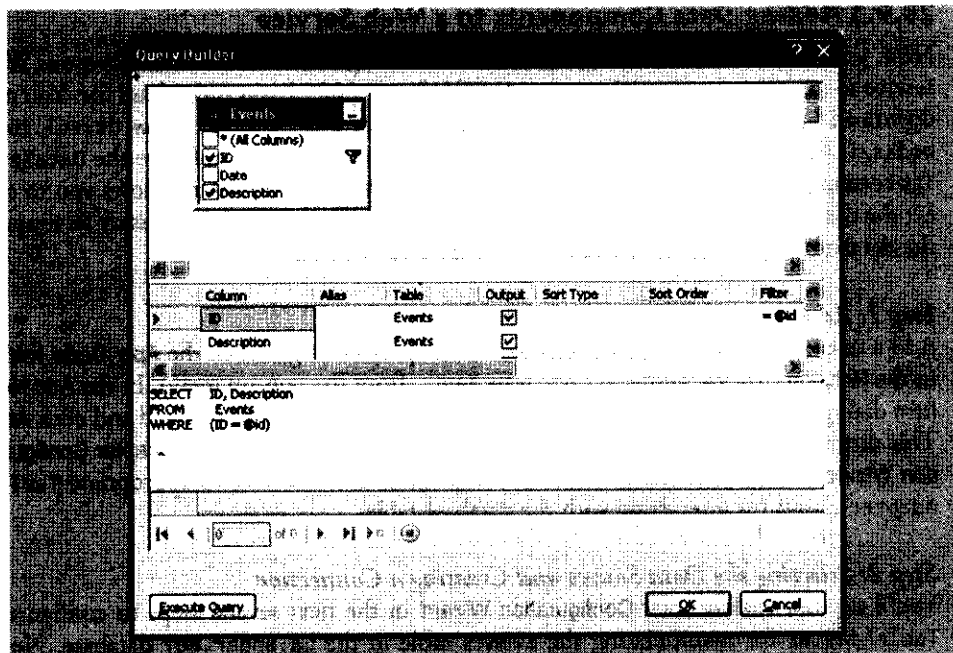
You'll use the **TableAdapter Configuration Wizard** in the next several steps to configure a `TableAdapter` for manipulating the `Events` table in the `Calendar.mdf` database. Now, you must select the database. In the **TableAdapter Configuration Wizard**, click the **New Connection...** button to display the **Add Connection** dialog. In this dialog, specify **Microsoft SQL Server Database File** as the **Data source**, then click the **Browse...** button to display the **Select SQL Server Database File** dialog. Locate `Calendar.mdf` on your computer, select it and click the **Open** button to return to the **Add Connection** dialog. Click the **Test Connection** button to test the database connection, then click **OK** to return to the **TableAdapter Configuration Wizard**. Click **Next >**, then click **Yes** when you are asked whether you would like to add the file to your project and modify the connection. Click **Next >** to save the connection string in the application configuration file.

#### *Step 3: Opening the Query Builder and Adding the Events Table from Calendar.mdf*

You must specify how the `TableAdapter` will access the database. In this example, you'll use SQL statements, so choose **Use SQL Statements**, then click **Next >**. Click **Query Builder...** to display the **Query Builder** and **Add Table** dialogs. Before building a SQL query, you must specify the table(s) to use in the query. The `Calendar.mdf` database contains only one table, named `Events`. Select this table from the **Tables** tab and click **Add**. Click **Close** to close the **Add Table** dialog.

#### *Step 4: Configuring a SELECT Query to Obtain a Specific Event*

Now let's create a query which selects an event with a particular ID. Select **ID** and **Description** from the `Events` table at the top of the **Query Builder** dialog. Next, specify the criteria for selecting seats. In the **Filter** column of the **ID** row specify `=@id` to indicate that this filter value also will be specified as a method argument. The **Query Builder** dialog should now appear as shown in Fig. 28.27. Click **OK** to close the **Query Builder** dialog. Click **Next >** to choose the names of the methods to generate. Name the `Fill` method `FillById`. Click the **Finish** button to generate this method.



