15a

SOAP Web Services

Objectives

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

15.1 Introduction

- A web service is a software component stored on one computer that can be accessed by an application (or other software component) on another computer over a network.
- Web services communicate using such technologies as XML, JSON and HTTP.
- In this lecture, we use two Java APIs that facilitate web services.
  - JAX-WS is based on the Simple Object Access Protocol (SOAP)—an **XML-based protocol** that allows web services and clients to communicate, even if the client and the web service are written in different languages.
  - JAX-RS uses Representational State Transfer (REST)—a network architecture that uses the **web's traditional request/response mechanisms** such as `GET` and `POST` requests.

15.1 Introduction

Web service is a way of communication that allows interoperability between different applications on different platforms, for example, a java based application on Windows can communicate with a .Net based one on Linux. The communication can be done through a set of XML messages over HTTP protocol.

Web services are browsers and operating system independent service, which means it can run on any browser without the need of making any changes.

Web Services take Web-applications to the Next Level.

15.1 Introduction

Reuse already developed(old) **functionality** into new software:

- Lets understand with very simple example. Lets say you are developing a finance software for a company on java and you have old .net software which manages salary of employees. So rather then developing new software for employee part, you can use old software and for other parts like infrastructure you can develop your own functionalities.

15.1 Introduction

Usability :

- Web Services allow the business logic of many different systems to be **exposed over the Web**. This gives your applications the freedom to chose the Web Services that they need. **Instead of re-inventing** the wheel for each client, you need only include additional application-specific business logic on the client-side. This allows you to develop services and/or client-side code using the languages and tools that you want.

15.1 Introduction

Interoperability :

- This is the **most important benefit** of Web Services. Web Services typically work outside of private networks, offering developers a non-proprietary route to their solutions. Web Services also let **developers use their preferred programming languages**. In addition, thanks to the use of standards-based communications methods, Web Services are virtually platform-independent.

15.1 Introduction

Loosely Coupled:

- Each **service exists independently** of the other services that make up the application. Individual pieces of the application to be modified without impacting unrelated areas.

Ease of Integration:

- Data is isolated between applications creating 'silos'. **Web Services act as glue** between these and enable easier communications within and across organisations.

15.1 Introduction

Deployability :

- Web Services are **deployed over standard Internet technologies**. This makes it possible to deploy Web Services even over the firewall to servers running on the Internet on the other side of the globe. Also thanks to the use of proven community standards, underlying security (such as SSL) is already built-in.

15.1 Introduction (cont.)

- Business-to-Business Transactions
  - Rather than relying on proprietary applications, businesses can conduct transactions via standardized, widely available web services.
  - This has important implications for business-to-business (B2B) transactions.
  - Web services are **platform and language independent**, enabling companies to collaborate without worrying about the compatibility of their hardware, software and communications technologies.

15.1 Introduction (cont.)

By purchasing some web services and using other free ones that are relevant to their businesses, companies can spend less time developing applications and can create new ones that are more innovative.

E-businesses for example, can provide their customers with enhanced shopping experiences.

Consider an online music store:

- website links to information about various CDs
- link to the site of a company that sells concert tickets

Additional service to its customers, increase its site traffic and perhaps earn a commission on concert-ticket sales

15.2 Web Service Basics

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

15.3 Simple Object Access Protocol (SOAP)

- The **Simple Object Access Protocol** (SOAP) is a platform-independent protocol that uses XML to interact with web services, **typically over HTTP**.
- Each **request** and **response** is packaged in a **SOAP message**.
  - **Written in XML** so that they are computer readable, human readable and platform independent.
- Most **firewalls** **allow HTTP traffic** to pass through, so that clients can browse the web by sending requests to and receiving responses from web servers.
  - Thus, **SOAP-based services** can **send** and **receive** SOAP messages **over HTTP connections** with few limitations.

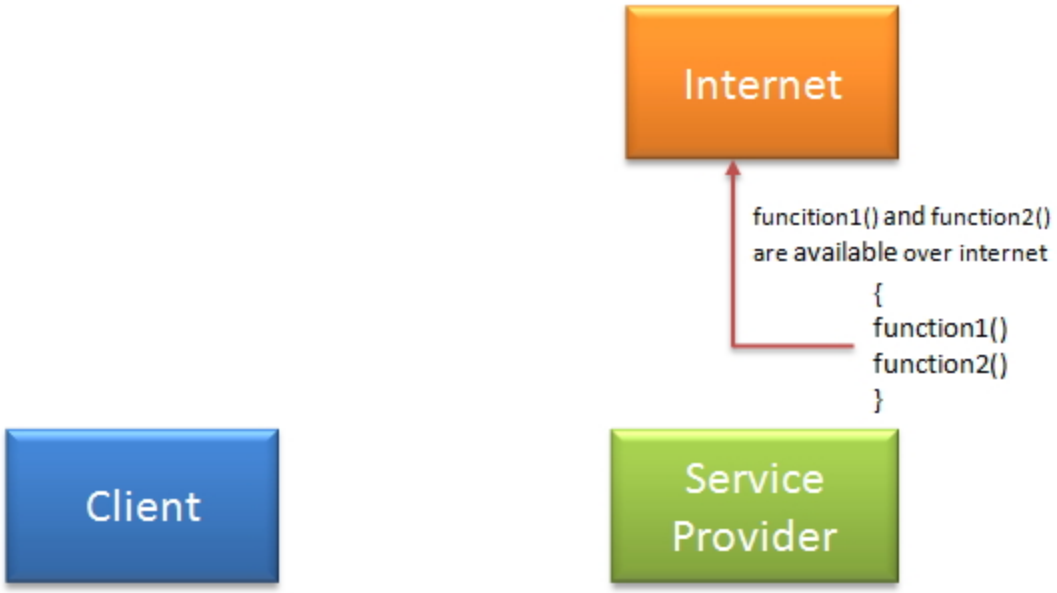
15.3 Simple Object Access Protocol (SOAP) (cont.)

- The **wire format** used to transmit requests and responses **must support all types** passed between the applications.
- When a program invokes a method of a SOAP web service, **the request and all relevant information are packaged in a SOAP message** enclosed in a SOAP **envelope** and sent to the server on which the web service resides.
- When the web service **receives this SOAP message**, it **parses** the **XML representing the message**, then processes the message's contents.
- The **message specifies the method** that the client **wishes to execute** and **the arguments the client passed** to that method.
- The **web service calls the method** with the specified arguments (if any) and sends the response back to the client in another SOAP message.
- The **client parses** the **response** to retrieve the method's result.

15.3 Simple Object Access Protocol (SOAP) (cont.)

A simple web service architecture have two components

- Client
- Service provider.



15.3 Simple Object Access Protocol (SOAP) (cont.)

In order to communicate with the Service provider, the client must know some information for the following:

- Location of webservices server
- Functions available, signature and return types of function.
- Communication protocol
- Input output formats

The **Service provider** will **create** a standard XML file which will have all above information. Provided the client has access to this file, then the client will be able to access web service. This **XML file** is called **WSDL**

15.4 JavaScript Object Notation (JSON)

- JavaScript Object Notation (**JSON**) is **an alternative** to XML for representing data.
  - **Text-based data-interchange format** used to represent objects in JavaScript as collections of name/value pairs represented as `Strings`.
  - Commonly used in Ajax applications.
  - Makes **objects easy to read**, create and parse
  - Much **less verbose than XML**, so it allows programs to transmit data efficiently across the Internet

15.4 JavaScript Object Notation (JSON) (cont.)

- Each JSON object is represented as a list of property names and values contained in curly braces:
  - `{ propertyName1 : value1, propertyName2 : value2 }`
- Arrays are represented with square brackets:
  - `[ value1, value2, value3 ]`
  - Each value in an array can be a string, a number, a JSON object, `true`, `false` or `null`.
- Representation of an array of address-book entries:
  - [{ first: 'Cheryl', last: 'Black' },
    { first: 'James', last: 'Blue' },
    { first: 'Mike', last: 'Brown' },
    { first: 'Meg', last: 'Gold' }]

15.5 Web Service Description Language

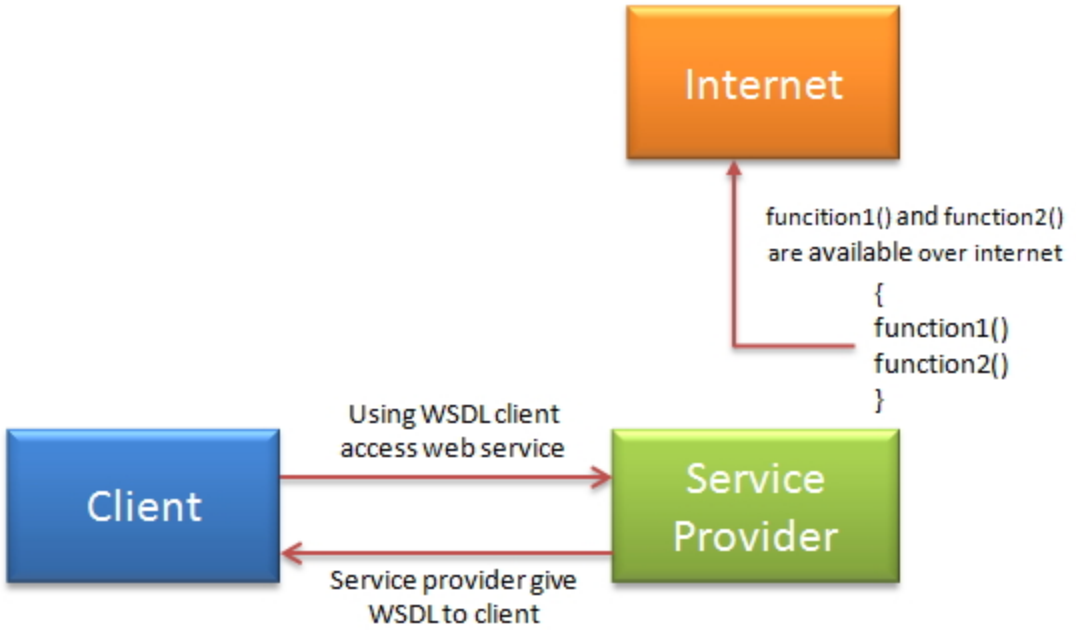
WSDL stands for **Web Service Description Language**. It is an XML file that describes the **technical details** of **how to implement a web service**, more specifically the **URI**, port, **method names**, **arguments**, and **data types**. Since WSDL is XML, it is both human-readable and machine-consumable, which aids in the ability to call and bind to services dynamically. Using this **WSDL** file we can understand things like,

- Port / **Endpoint** – URL of the web service
- **Input message format**
- **Output message format**
- **Security protocol** that needs to be followed
- Which **protocol the web service uses**

15.5 Web Service Description Language

There are two ways to access web service.

- If **Service provider knows client**. If the Service provider knows its client then it will provide its WSDL to client and client will be able to access web service.

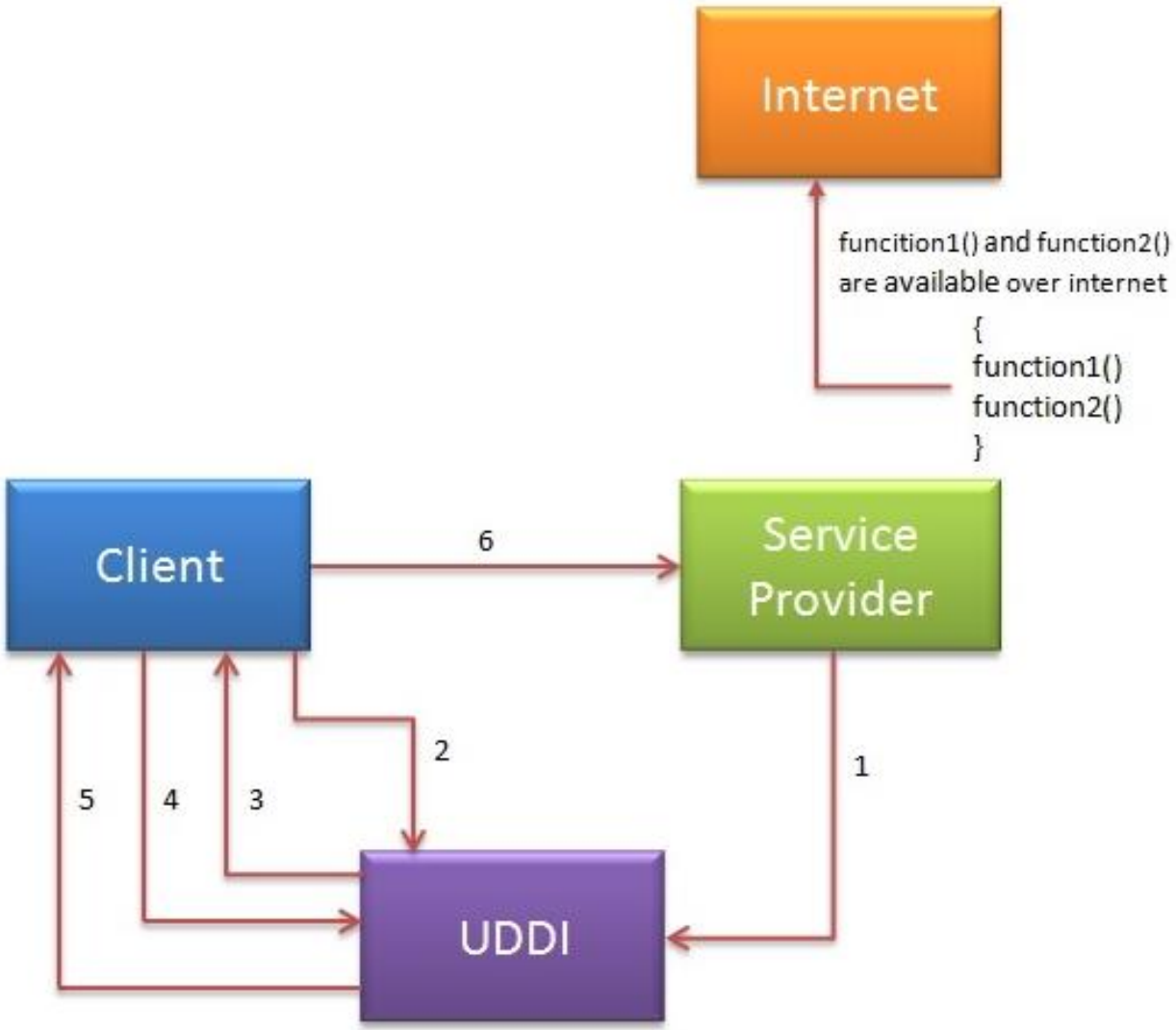


15.5 Web Service Description Language

The **Service provider** registers its **WSDL** to **UDDI** and client can access it from **UDDI**. **UDDI** stands for **Universal Description, Discovery and Integration**. It is **a directory service**. Web services can register with a UDDI and make themselves available **through it for discovery**. So **following steps** are involved.