**Fig. 28.27** | QueryBuilder dialog specifying a SELECT query that selects an event with a specific ID.

**Step 5: Adding Another Query to the EventsTableAdapter for the CalendarDataSet**

Now, you'll create an UPDATE query that modifies a description of a specific event. In the design area for the CalendarDataSet, click **EventsTableAdapter** to select it, then right click it and select **Add Query...** to display the **TableAdapter Query Configuration Wizard**. Select **Use SQL Statements** and click **Next >**. Select **Update** as the query type and click **Next >**. Clear the text field and click **Query Builder...** to display the **Query Builder** and **Add Table** dialogs. Then add the Events table as you did in *Step 3* and click **Close** to return to the **Query Builder** dialog.

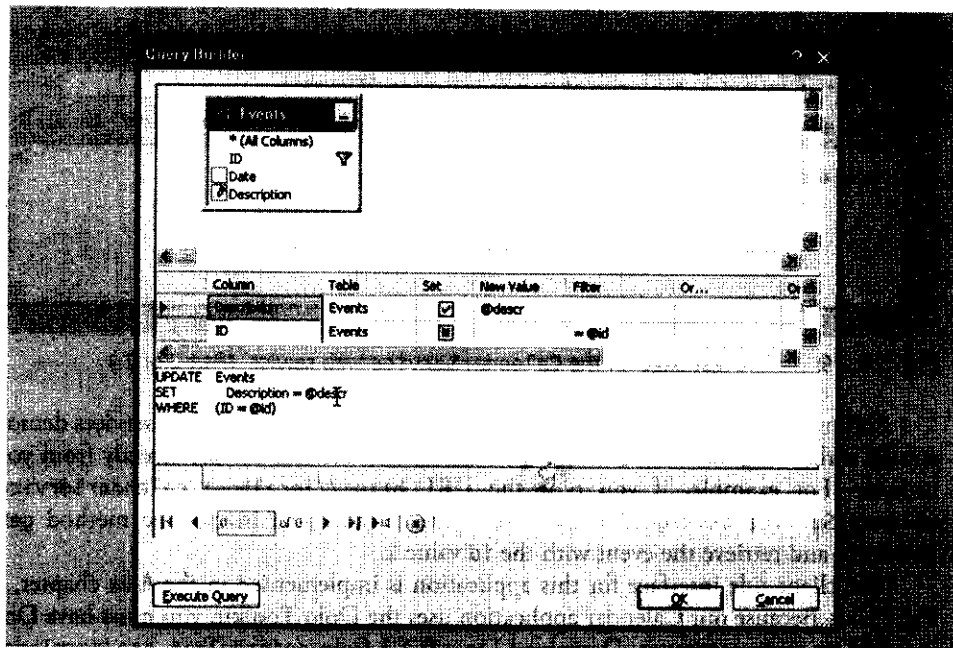
**Step 6: Configuring an UPDATE Statement to Modify a Description of a Specific Event**

In the **Query Builder** dialog, select the **Description** column from the **Events** table at the top of the dialog. In the middle of the dialog, place the **@descr** in the **New Value** column for the **Description** row to indicate that the new description will be specified as an argument to the method that implements this query. In the row below **Description**, select **ID** and specify **@id** as the **Filter** value to indicate that the ID will be specified as an argument to the method that implements this query. The **Query Builder** dialog should now appear as shown in Fig. 28.28. Click **OK** to return to the **TableAdapter Query Configuration Wizard**. Then click **Next >** to choose the name of the update method. Name the method **UpdateDescription**, then click **Finish** to close the **TableAdapter Query Configuration Wizard**.

**Step 7: Adding a getItemByDate Query**

Using similar techniques to *Steps 5–6*, add a query that selects all events that have a specified date. Name the query **FillByDate**.

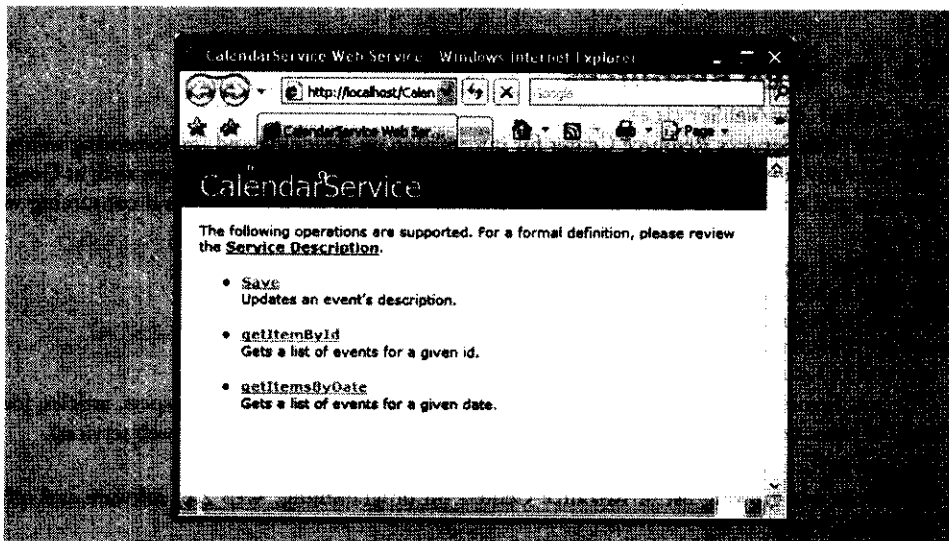




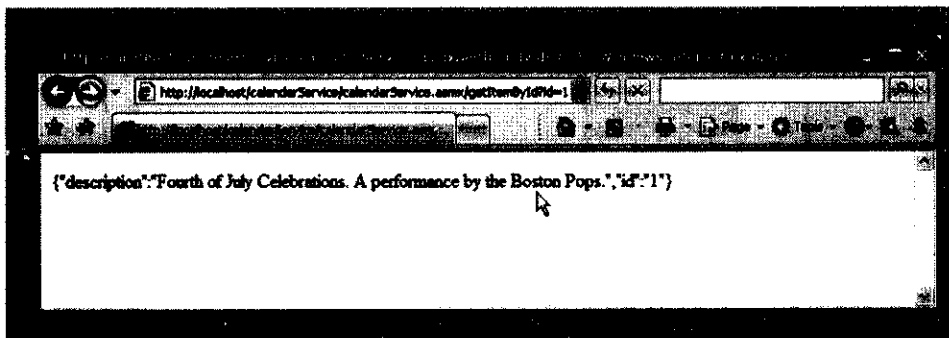
**Fig. 28.28** | QueryBuilder specifying an UPDATE statement used to modify a description.

#### Step 8: Testing the Web Service

At this point, you can use the `CalendarService.asmx` page to test the web service's methods. To do so, select **Start Without Debugging** from the **Debug** menu. Figure 28.29 shows the test page that is displayed for this web service when you run the web service application.



**Fig. 28.29** | The test page for the CalendarService web service. (Part 1 of 2.)



**Fig. 28.29** | The test page for the CalendarService web service. (Part 2 of 2.)

Calling a REST-based web service is simpler than calling SOAP web services demonstrated earlier. Fig. 28.29 shows that you can invoke the web service directly from your browser. For example, if you type the URL `http://localhost/calendarService/calendarService.asmx/getItemById?id=1` the browser will invoke the method `getItemById` and retrieve the event with the `id` value 1.

The client side interface for this application is implemented in the Ajax chapter, in Fig 15.11. Because our Calendar application uses the Dojo Toolkit, you must have Dojo installed on your computer. Download Dojo 0.4.3 from [dojotoolkit.org/downloads](http://dojotoolkit.org/downloads), extract the Dojo directory and rename it to `dojo043`. Then place the `CalendarService` folder that contains your web service, the `dojo043` folder that contains the Dojo toolkit and the `Calendar.html` file in the same directory in the root directory of your web server.

Run the web service and direct your browser to the location of the `Calendar.html` file. We populated the database only with events for July 2007, so the calendar is coded to always display July 2007 when the application is loaded. To test whether the web service works click a few dates like the fourth of July, the sixth of July or the twentieth of July for which events exist in the Calendar database

## 28.10 Web Resources

[www.deitel.com/WebServices/](http://www.deitel.com/WebServices/)

Visit our Web Services Resource Center for information on designing and implementing web services in many languages, and information about web services offered by companies such as Google, Amazon and eBay. You'll also find many additional Java tools for publishing and consuming web services.