- The Service provider registers with UDDI.
- The Client searches for service in UDDI.
- The UDDI returns all service providers offering that service.
- The Client chooses service provider
- The UDDI returns WSDL of chosen service provider.
- Using WSDL of the Service provider, a client accesses web service

15.5 Web Service Description Language



15.6 Publishing and Consuming SOAP-Based Web Services

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

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

- To create a web service in NetBeans, you first create a Web Application project.
- To create a web application, perform the following steps:
  - 1. Select File > New Project... to open the New Project dialog.
  - 2. Select **Java Web** from the dialog's Categories list, then select **Web Application** from the Projects list. Click Next >.
  - 3. Specify the name of your project in the Project Name field and specify where you'd like to store the project in the Project Location field. You can click the Browse button to select the location. Click Next >.
  - 4. Select **GlassFish Server 4.x** from the Server drop-down list and **Java EE 7** from the Java EE Version drop-down list.
  - 5. Click Finish to create the project.

15.15.1 Creating a Web Application Project and Adding a Web Service Class in NetBeans (cont.)

- Perform the following steps to **add a web service class** to the project:
  - 1. In the Projects tab in NetBeans, right click the project's node and select New > **Web Service**... to open the **New Web Service** dialog.
  - 2. Specify the **name of the service** in the Web Service Name field.
  - 3. Specify the **package name** in the Package field.
  - 4. Click **Finish** to create the web service class.
- The IDE **generates a sample web service class** with the name from *Step 2* in the package from *Step 3*.
- You can find this class in your project's Web Services node.
- In this class, you'll **define the methods** that your web service makes available to client applications.

15.15.2 Defining the `WelcomeSOAP` Web Service in NetBeans

- Figure 15.1 contains the `WelcomeSOAPService` code.
- By default, each new web service class created with the JAX-WS APIs is a POJO (plain old Java object)
  - You do *not* need to extend a class or implement an interface to create a web service.

Figure 15.1 contains the `WelcomeSOAPService` code

```
1  // Fig. 31.1: WelcomeSOAP.java
2  // Web service that returns a welcome message via SOAP.
3  package com.deitel.welcomesoap;
4
5  import javax.jws.WebService; // program uses the annotation @WebService
6  import javax.jws.WebMethod; // program uses the annotation @WebMethod
7  import javax.jws.WebParam; // program uses the annotation @WebParam
8
9  @WebService() // annotates the class as a web service
10 public class WelcomeSOAP
11 {
12    // WebMethod that returns welcome message
13    @WebMethod( operationName = "welcome" )
14    public String welcome( @WebParam( name = "name" ) String name )
15    {
16       return "Welcome to JAX-WS web services with SOAP, " + name + "!";
17    } // end method welcome
18 } // end class WelcomeSOAP
```

15.15.2 Defining the `welcomeSOAP` Web Service in NetBeans (cont.)

- When you deploy a web application containing a class that uses the `@WebService` annotation, the server recognizes that the class implements a web service and creates all the **server-side artifacts** that support the web service
  - **Framework** that allows the web service to **wait for client requests** and **respond to those requests** once it is deployed on an application server.

15.15.2 Defining the `welcomeSOAP` Web Service in NetBeans (cont.)

- The **@WebService** annotation indicates that a class implements a web service.The annotation is followed by parentheses containing optional annotation attributes.
- The **name** attribute s**pecifies the name of the service endpoint interface class** that will be generated for the client.
- A **service endpoint interface** (SEI) class (sometimes called a proxy class) is used to interact with the web service
  - a client application consumes the web service by invoking methods on the service endpoint interface object.
- The **serviceName** attribute specifies the service name, which is also **the name of the class** that the client uses to obtain a service endpoint interface object.
  - If not specified, the **web service's name** is **assumed** to be the java class name followed by the word **`Service`.**

15.15.2 Defining the `welcomeSOAP` Web Service in NetBeans (cont.)

- A **method** is tagged with the **`@WebMethod`** annotation to indicate that it can be called remotely.
  - Any methods that are not tagged with **`@webMethod`** are **not accessible to clients** that consume the web service.
- An **`@webMethod`** annotation's **`operationName`** attribute to specifies the method name that is exposed to the web service's client.
  - If the **`operationName`** is not specified, it is set to the actual Java method's name.
- A **parameter** is annotated with the **`@WebParam`** annotation.
- The optional **`@webParam`** attribute **`name`** indicates the parameter name that is exposed to the web service's clients.
  - If you don't specify the name, the actual parameter name is used.

Common Programming Error 31.1

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

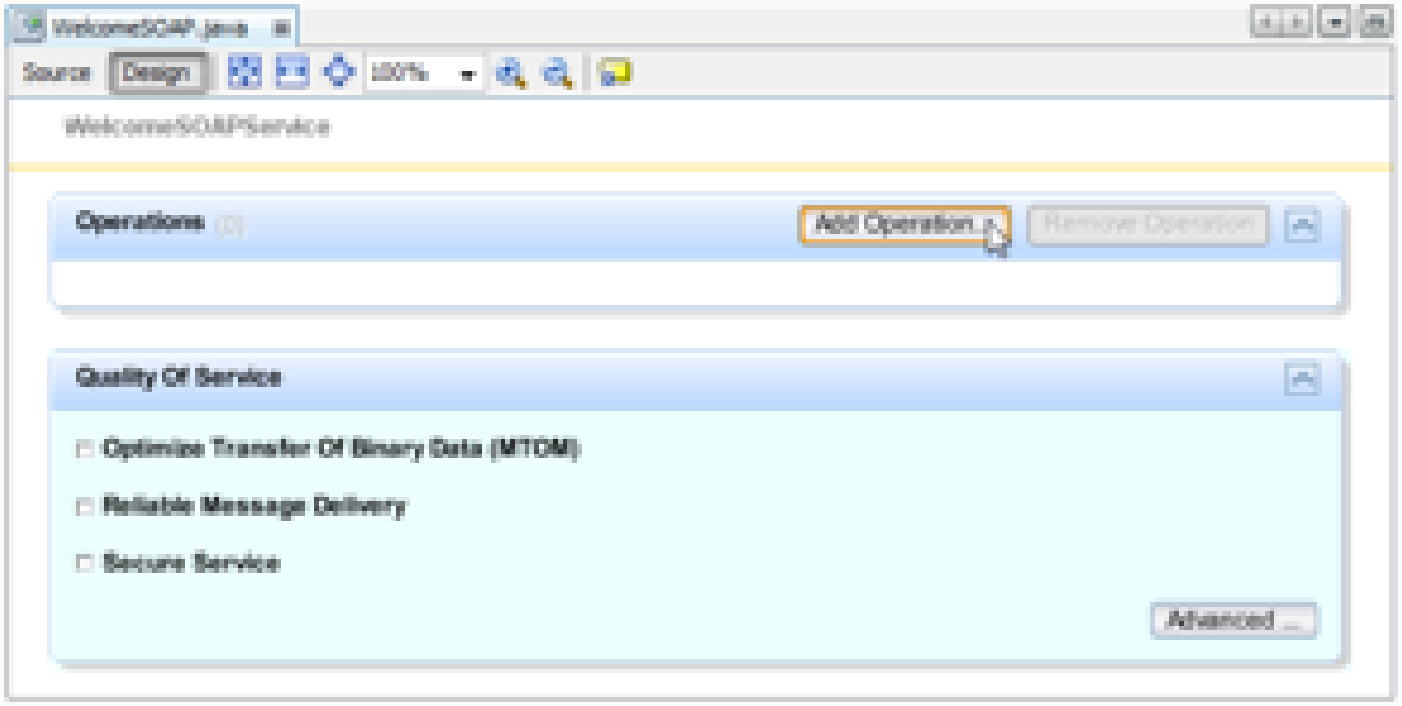
Common Programming Error 31.2

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.*

15.15.2 Defining the `WelcomeSOAP` Web Service in NetBeans (cont.)

- To define the `WelcomeSOAP` class's `welcome` method, **perform the following steps:**
  - 1. In the project's Web Services node, double click `WelcomeSOAP` to open the file **`WelcomeSOAPService.java`** in the code editor.
  - 2. Click the Design button at the top of the code editor to show the web service design view (Fig. 15.2).

Fig. 15.2 web service design view



15.15.2 Defining the `welcomeSOAP` Web Service in NetBeans (cont.)

- 3. Click the **Add Operation**… button to display the Add Operation… dialog (Fig. 15.3).
- 4. Specify the **method name** `welcome` in the Name field. The default Return Type (`String`) is correct for this example.
- 5. Add the method's `name` parameter by clicking the Add button to the right of the Parameters tab then entering name in the Name field. The parameter's default Type (`String`) is correct for this example.
- 15. Click OK to create the `welcome` method. The design view should now appear as shown in Fig. 15.3.
- 7. At the top of the design view, click the **Source** button to display the class's source code and **add the code** line 18 of Fig. 15.1 to the body of method `welcome`.