[www.deitel.com/java/](http://www.deitel.com/java/)

[www.deitel.com/JavaSE6Mustang/](http://www.deitel.com/JavaSE6Mustang/)

[www.deitel.com/JavaEE5/](http://www.deitel.com/JavaEE5/)

[www.deitel.com/JavaCertification/](http://www.deitel.com/JavaCertification/)

[www.deitel.com/JavaDesignPatterns/](http://www.deitel.com/JavaDesignPatterns/)

Our Java Resource Centers provide Java-specific information, such as books, papers, articles, journals, websites and blogs that cover a broad range of Java topics (including Java web services).

[www.deitel.com/ResourceCenters.html](http://www.deitel.com/ResourceCenters.html)

Check out our growing list of Resource Centers on programming, Web 2.0, software and other interesting topics.

[java.sun.com/webservices/jaxws/index.jsp](http://java.sun.com/webservices/jaxws/index.jsp)

The official site for the Sun Java API for XML Web Services (JAX-WS). Includes the API, documentation, tutorials and other useful links.

[www.webservices.org](http://www.webservices.org)

Provides industry-related news, articles and resources for web services.

[www-130.ibm.com/developerworks/webservices](http://www-130.ibm.com/developerworks/webservices)

IBM's site for service-oriented architecture (SOA) and web services includes articles, downloads, demos and discussion forums regarding web services technology.

[www.w3.org/TR/wsd1](http://www.w3.org/TR/wsd1)

Provides extensive documentation on WSDL, including a thorough discussion of web services and related technologies such as XML, SOAP, HTTP and MIME types in the context of WSDL.

[www.w3.org/TR/soap](http://www.w3.org/TR/soap)

Provides extensive documentation on SOAP messages, using SOAP with HTTP and SOAP security issues.

[www.ws-i.org](http://www.ws-i.org)

The Web Services Interoperability Organization's website provides detailed information regarding building web services based on standards that promote interoperability and true platform independence.

[webservices.xml.com/security](http://webservices.xml.com/security)

Articles about web services security and standard security protocols.

### *REST-Based Web Services*

[en.wikipedia.org/wiki/REST](http://en.wikipedia.org/wiki/REST)

Wikipedia resource explaining Representational State Transfer (REST).

[www.xfront.com/REST-Web-Services.html](http://www.xfront.com/REST-Web-Services.html)

Article entitled "Building Web Services the REST Way."

[www.ics.uci.edu/~fielding/pubs/dissertation/rest\\_arch\\_style.htm](http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm)

The dissertation that originally proposed the concept of REST-based services.

[rest.blueoxen.net/cgi-bin/wiki.pl?ShortSummaryOfRest](http://rest.blueoxen.net/cgi-bin/wiki.pl?ShortSummaryOfRest)

A short introduction to REST.

[www.prescod.net/rest](http://www.prescod.net/rest)

Links to many REST resources.

### *Summary*

#### *Building a Web Service*

When you create a web service, you are creating a set of endpoints that can be accessed by other applications. The endpoints are defined by the WSDL file, which is a XML document that describes the service's interface.

The WSDL file is used by the client applications to discover the service's endpoints and to generate the client-side proxy class.

The client-side proxy class is a Java class that implements the service's interface and provides a simple way for the client applications to access the service's endpoints.

The client-side proxy class is generated by the IDE, and it is compiled into a JAR file that can be used by the client applications.

The client-side proxy class is also used by the IDE to generate the client-side stubs, which are used to generate the client-side proxy class.

The client-side stubs are generated by the IDE, and they are used to generate the client-side proxy class.

The client-side stubs are also used by the IDE to generate the client-side proxy class.

The client-side stubs are used by the IDE to generate the client-side proxy class.

The client-side stubs are used by the IDE to generate the client-side proxy class.

The client-side stubs are used by the IDE to generate the client-side proxy class.

The client-side stubs are used by the IDE to generate the client-side proxy class.

The client-side stubs are used by the IDE to generate the client-side proxy class.