Fig. 15.3 Add Operation…

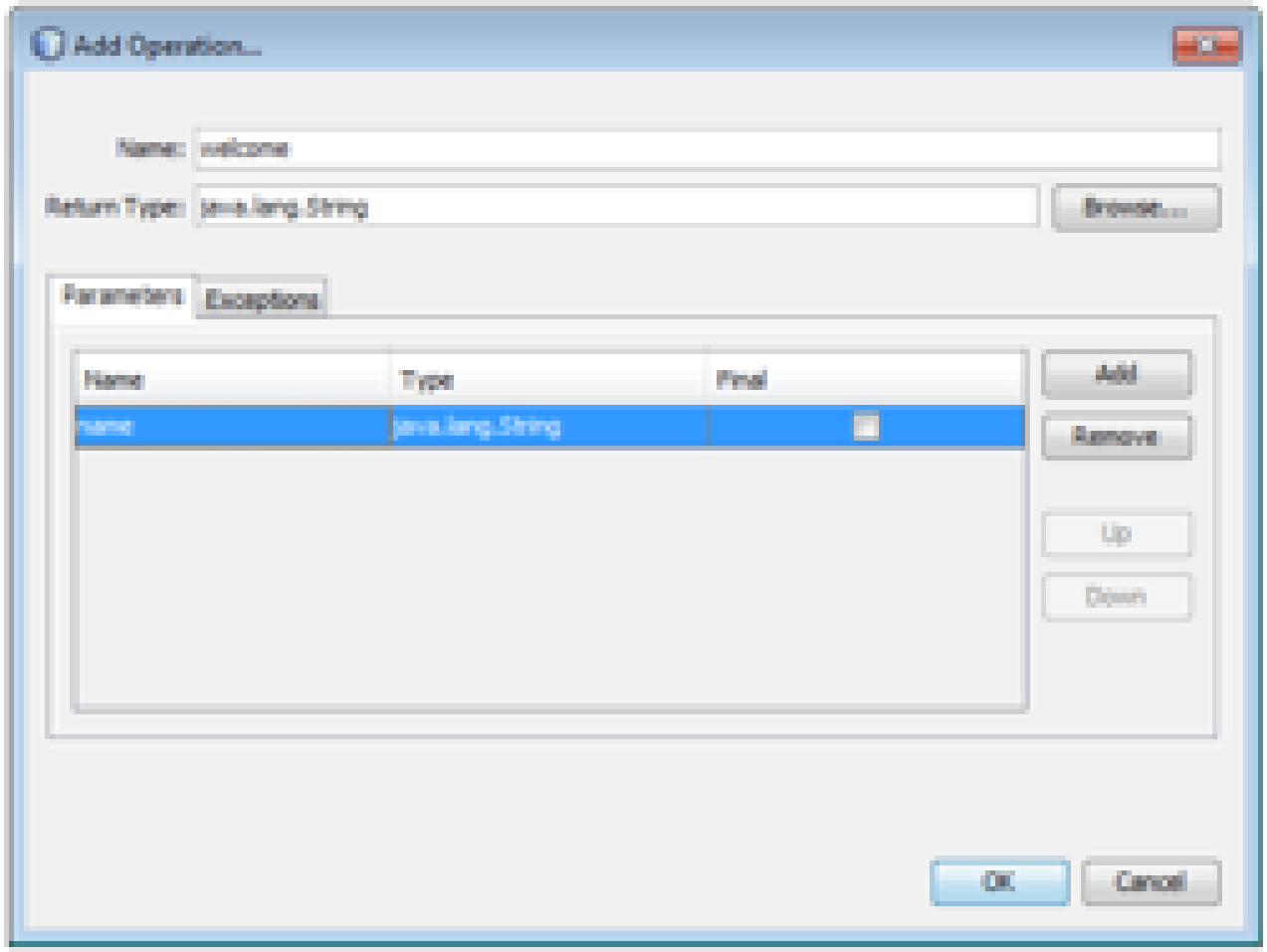
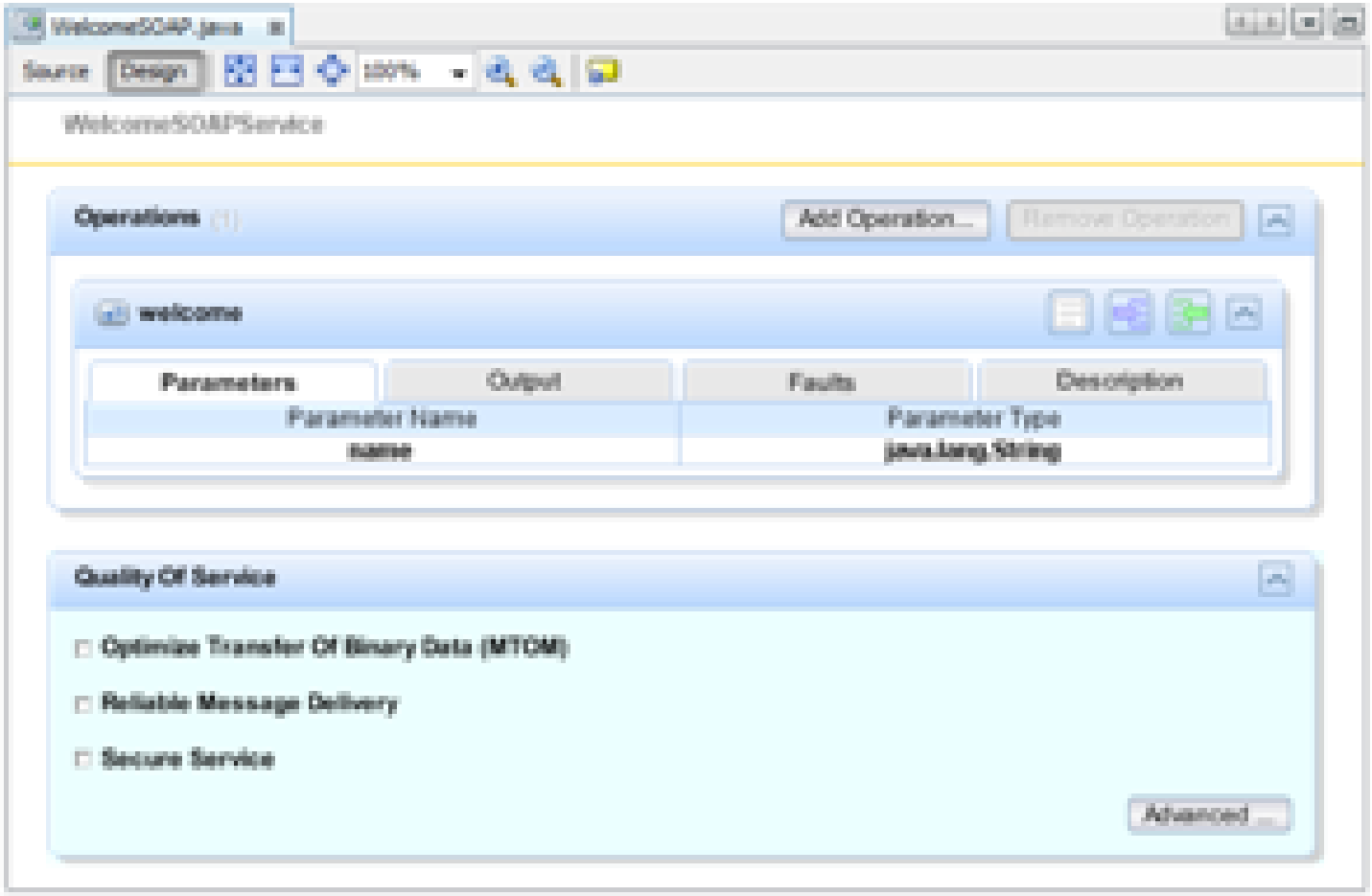


Fig. 15.4 Add Operation…parameters



15.15.3 Publishing the welcomeSOAP Web Service from NetBeans

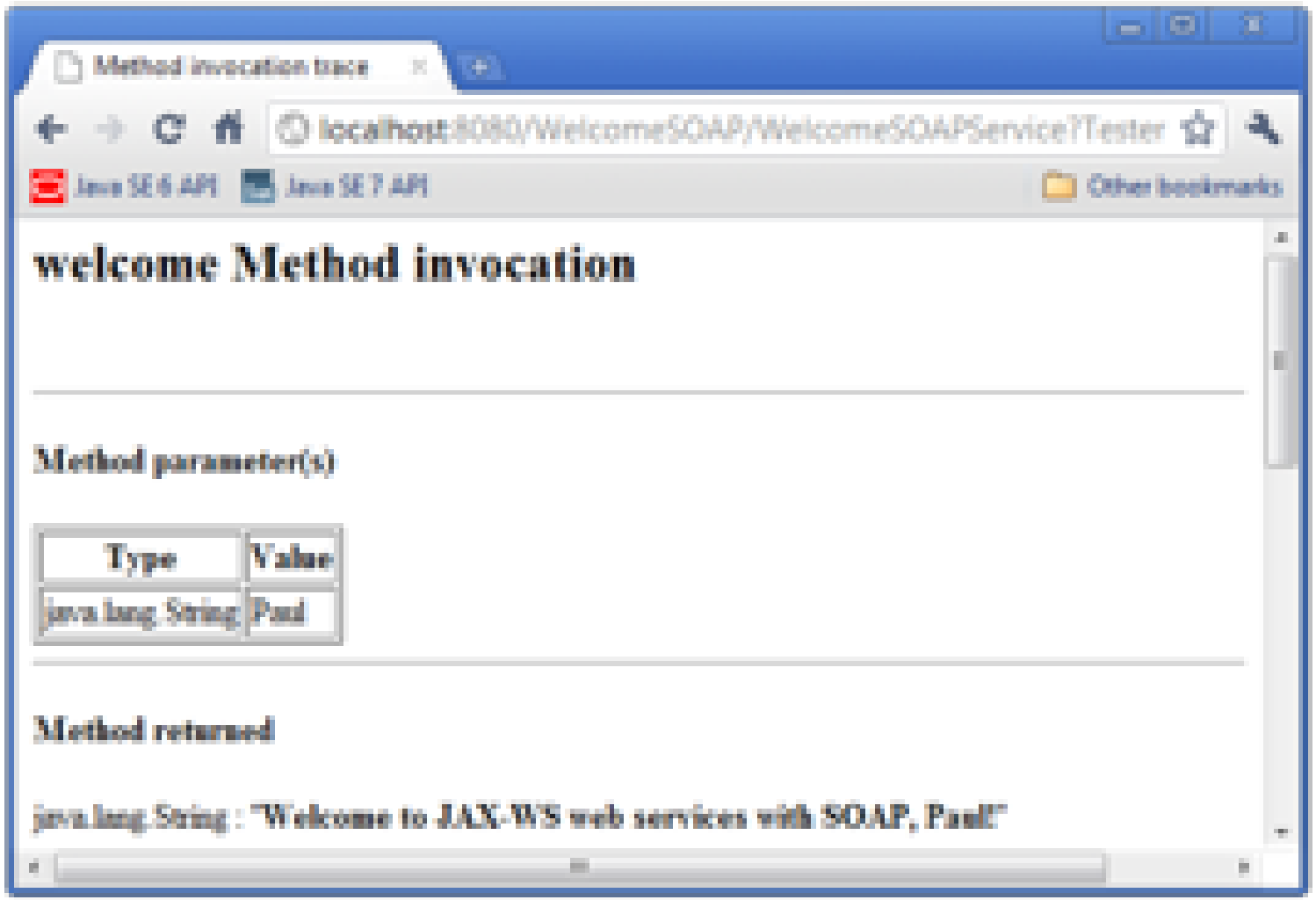
- NetBeans handles all the details of building and deploying a web service for you.
  - This includes creating the framework required to support the web service.
- Right click the project name welcomeSoap in the Projects tab and select Deploy to build and deploy the web application in the GlassFish server.

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

- The GlassFish application server can dynamically create a web page for testing a web service's methods from a web browser.
- Expand the project's Web Services in the NetBeans Projects tab.
- Right click the web service class name and select Test Web Service.
- The GlassFish application server builds the `Tester` web page and loads it into your web browser.
- Figure 15.5 shows the `Tester` web page for the `WelcomeSOAP` web service.

Fig. 15.5 The `Tester` web page

Fig. 15.6 The `Tester` web page



15.15.5 Testing the WelcomeSOAP Web Service with GlassFish Application Server's Tester Web Page (cont.)

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

```
http://host:8080/WelcomeSoap/WelcomeSoapService?Tester
```

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

- To consume a web service, a client must determine its functionality and how to use it.
- Web services normally contain a service description.
  - Web Service Description Language (WSDL)—an XML vocabulary that defines the methods a web service makes available and how clients interact with them.
  - A WSDL document also specifies lower-level information that clients might need, such as the required formats for requests and responses.
- WSDL documents help applications determine how to interact with the web services described in the documents.
- GlassFish generates a web service's WSDL dynamically for you.

15.15.5 Describing a Web Service with the Web Service Description Language (WSDL) (cont.)

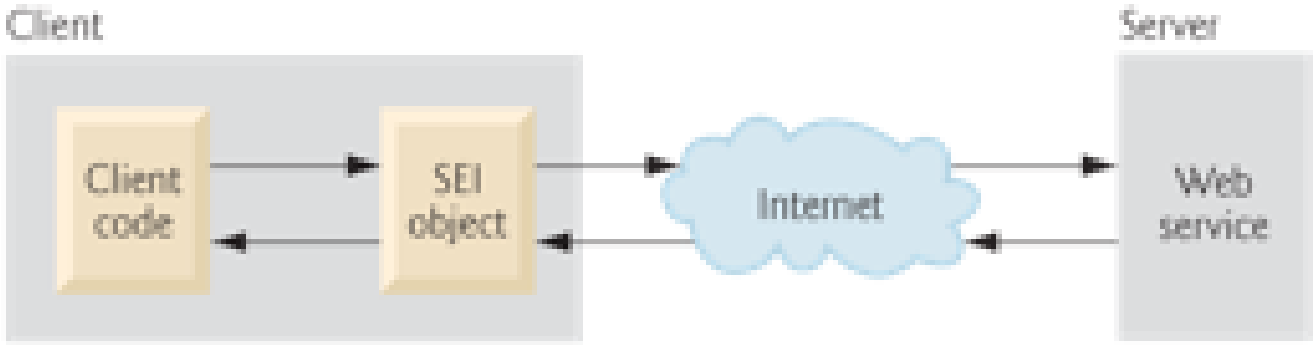
- To access the `WelcomeSOAP` web service, the client code will need the following WSDL URL:
  - `http://localhost:8080/WelcomeSoap/WelcomeSoapService?Tester`
- Eventually, you'll want clients on other computers to use your web service.
- Such clients need access to the web service's WSDL, which they would access with the following URL:
  - `http://`*`host`*`:8080/WelcomeSOAP/WelcomeSOAPService?WSDL`

  where *host* is the hostname or IP address of the server that hosts the web service.

15.15.6 Creating a Client to Consume the WelcomeSOAP Web Service

- You enable a Java-based client application to consume a web service by adding a web service reference to the client application.
  - Defines the service endpoint interface class that allows the client to access the web service.
- An application that consumes a web service consists of
  - an object of a service endpoint interface (SEI) class that's used to interact with the web service
  - a client application that consumes the web service by invoking methods on the service endpoint interface object
- The service endpoint interface object handles the details of passing method arguments to and receiving return values from the web service on the client's behalf.
- Figure 15.5 depicts the interactions among the client code, the SEI object and the web service.

Fig. 15.7 Interaction between a web service and a web service



15.15.6 Creating a Client to Consume the WelcomeSOAP Web Service (cont.)

- Requests to and responses from web services created with JAX-WS are typically transmitted via SOAP.
- Any client capable of generating and processing SOAP messages can interact with a web service, regardless of the language in which the web service is written.
- We now use NetBeans to create a client Java desktop GUI application.
- When you add a web service reference, the IDE creates and compiles the client-side artifacts
  - framework of Java code that supports the client-side service endpoint interface class.
- The client then calls methods on an object of the service endpoint interface class, which uses the rest of the artifacts to interact with the web service.

15.15.6 Creating a Client to Consume the WelcomeSOAP Web Service (cont.)

- Perform the following steps to create a client Java desktop application in NetBeans:
  - 1. Select File > New Project... to open the New Project dialog.
  - 2. Select Java from the Categories list and Java Application from the Projects list, then click Next >.
  - 3. Specify the name in the Project Name field and uncheck the Create Main Class checkbox if you intend to create a your own class that contains main.
  - 4. Click Finish to create the project.

15.15.6 Creating a Client to Consume the WelcomeSOAP Web Service (cont.)

- To add a web service reference, perform the following steps.
  - 1. Right click the project name (WelcomeSOAPClient) in the NetBeans Projects tab and select New > Web Service Client... from the pop-up menu to display the New Web Service Client dialog.
  - 2. In the WSDL URL field, specify the URL http://localhost:8080/WelcomeSoap/WelcomeSoapService?WSDL (Fig. 15.8). The IDE uses this WSDL to generate the client-side artifacts.
  - 3. For the other options, leave the default settings, then click Finish to create the web service reference and dismiss the New Web Service Client dialog.
- In the NetBeans Projects tab, the project now contains a Web Service References folder with the web service's service endpoint interface (Fig. 15.9).

Fig. 15.8 Create a web client

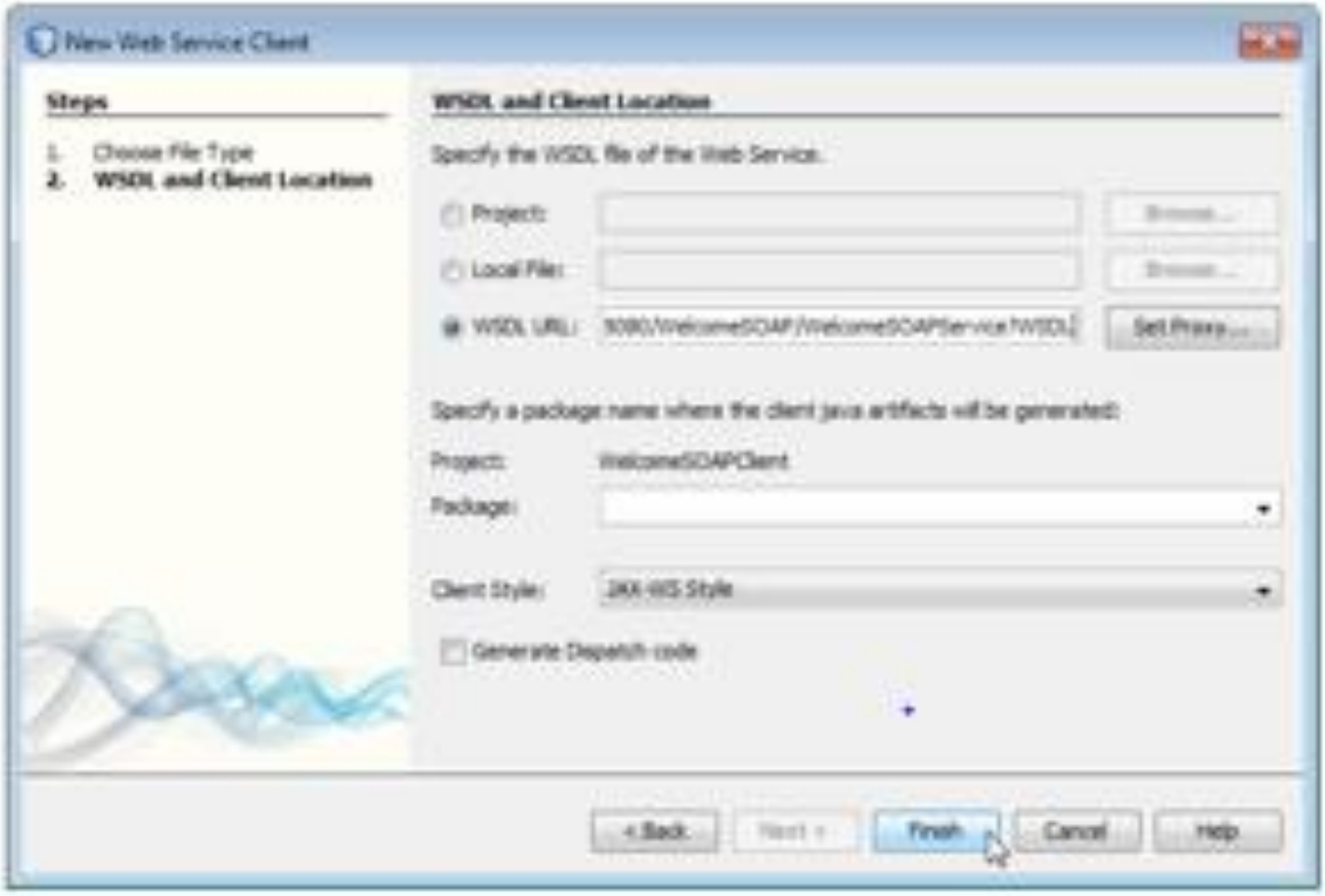
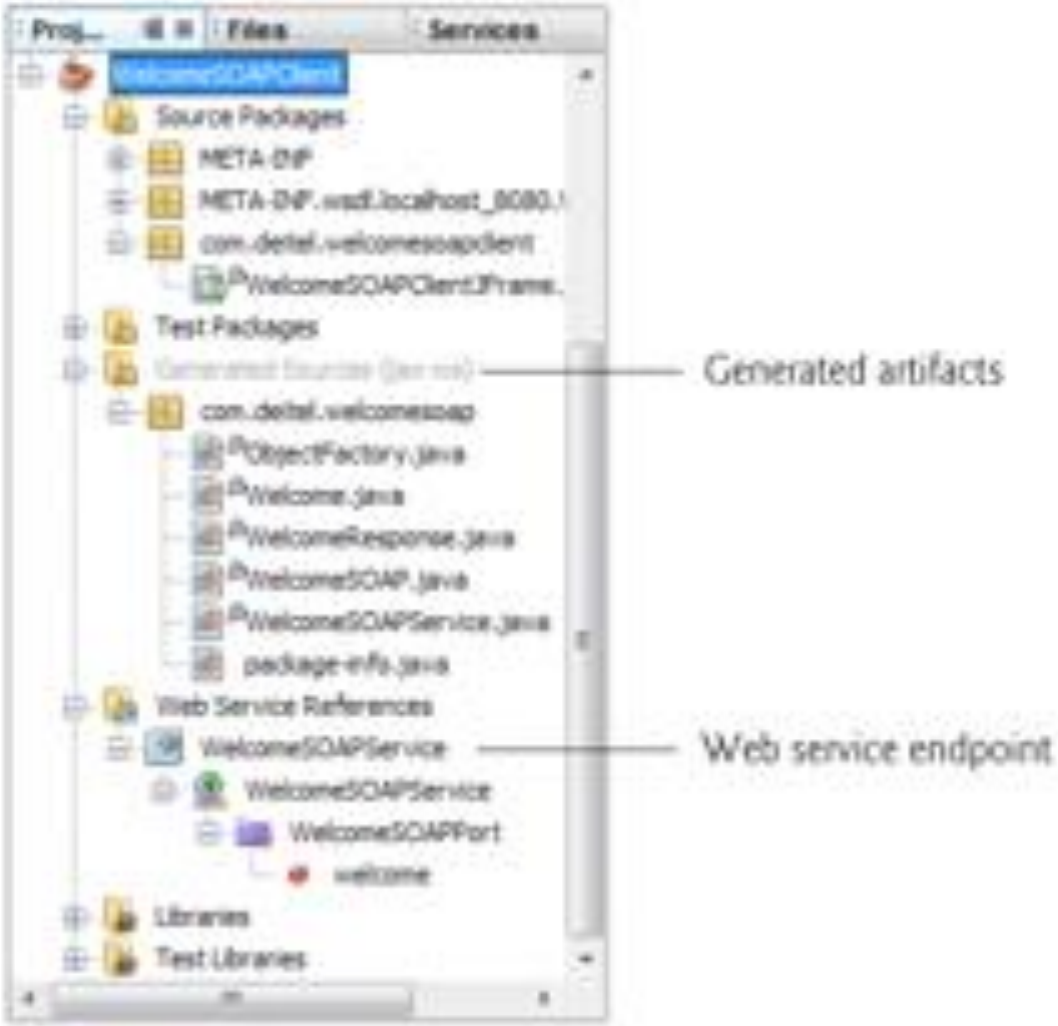


Fig. 15.9 NB project after adding a web service reference



15.15.8 Consuming the `welcomeSOAP` Web Service

- For this example, we use a GUI application to interact with the `welcomeSOAP` web service.

```
1  // Fig. 31.10: WelcomeSOAPClientJFrame.java
2  // Client desktop application for the WelcomeSOAP web service.
3  package com.deitel.welcomesoapclient;
4
5  import com.deitel.welcomesoap.WelcomeSOAP;
6  import com.deitel.welcomesoap.WelcomeSOAPService;
7  import javax.swing.JOptionPane;
8
```

Fig. 31.10 | Client desktop application for the `WelcomeSOAP` web service. (Part 1 of 5.)

```
 9  public class WelcomeSOAPClientJFrame extends javax.swing.JFrame
10  {
11     // references the service endpoint interface object (i.e., the proxy)
12     private WelcomeSOAP welcomeSOAPProxy;
13
14     // no-argument constructor
15     public WelcomeSOAPClientJFrame()
16     {
17        initComponents();
18
19        try
20        {
21           // create the objects for accessing the WelcomeSOAP web service
22           WelcomeSOAPService service = new WelcomeSOAPService();
23           welcomeSOAPProxy = service.getWelcomeSOAPPort();
24        } // end try
25        catch ( Exception exception )
26        {
27           exception.printStackTrace();
28           System.exit( 1 );
29        } // end catch
30     } // end WelcomeSOAPClientJFrame constructor
31
```

Fig. 31.10 | Client desktop application for the `WelcomeSOAP` web service. (Part 2 of 5.)

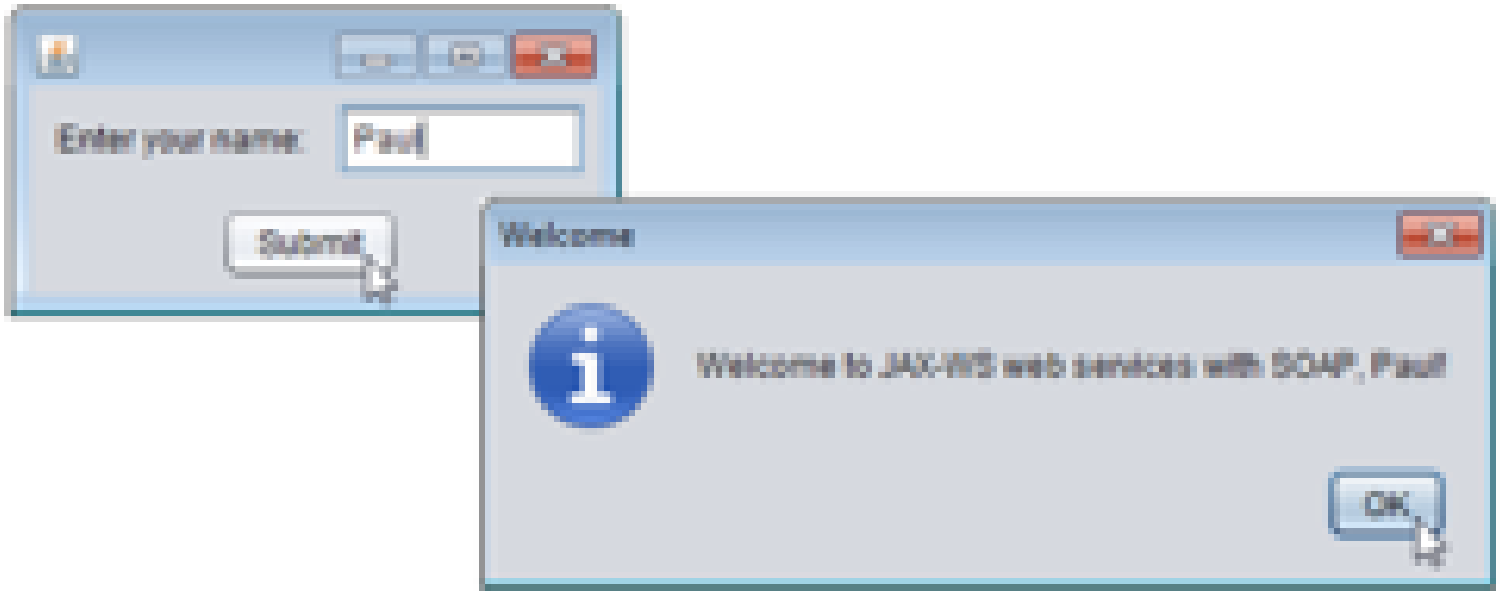
```
32      // The initComponents method is autogenerated by NetBeans and is called
33      // from the constructor to initialize the GUI. This method is not shown
34      // here to save space. Open WelcomeSOAPClientJFrame.java in this
35      // example's folder to view the complete generated code.
36
37      // call the web service with the supplied name and display the message
38      private void submitJButtonActionPerformed(
39         java.awt.event.ActionEvent evt )
40      {
41         String name = nameJTextField.getText(); // get name from JTextField
42
43         // retrieve the welcome string from the web service
44         String message = welcomeSOAPProxy.welcome( name );
45         JOptionPane.showMessageDialog( this, message,
46            "Welcome", JOptionPane.INFORMATION_MESSAGE );
47      } // end method submitJButtonActionPerformed
48
```

Fig. 31.10 | Client desktop application for the `WelcomeSOAP` web service. (Part 3 of 5.)

```
49      // main method begins execution
50      public static void main( String args[] )
51      {
52         java.awt.EventQueue.invokeLater(
53            new Runnable()
54            {
55               public void run()
56               {
57                  new WelcomeSOAPClientJFrame().setVisible( true );
58               } // end method run
59            } // end anonymous inner class
60         ); // end call to java.awt.EventQueue.invokeLater
61      } // end main
62
63      // Variables declaration - do not modify
64      private javax.swing.JLabel nameJLabel;
65      private javax.swing.JTextField nameJTextField;
66      private javax.swing.JButton submitJButton;
67      // End of variables declaration
68   } // end class WelcomeSOAPClientJFrame
```