The client-side stubs are used by the IDE to generate the client-side proxy class.

The client-side stubs are used by the IDE to generate the client-side proxy class.

The client-side stubs are used by the IDE to generate the client-side proxy class.

The client-side stubs are used by the IDE to generate the client-side proxy class.

The client-side stubs are used by the IDE to generate the client-side proxy class.

The client-side stubs are used by the IDE to generate the client-side proxy class.

When you add a web service reference, the IDE creates and compiles the client-side stubs—the framework of Java code that supports the client-side proxy class.

## Figure 28.6 Internet & World Wide Web How to Program

- Therefore, calls methods on a proxy object, which uses the client-side artifacts to interact with the web service.
- To add a web service reference, right click the client project name in the NetBeans Projects tab, then select **New > Web Services Client...** In the dialog's WSDL URL field, specify the URL of the web service's WSDL.
- NetBeans uses the WSDL description to generate the client-side proxy class and artifacts.
- When you specify the web service you want to consume, NetBeans copies the web service's WSDL into a file in your project. You can view this file from the NetBeans Files tab by expanding the folder in the project's `src1-resources` folder.
- The client-side artifacts and the client's copy of the WSDL file can be regenerated by right clicking the web service's node in the NetBeans Projects tab and selecting **Refresh Client**.
- You can view the IDE-generated client-side artifacts by selecting the NetBeans Files tab and expanding the project's `src1` folder.

### Section 28.5 SOAP

- SOAP is a commonly used, platform-independent, XML-based protocol that facilitates remote procedure calls, typically over HTTP.
- The protocol that transmits request-and-response messages is also known as the web service's wire format or wire protocol, because it defines how information is sent "along the wire."
- Each request and response is packaged in a SOAP message (also known as a SOAP envelope) containing the information that a web service requires to process the message.
- The wire format used to transmit requests and responses must support all types passed between the applications. SOAP supports primitive types and their wrapper types, as well as Date, Time and others. SOAP can also transmit arrays and objects of user-defined types.
- When a program invokes a web method, the request and all relevant information are packaged in a SOAP message and sent to the server on which the web service resides. The web service processes the SOAP message's contents, which specify the method to invoke and its arguments. After the web service receives and parses a request, the proper method is called, and the response is sent back to the client in another SOAP message. The client-side proxy parses the response, which contains the result of the method call, and returns the result to the client application.
- The SOAP messages are generated for you automatically. So you don't need to understand the details of SOAP or XML to take advantage of it when publishing and consuming web services.

### Section 28.6 Session Tracking in Web Services

- Sending session information also enables a web service to distinguish between clients and eliminates the need to pass client information between the client and the web service multiple times.

#### Section 28.6.1 Creating a Blackjack Web Service

- To do session tracking in a web service, you must include code for the resources that maintain the session state information. In the past, you had to write the sometimes tedious code to create these resources. JAX-WS, however, handles this for you via the `@Resource` annotation. This annotation enables tools like NetBeans to "inject" complex support code into your class, thus allowing you to focus on your business logic rather than the support code.
- Using annotations to add code that supports your classes is known as dependency injection. Annotations like `@WebService`, `@WebMethod` and `@WebParam` also perform dependency injection.
- A `WebServiceContext` object enables a web service to access and maintain information for a specific request, such as session state. The code that creates a `WebServiceContext` object is injected into the class by an `@Resource` annotation.



- The `WebServiceContext` object is used to obtain a `MessageContext` object. A web service uses a `MessageContext` to obtain an `HttpSession` object for the current client.
- The `MessageContext` object's `get` method is used to obtain the `HttpSession` object for the current client. Method `get` receives a constant indicating what to get from the `MessageContext`. The constant `MessageContext.SERVLET_REQUEST` indicates that we'd like to get the `HttpServletRequest` object for the current client. We then call method `getSession` to get the `HttpSession` object from the `HttpServletRequest` object.
- `HttpSession` method `getAttribute` receives a `String` that identifies the object to obtain from the session state.