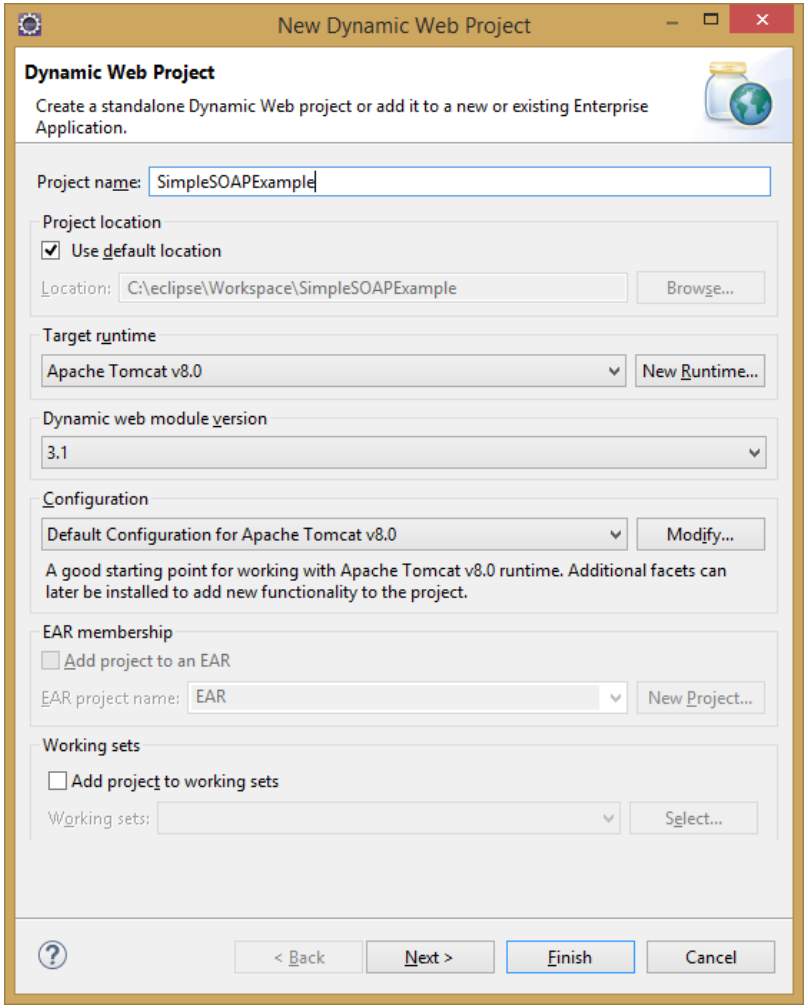
Fig. 31.10 | Client desktop application for the `WelcomeSOAP` web service. (Part 4 of 5.)

Fig. 15.10 Client desktop for the WelcomeSOAP web service



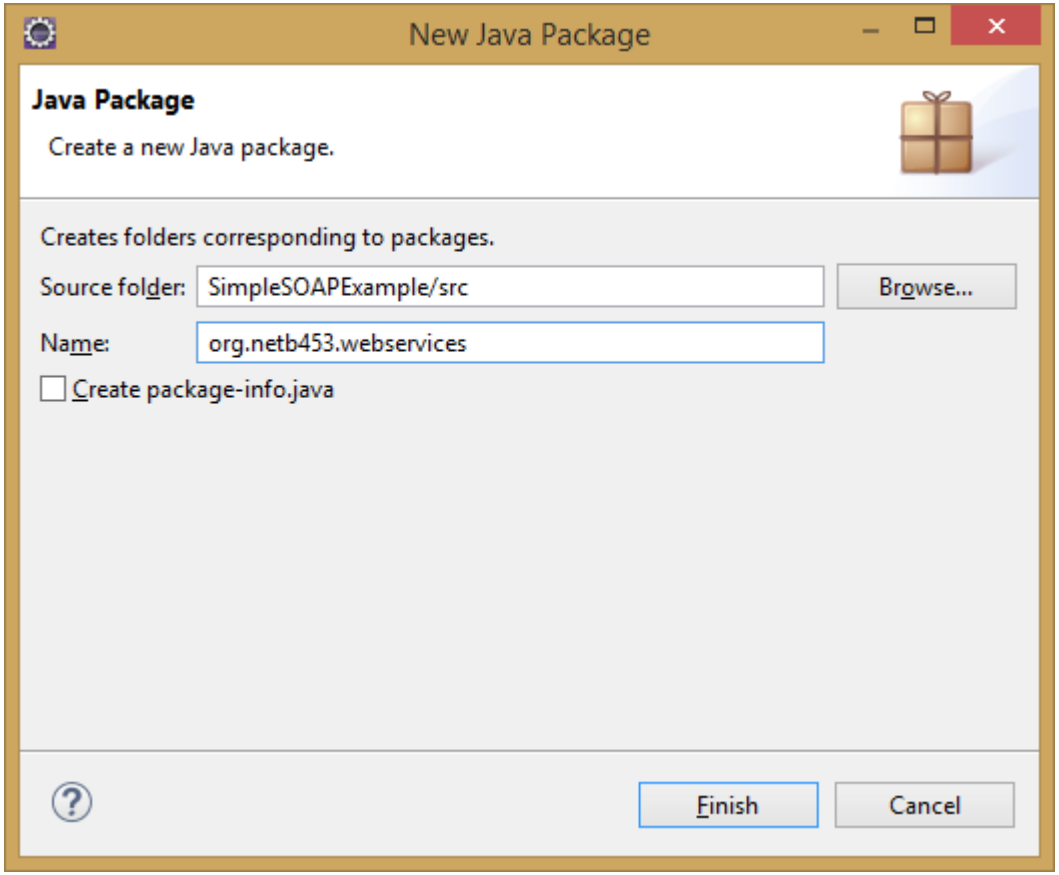
15.7 Creating web services in eclipse

1.Create new **Dynamic web project** and **name** it "SimpleSOAPExample".



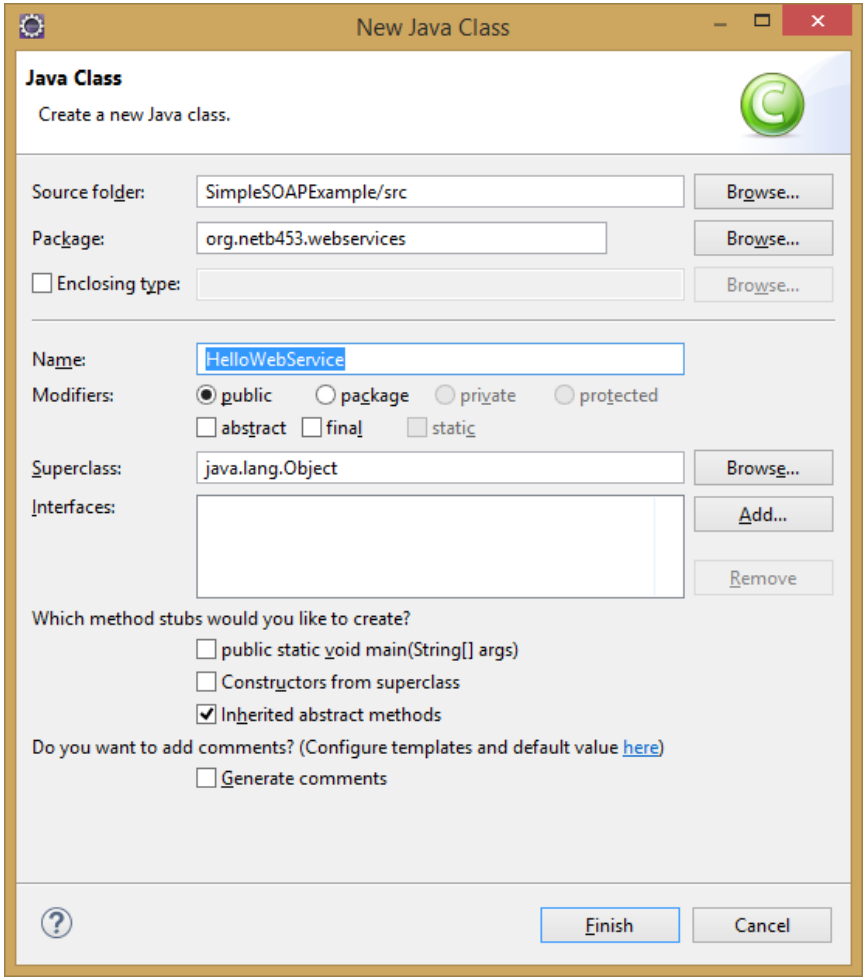
15.7 Creating web services in eclipse

2.Create new package named "org.netb453.webservices"



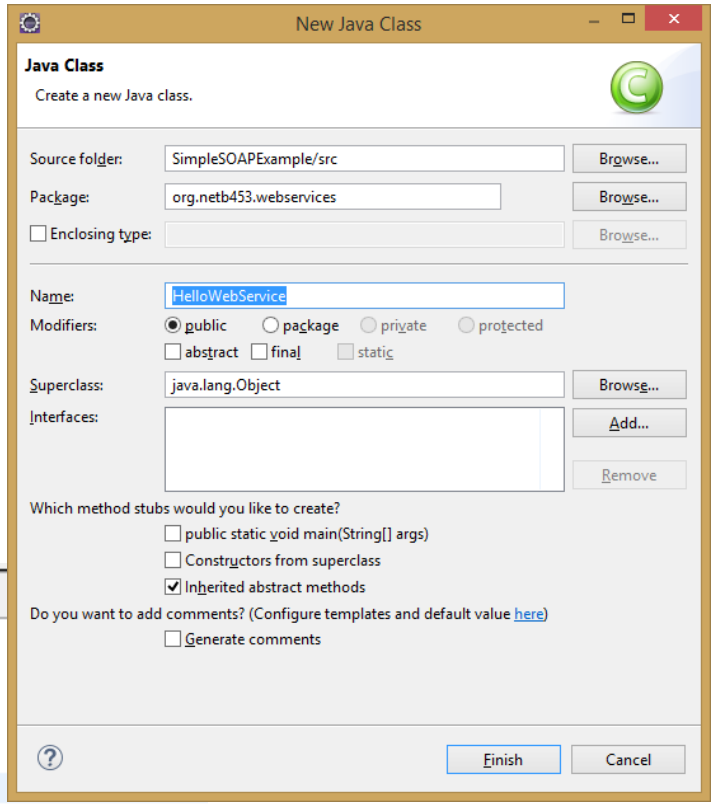
15.7 Creating web services in eclipse

3.Create a simple java class named “HelloWebService.java"



15.7 Creating web services in eclipse

3.Create a simple java class named “HelloWebService.java"

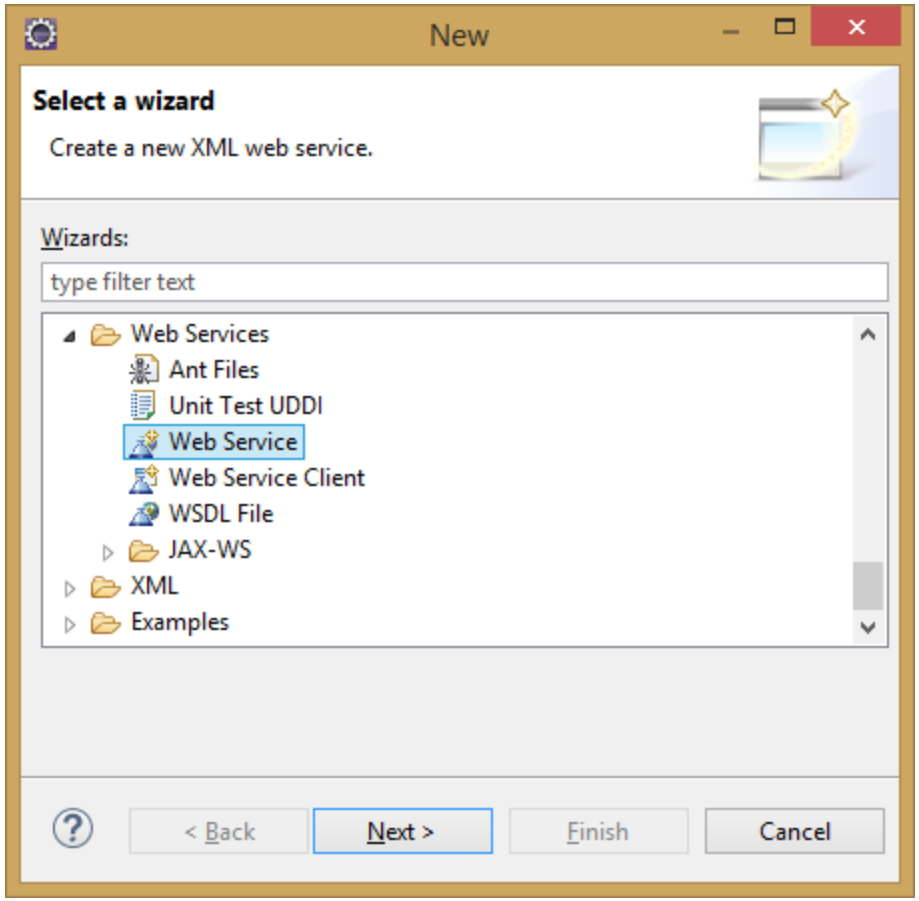


HelloWebService.java | Web Services Test Client

```
1  package org.netb453.webservices;
2
3  import javax.jws.*;
4
5  @WebService()
6  public class HelloWebService {
7      @WebMethod( operationName = "welcome" )
8      public String sayHelloWorld(@WebParam( name = "name" ) String name){
9          return "Hello world from "+ name;
10     }
11 }
12
```

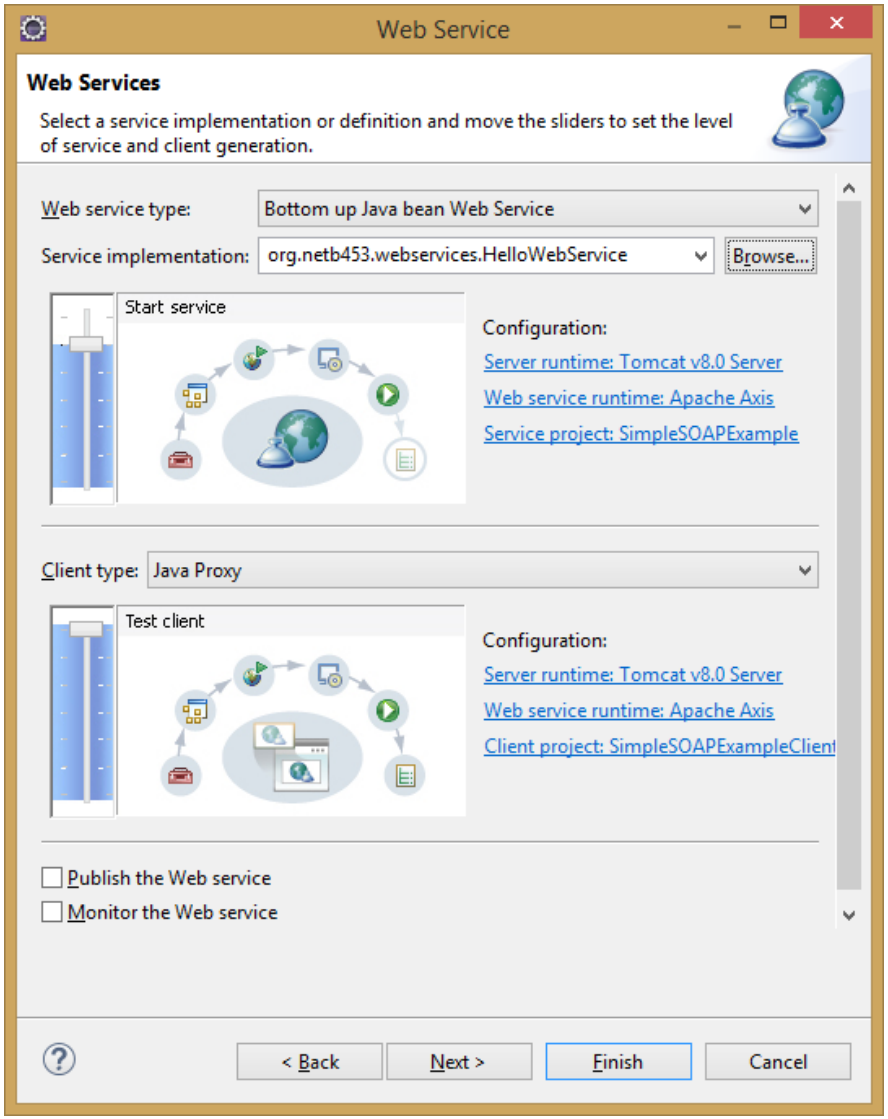
15.7 Creating web services in eclipse

4.Right click on Project->New->Web service



15.7 Creating web services in eclipse

5.Click on next.

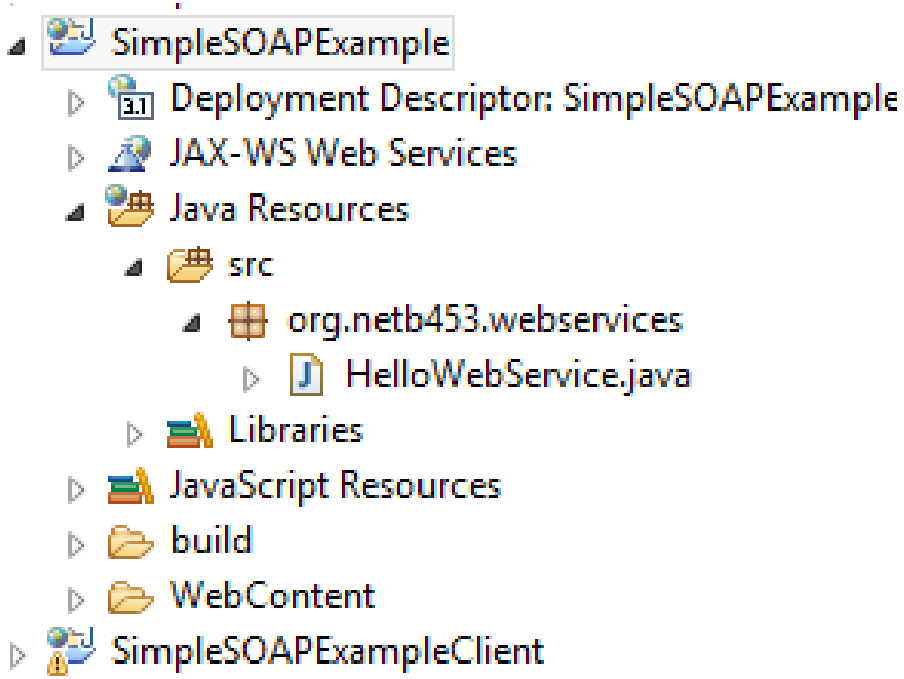


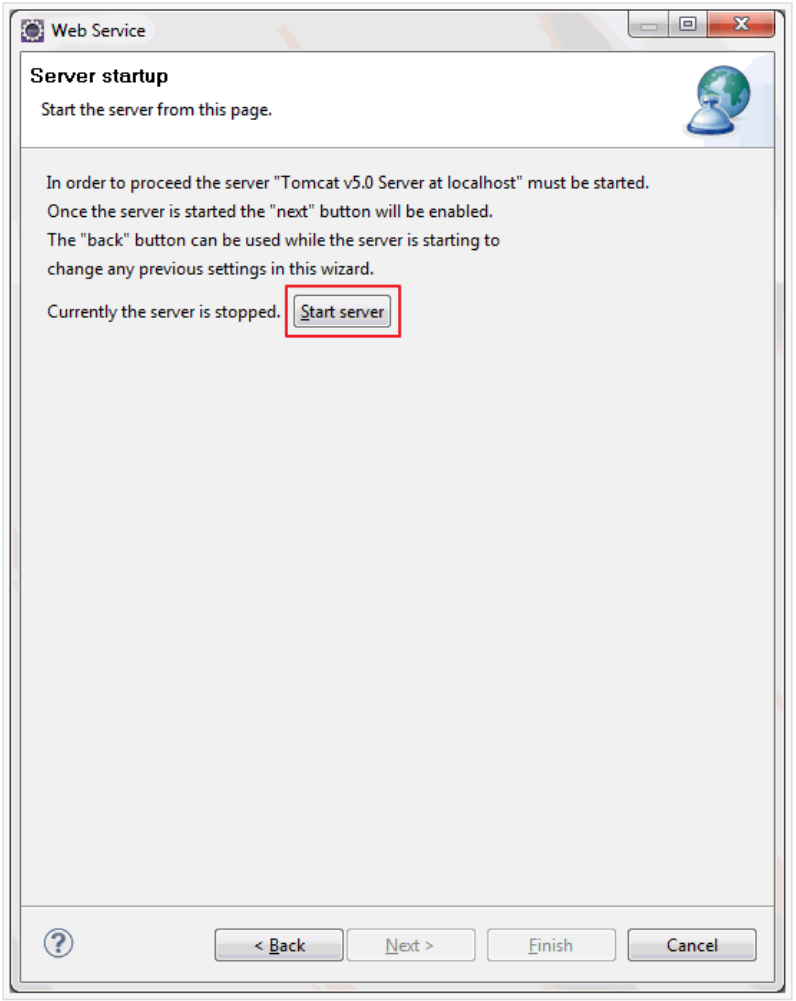
15.7 Creating web services in eclipse

In service implementation text box, browse to get the fully qualified class name of above created class(**HelloWebService.java**) and move **both above slider to maximum level** (i.e. Test service and Test Client level) and click on **Finish**.

15. Click on start server.

15.7 Creating web services in eclipse

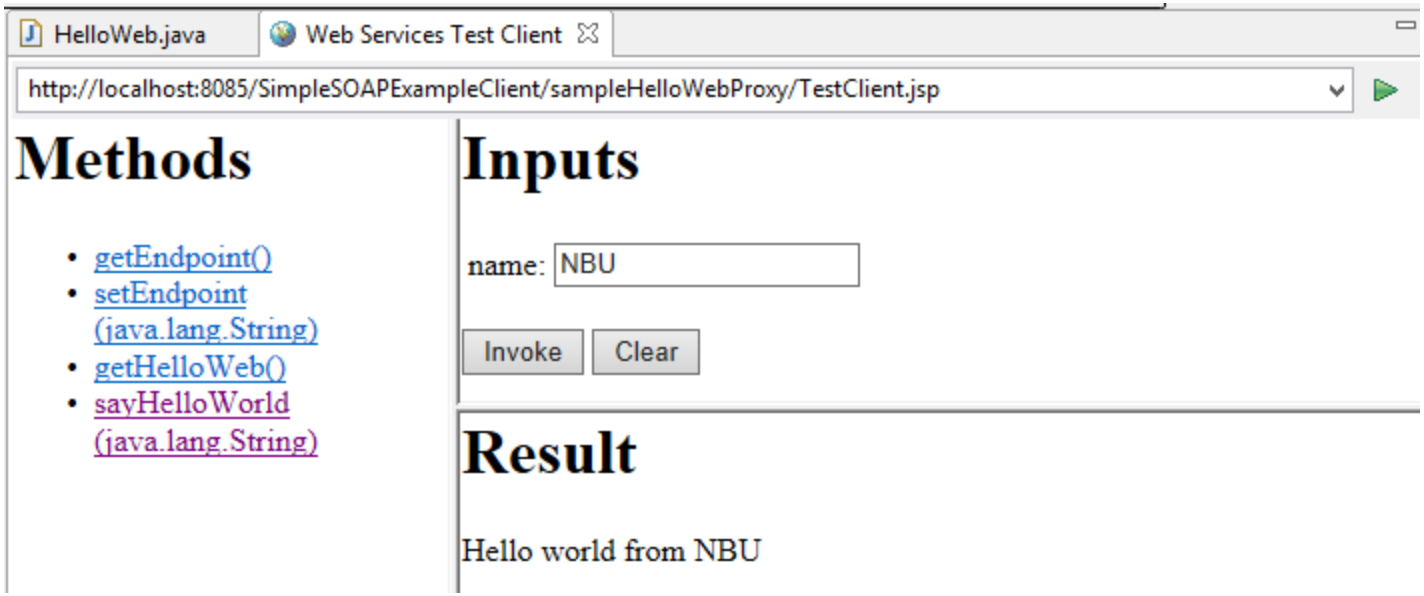




15.7 Creating web services in eclipse

7.After clicking start server, eclipse will open test web service API.

With this test API, you can test your web service



15.9 Session Tracking in a SOAP-Based Web Service

- Section 15.8 described the advantages of using session tracking to maintain client-state information so you can personalize the users' browsing experiences.
- Now we'll incorporate session tracking into a web service.
- Storing session information also enables a web service to distinguish between clients.

15.9.1 Creating a `Blackjack` Web Service

- Our next example is a web service that assists you in developing a blackjack card game.
- The `Blackjack` web service (Fig. 31.17) provides web methods to shuffle a deck of cards, deal a card from the deck and evaluate a hand of cards.
- The web service (Fig. 31.17) stores each card as a `String` consisting of a number, `1`–`13`, representing the card's face (ace through king, respectively), followed by a space and a digit, `0`–`3`, representing the card's suit (hearts, diamonds, clubs or spades, respectively).
- For example, the jack of clubs is represented as `"11 2"` and the two of hearts as `"2 0"`.
- To create and deploy this web service, follow the steps that we presented in Sections 15.15.2–15.15.3 for the `WelcomeSOAP` service.

```
1  // Fig. 31.17: Blackjack.java
2  // Blackjack web service that deals cards and evaluates hands
3  package com.deitel.blackjack;
4
5  import com.sun.xml.ws.developer.servlet.HttpSessionScope;
6  import java.util.ArrayList;
7  import java.util.Random;
8  import javax.jws.WebMethod;
9  import javax.jws.WebParam;
10 import javax.jws.WebService;
11
12 @HttpSessionScope // enable web service to maintain session state
13 @WebService()
14 public class Blackjack
15 {
16    private ArrayList< String > deck; // deck of cards for one user session
17    private static final Random randomObject = new Random();
18
```

Fig. 31.17 | `Blackjack` web service that deals cards and evaluates hands. (Part 1 of 6.)

```
19      // deal one card
20      @WebMethod( operationName = "dealCard" )
21      public String dealCard()
22      {
23         String card = "";
24         card = deck.get( 0 ); // get top card of deck
25         deck.remove( 0 ); // remove top card of deck
26         return card;
27      } // end WebMethod dealCard
28
29      // shuffle the deck
30      @WebMethod( operationName = "shuffle" )
31      public void shuffle()
32      {
33         // create new deck when shuffle is called
34         deck = new ArrayList< String >();
35
```

Fig. 31.17 | `Blackjack` web service that deals cards and evaluates hands. (Part 2 of 6.)

```
36          // populate deck of cards
37          for ( int face = 1; face <= 13; face++ ) // loop through faces
38             for ( int suit = 0; suit <= 3; suit++ ) // loop through suits
39                deck.add( face + " " + suit ); // add each card to deck
40
41          String tempCard; // holds card temporarily during swapping
42          int index; // index of randomly selected card
43
44          for ( int i = 0; i < deck.size() ; i++ ) // shuffle
45          {
46             index = randomObject.nextInt( deck.size() - 1 );
47
48             // swap card at position i with randomly selected card
49             tempCard = deck.get( i );
50             deck.set( i, deck.get( index ) );
51             deck.set( index, tempCard );
52          } // end for
53       } // end WebMethod shuffle
54
```

Fig. 31.17 | `Blackjack` web service that deals cards and evaluates hands. (Part 3 of 6.)

```
55      // determine a hand's value
56      @WebMethod( operationName = "getHandValue" )
57      public int getHandValue( @WebParam( name = "hand" ) String hand )
58      {
59         // split hand into cards
60         String[] cards = hand.split( "\t" );
61         int total = 0; // total value of cards in hand
62         int face; // face of current card
63         int aceCount = 0; // number of aces in hand
64
65         for ( int i = 0; i < cards.length; i++ )
66         {
67            // parse string and get first int in String
68            face = Integer.parseInt(
69               cards[ i ].substring( 0, cards[ i ].indexOf( " " ) ) );
70
```

Fig. 31.17 | `Blackjack` web service that deals cards and evaluates hands. (Part 4 of 6.)

```
71            switch ( face )
72            {
73               case 1: // if ace, increment aceCount
74                  ++aceCount;
75                  break;
76               case 11: // jack
77               case 12: // queen
78               case 13: // king
79                  total += 10;
80                  break;
81               default: // otherwise, add face
82                  total += face;
83                  break;
84            } // end switch
85         } // end for
86
```

Fig. 31.17 | `Blackjack` web service that deals cards and evaluates hands. (Part 5 of 6.)

```
87         // calculate optimal use of aces
88         if ( aceCount > 0 )
89         {
90            // if possible, count one ace as 11
91            if ( total + 11 + aceCount - 1 <= 21 )
92               total += 11 + aceCount - 1;
93            else // otherwise, count all aces as 1
94               total += aceCount;
95          } // end if
96
97         return total;
98      } // end WebMethod getHandValue
99   } // end class Blackjack
```

Fig. 31.17 | `Blackjack` web service that deals cards and evaluates hands. (Part 6 of 6.)