### Section 28.6.2 Consuming the Blackjack Web Service

- In the JAX-WS 2.0 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 28.8 Passing an Object of a User-Defined Type to a Web Service

- Web services can receive and return objects of user-defined types—known as custom types.
- Custom types that are sent to or from a web service using SOAP are serialized into XML format. This process is referred to as XML serialization and is handled for you automatically.
- A class that is used to specify parameter or return types in web methods must provide a public default or no-argument constructor. Also, any instance variables that should be serialized must have public `set` and `get` methods or the instance variables must be declared public.
- Any instance variable that is not serialized simply receives its default value (or the value provided by the no-argument constructor) when an object of the class is deserialized.

## Terminology

AbstractPageBean class	dependency injection
adding a web service reference to a project in Netbeans	Deploy Project option in Netbeans
Apache Tomcat server	deploy a web service
application server	get method of interface <code>MessageContext</code>
B2B (business-to-business) transactions	getRequestContext method of interface <code>BindingProvider</code>
BEA WebLogic Server	GlassFish server
<code>BindingProvider</code> interface	JAX-WS 2.0
Built Project option in Netbeans	JBoss Application Server
business-to-business (B2B) transactions	<code>MessageContext</code> interface
Clean and Built Project option in Netbeans	Netbeans 5.5 IDE
Clean Project option in Netbeans	New Project dialog in Netbeans
client-side session	New Web Service Client dialog in Netbeans
creating a web service	New Web Service dialog in Netbeans
custom type	parse a SOAP message

POJO (plain old Java object)	Sun Java System Application Server Tester web page
Project Properties dialog in Netbeans	test a web service
proxy class for a web service	Web Application project in Netbeans
proxy object handles the details of communicating with the web service	Web Service Description Language (WSDL)
publish a web service	web service reference
put method of interface RequestContext	Web Services Interoperability Organization (WS-I)
remote machine	webMethod annotation
Representational State Transfer (REST)	webMethod annotation operationName element
RequestContext interface	WebParam annotation
Resource annotation	WebParam annotation name element
REST (Representational State Transfer)	WebService annotation
Run Project option in Netbeans	WebService annotation name element
server-side artifacts	WebService annotation serviceName element
session tracking in web services	WebServiceContext interface
SOAP envelope	wire format
SOAP message	wire protocol
SOAP (Simple Object Access Protocol)	WS-I Basic Profile 1.1 (BP 1.1)
Sun Java System Application Server	XML serialization

### Self-Review Exercises

- 28.1** State whether each of the following is *true* or *false*. If *false*, explain why.
- All methods of a web service class can be invoked by clients of that web service.
  - 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.
  - A proxy class communicating with a web service normally uses SOAP to send and receive messages.
  - Session tracking is automatically enabled in a client of a web service.
  - Web methods cannot be declared static.
  - A user-defined type used in a web service must define both *get* and *set* methods for any property that will be serialized.
  - RESTful web services are implemented using web standards, such as HTTP, XML, and JSON.
- 28.2** Fill in the blanks for each of the following statements:
- When messages are sent between an application and a web service using SOAP, each message is placed in a(n) \_\_\_\_\_.
  - \_\_\_\_\_ refers to an architectural style for implementing web services.
  - A web service in Java is a(n) \_\_\_\_\_—it does not need to implement any interfaces or extend any classes.
  - Web service requests are typically transported over the Internet via the \_\_\_\_\_ protocol.
  - To set the exposed name of a web method, use the \_\_\_\_\_ element of the *webMethod* annotation.
  - \_\_\_\_\_ transforms an object into a format that can be sent between a web service and a client.

### Exercises

- 28.3 (Phone Book Web Service)** Create a web service that stores phone book entries in the database PhonebookDB and a web client application that consumes this service. Use the steps in Section 28.7.1.



to create the PhoneBook database. The database should contain one table—PhoneBook—with three columns—LastName, FirstName and PhoneNumber—each of type VARCHAR. The LastName and FirstName columns should store up to 30 characters. The PhoneNumber column should support phone numbers of the form (800) 555-1212 that contain 14 characters.

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 Strings as arguments to the web service. The getEntries web method should return an array of Strings 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)
```

**28.4 (Blackjack Web Service Modification)** Modify the blackjack web service example in Section 28.6 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 ArrayLists 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 ArrayLists 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.

.....