15.9.1 Creating a Blackjack Web Service (cont.)

- To enable session tracking in a web service in JAX-WS 2.2, precede your web service class with the @HttpSessionScope- annotation.
  - Annotation is located in package `com.sun.xml.ws.developer.servlet`.
- Add the JAX-WS 2.2 library to your project.
  - Right click the Libraries node in your `Blackjack` web application project and select Add Library…
  - In the dialog that appears, locate and select JAX-WS 2.2, then click Add Library.
  - Once a web service is annotated with `@HttpSessionScope`, the server **automatically maintains a separate instance of the class for each client session**. The deck instance variable (line 16) will be maintained separately for each client.

15.9.2 Consuming the Blackjack Web Service

- The blackjack application in Fig. 15.18 keeps track of the player's and dealer's cards, and the web service tracks the cards that have been dealt.

```
1  // Fig. 31.18: BlackjackGameJFrame.java
2  // Blackjack game that uses the Blackjack Web Service.
3  package com.deitel.blackjackclient;
4
5  import com.deitel.blackjack.Blackjack;
6  import com.deitel.blackjack.BlackjackService;
7  import java.awt.Color;
8  import java.util.ArrayList;
9  import javax.swing.ImageIcon;
10 import javax.swing.JLabel;
11 import javax.swing.JOptionPane;
12 import javax.xml.ws.BindingProvider;
13
14 public class BlackjackGameJFrame extends javax.swing.JFrame
15 {
16    private String playerCards;
17    private String dealerCards;
18    private ArrayList<JLabel> cardboxes; // list of card image JLabels
19    private int currentPlayerCard; // player's current card number
20    private int currentDealerCard; // blackjackProxy's current card number
21    private BlackjackService blackjackService; // used to obtain proxy
22    private Blackjack blackjackProxy; // used to access the web service
23
```

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 1 of 24.)

```
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
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
```

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 2 of 24.)

```
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( dealerCard1JLabel );
62         cardboxes.add( dealerCard2JLabel );
63         cardboxes.add( dealerCard3JLabel );
64         cardboxes.add( dealerCard4JLabel );
65         cardboxes.add( dealerCard5JLabel );
```

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 3 of 24.)

```
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 constructor
84
```

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 4 of 24.)

```
85      // play the dealer's hand
86      private void dealerPlay()
87      {
88         try
89         {
90            // while the value of the dealers'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            {
97               displayCard( i, cards[i] );
98            }
99
100           while ( blackjackProxy.getHandValue( dealerCards ) < 17 )
101           {
102              String newCard = blackjackProxy.dealCard(); // deal new card
103              dealerCards += "\t" + newCard; // deal new card
104              displayCard( currentDealerCard, newCard );
105              ++currentDealerCard;
106              JOptionPane.showMessageDialog( this, "Dealer takes a card",
107                 "Dealer's turn", JOptionPane.PLAIN_MESSAGE );
108           } // end while
```

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 5 of 24.)

```
109
110          int dealersTotal = blackjackProxy.getHandValue( dealerCards );
111          int playersTotal = blackjackProxy.getHandValue( playerCards );
112
113          // if dealer busted, player wins
114          if ( dealersTotal > 21 )
115          {
116             gameOver( GameStatus.WIN );
117             return;
118          } // end if
119
```

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 6 of 24.)

```
120          // if dealer and player are below 21
121          // higher score wins, equal scores is a push
122          if ( dealersTotal > playersTotal )
123          {
124             gameOver( GameStatus.LOSE );
125          }
126          else if ( dealersTotal < playersTotal )
127          {
128             gameOver( GameStatus.WIN );
129          }
130          else
131          {
132             gameOver( GameStatus.PUSH );
133          }
134       } // end try
135       catch ( Exception e )
136       {
137          e.printStackTrace();
138       } // end catch
139    } // end method dealerPlay
140
```

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 7 of 24.)

```
141     // displays the card represented by cardValue in specified JLabel
142     private void displayCard( int card, String cardValue )
143     {
144        try
145        {
146           // retrieve correct JLabel from cardBoxes
147           JLabel displayLabel = cardboxes.get( card );
148
149           // if string representing card is empty, display back of card
150           if ( cardValue.equals( "" ) )
151           {
152              displayLabel.setIcon( new ImageIcon( getClass().getResource(
153                 "/com/deitel/java/blackjackclient/" +
154                 "blackjack_images/cardback.png" ) ) );
155              return;
156           } // end if
157
```

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 8 of 24.)

```
158          // retrieve the face value of the card
159          String face = cardValue.substring( 0, cardValue.indexOf( " " ) );
160
161          // retrieve the suit of the card
162          String suit =
163             cardValue.substring( cardValue.indexOf( " " ) + 1 );
164
165          char suitLetter; // suit letter used to form image file
166
167          switch ( Integer.parseInt( suit ) )
168          {
169             case 0: // hearts
170                suitLetter = 'h';
171                break;
172             case 1: // diamonds
173                suitLetter = 'd';
174                break;
175             case 2: // clubs
176                suitLetter = 'c';
177                break;
178             default: // spades
179                suitLetter = 's';
180                break;
181          } // end switch
```

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 9 of 24.)

```
182
183          // set image for displayLabel
184          displayLabel.setIcon( new ImageIcon( getClass().getResource(
185             "/com/deitel/java/blackjackclient/blackjack_images/" +
186             face + suitLetter + ".png" ) ) );
187       } // end try
188       catch ( Exception e )
189       {
190          e.printStackTrace();
191       } // end catch
192    } // end method displayCard
193
194    // displays all player cards and shows appropriate message
195    private void gameOver( GameStatus winner )
196    {
197       String[] cards = dealerCards.split( "\t" );
198
199       // display blackjackProxy's cards
200       for ( int i = 0; i < cards.length; i++ )
201       {
202          displayCard( i, cards[i] );
203       }
204
```

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 10 of 24.)

```
205         // display appropriate status image
206         if ( winner == GameStatus.WIN )
207         {
208            statusJLabel.setText( "You win!" );
209         }
210         else if ( winner == GameStatus.LOSE )
211         {
212            statusJLabel.setText( "You lose." );
213         }
214         else if ( winner == GameStatus.PUSH )
215         {
216            statusJLabel.setText( "It's a push." );
217         }
218         else // blackjack
219         {
220            statusJLabel.setText( "Blackjack!" );
221         }
222
223         // display final scores
224         int dealersTotal = blackjackProxy.getHandValue( dealerCards );
225         int playersTotal = blackjackProxy.getHandValue( playerCards );
226         dealerTotalJLabel.setText( "Dealer: " + dealersTotal );
227         playerTotalJLabel.setText( "Player: " + playersTotal );
228
```

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 11 of 24.)

```
229         // reset for new game
230         standJButton.setEnabled( false );
231         hitJButton.setEnabled( false );
232         dealJButton.setEnabled( true );
233      } // end method gameOver
234
235      // The initComponents method is autogenerated by NetBeans and is called
236      // from the constructor to initialize the GUI. This method is not shown
237      // here to save space. Open BlackjackGameJFrame.java in this
238      // example's folder to view the complete generated code
239
240      // handles dealJButton click
241      private void dealJButtonActionPerformed(
242         java.awt.event.ActionEvent evt )
243      {
244         String card; // stores a card temporarily until it's added to a hand
245
```

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 12 of 24.)

```
246         // clear card images
247         for ( int i = 0; i < cardboxes.size(); i++ )
248         {
249            cardboxes.get( i ).setIcon( null );
250         }
251
252         statusJLabel.setText( "" );
253         dealerTotalJLabel.setText( "" );
254         playerTotalJLabel.setText( "" );
255
256         // create a new, shuffled deck on remote machine
257         blackjackProxy.shuffle();
258
259         // deal two cards to player
260         playerCards = blackjackProxy.dealCard(); // add first card to hand
261         displayCard( 11, playerCards ); // display first card
262         card = blackjackProxy.dealCard(); // deal second card
263         displayCard( 12, card ); // display second card
264         playerCards += "\t" + card; // add second card to hand
265
```

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 13 of 24.)

```
266         // deal two cards to blackjackProxy, but only show first
267         dealerCards = blackjackProxy.dealCard(); // add first card to hand
268         displayCard( 0, dealerCards ); // display first card
269         card = blackjackProxy.dealCard(); // deal second card
270         displayCard( 1, "" ); // display back of card
271         dealerCards += "\t" + card; // add second card to hand
272
273         standJButton.setEnabled( true );
274         hitJButton.setEnabled( true );
275         dealJButton.setEnabled( false );
276
277         // determine the value of the two hands
278         int dealersTotal = blackjackProxy.getHandValue( dealerCards );
279         int playersTotal = blackjackProxy.getHandValue( playerCards );
280
```

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 14 of 24.)

```
281         // if hands both equal 21, it is a push
282         if ( playersTotal == dealersTotal && playersTotal == 21 )
283         {
284            gameOver( GameStatus.PUSH );
285         }
286         else if ( dealersTotal == 21 ) // blackjackProxy has blackjack
287         {
288            gameOver( GameStatus.LOSE );
289         }
290         else if ( playersTotal == 21 ) // blackjack
291         {
292            gameOver( GameStatus.BLACKJACK );
293         }
294
295         // next card for blackjackProxy has index 2
296         currentDealerCard = 2;
297
298         // next card for player has index 13
299         currentPlayerCard = 13;
300      } // end method dealJButtonActionPerformed
301
```

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 15 of 24.)

```
302     // handles standJButton click
303     private void hitJButtonActionPerformed(
304        java.awt.event.ActionEvent evt )
305     {
306        // get player another card
307        String card = blackjackProxy.dealCard(); // deal new card
308        playerCards += "\t" + card; // add card to hand
309
310        // update GUI to display new card
311        displayCard( currentPlayerCard, card );
312        ++currentPlayerCard;
313
314        // determine new value of player's hand
315        int total = blackjackProxy.getHandValue( playerCards );
316
317        if ( total > 21 ) // player busts
318        {
319           gameOver( GameStatus.LOSE );
320        }
321        else if ( total == 21 ) // player cannot take any more cards
322        {
323           hitJButton.setEnabled( false );
324           dealerPlay();
325        } // end if
326     } // end method hitJButtonActionPerformed
```

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 16 of 24.)

```
327
328    // handles standJButton click
329    private void standJButtonActionPerformed(
330       java.awt.event.ActionEvent evt )
331    {
332       standJButton.setEnabled( false );
333       hitJButton.setEnabled( false );
334       dealJButton.setEnabled( true );
335       dealerPlay();
336    } // end method standJButtonActionPerformed
337
338    // begins application execution
339    public static void main( String args[] )
340    {
341       java.awt.EventQueue.invokeLater(
342          new Runnable()
343          {
344             public void run()
345             {
346                new BlackjackGameJFrame().setVisible( true );
347             }
348          }
349       ); // end call to java.awt.EventQueue.invokeLater
350    } // end main
```

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 17 of 24.)

```
351
352     // Variables declaration - do not modify
353     private javax.swing.JButton dealJButton;
354     private javax.swing.JLabel dealerCard10JLabel;
355     private javax.swing.JLabel dealerCard11JLabel;
356     private javax.swing.JLabel dealerCard1JLabel;
357     private javax.swing.JLabel dealerCard2JLabel;
358     private javax.swing.JLabel dealerCard3JLabel;
359     private javax.swing.JLabel dealerCard4JLabel;
360     private javax.swing.JLabel dealerCard5JLabel;
361     private javax.swing.JLabel dealerCard6JLabel;
362     private javax.swing.JLabel dealerCard7JLabel;
363     private javax.swing.JLabel dealerCard8JLabel;
364     private javax.swing.JLabel dealerCard9JLabel;
365     private javax.swing.JLabel dealerJLabel;
366     private javax.swing.JLabel dealerTotalJLabel;
367     private javax.swing.JButton hitJButton;
368     private javax.swing.JLabel playerCard10JLabel;
369     private javax.swing.JLabel playerCard11JLabel;
370     private javax.swing.JLabel playerCard1JLabel;
371     private javax.swing.JLabel playerCard2JLabel;
372     private javax.swing.JLabel playerCard3JLabel;
373     private javax.swing.JLabel playerCard4JLabel;
374     private javax.swing.JLabel playerCard5JLabel;
```

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 18 of 24.)

```
375     private javax.swing.JLabel playerCard6JLabel;
376     private javax.swing.JLabel playerCard7JLabel;
377     private javax.swing.JLabel playerCard8JLabel;
378     private javax.swing.JLabel playerCard9JLabel;
379     private javax.swing.JLabel playerJLabel;
380     private javax.swing.JLabel playerTotalJLabel;
381     private javax.swing.JButton standJButton;
382     private javax.swing.JLabel statusJLabel;
383     // End of variables declaration
384 } // end class BlackjackGameJFrame
```

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 19 of 24.)

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

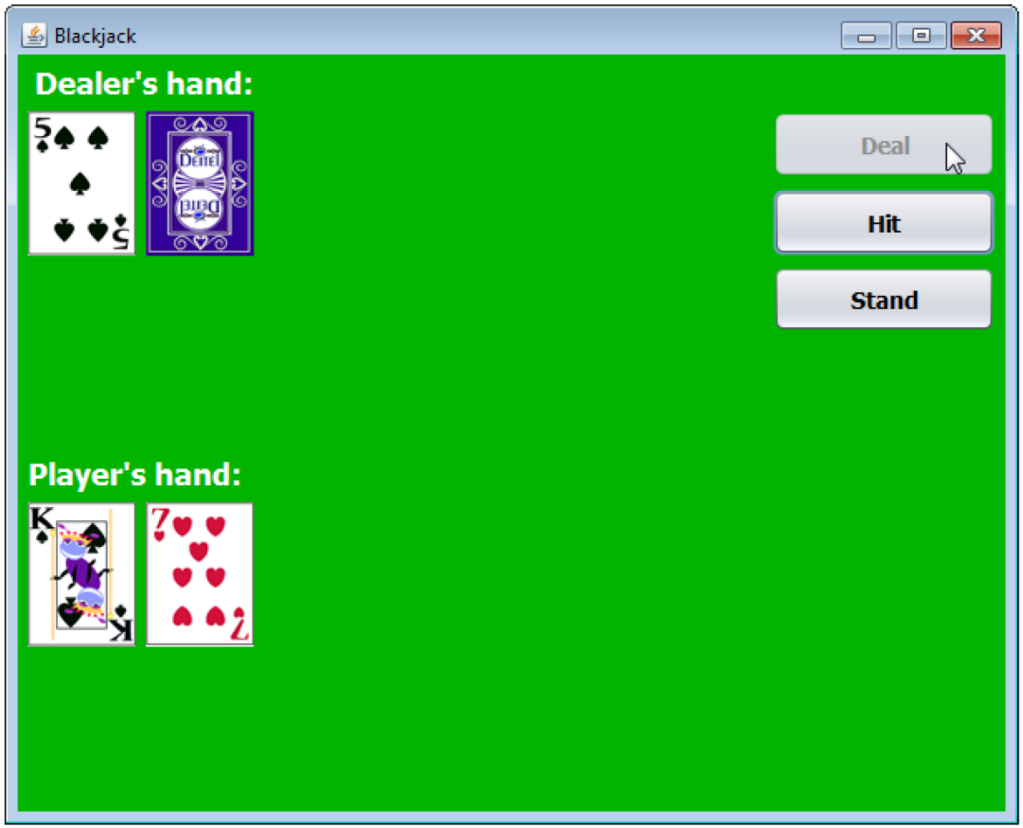


Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 20 of 24.)

b) Dealer and player hands after the user clicks **Stand**. In this case, the result is a push

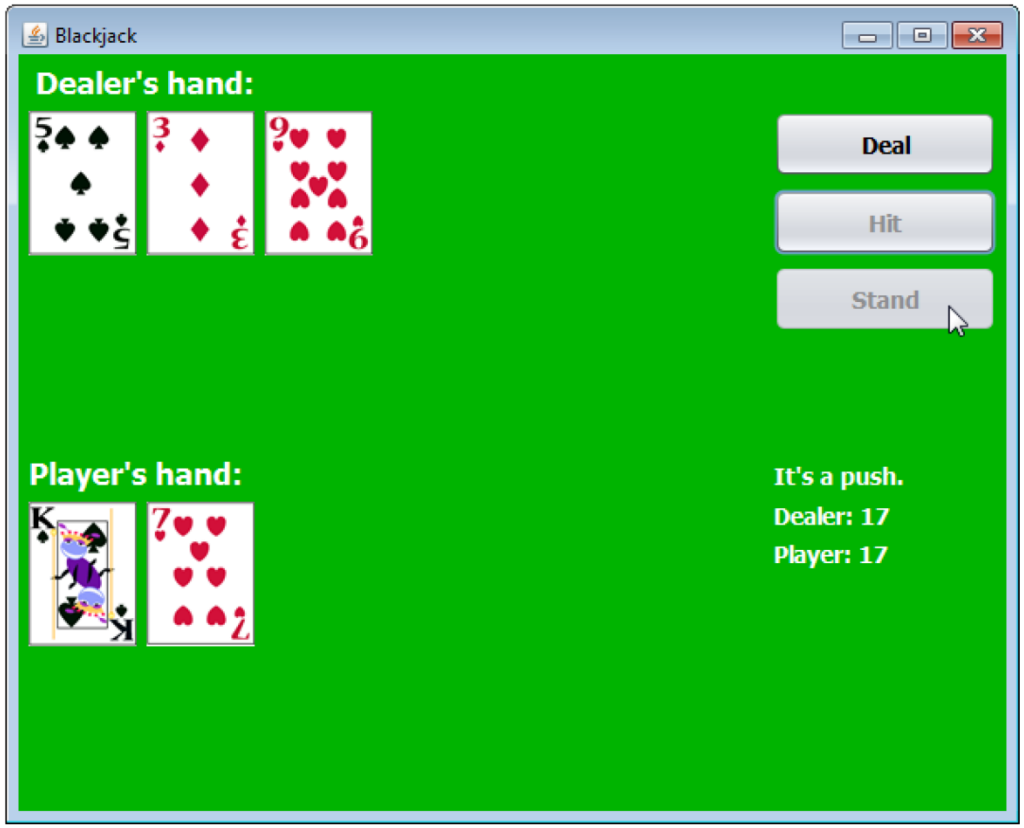


Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 21 of 24.)

c) Dealer and player hands after the user clicks **Hit** and draws 21. In this case, the player wins

Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 22 of 24.)

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

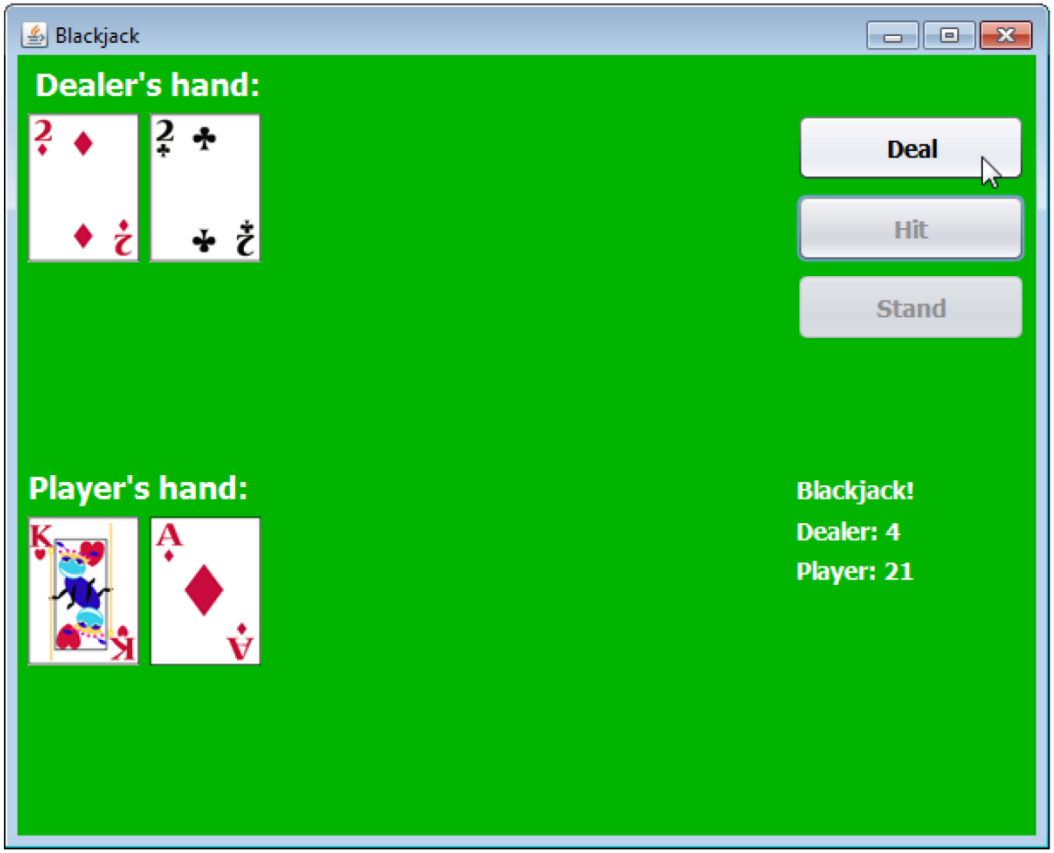


Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 23 of 24.)

e) Dealer and player hands after the dealer is dealt blackjack

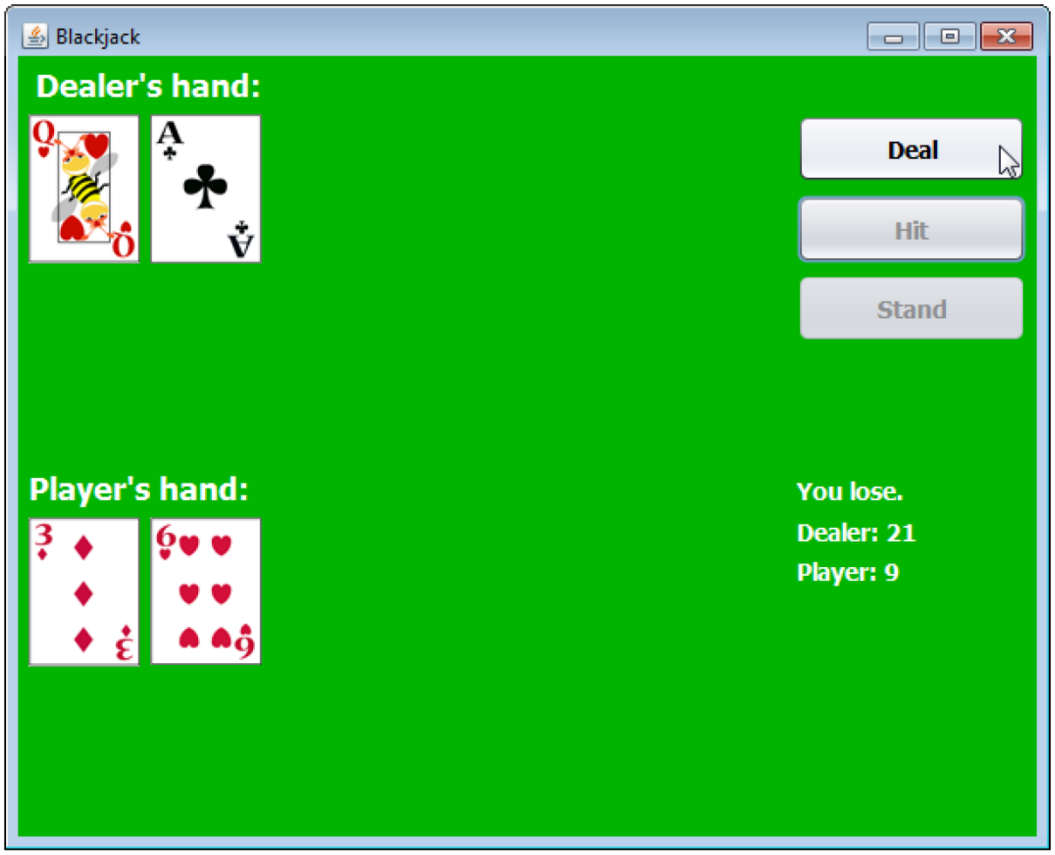


Fig. 31.18 | Blackjack game that uses the `Blackjack` web service. (Part 24 of 24.)

15.9.2 Consuming the `Blackjack` Web Service (cont.)

- When interacting with a JAX-WS web service that performs session tracking, the client application must indicate whether it wants to allow the web service to maintain session information.
- We first cast the service endpoint interface object to interface type `BindingProvider`.
- A BindingProvider enables the client to manipulate the request information that will be sent to the server.
- This information is stored in an object that implements interface RequestContext.
- The `BindingProvider` and `RequestContext` are part of the framework that is created by the IDE when you add a web service client to the application.

15.9.2 Consuming the Blackjack Web Service (cont.)

- Next, we invoke the `BindingProvider`'s getRequestContext method to obtain the `RequestContext` object.
- Then we call the `RequestContext`'s put method to set the property

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

7. Problems

1. **Write a SOAP web service that encrypts a String using the transposition cypher http://www.vectorsite.net/ttcode_01.html**
2. **Write a SOAP web service that decrypts a String using the transposition cypher http://www.vectorsite.net/ttcode_01.html**