13

Remote Method Invocation

Objectives

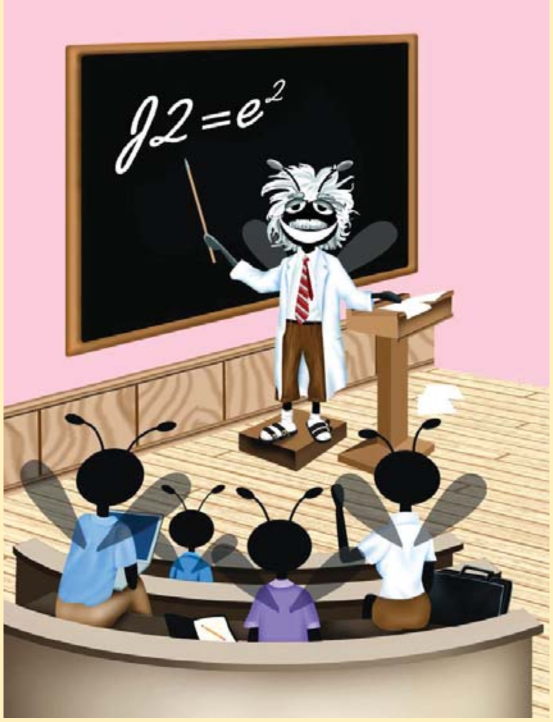
- To understand the distributed computing concepts.
- To understand the architecture of RMI.
- To be able to use activatable RMI objects to build resilient distributed systems.
- To understand how to use RMI callbacks.
- To be able to build RMI clients that download necessary classes dynamically.
- To be able to build activatable RMI objects.

Dealing with more than one client at a time is the business world's equivalent of bigamy. It's so awkward to tell one client that you're working on someone else's business that you inevitably start lying.
Andrew Frothingham

They also serve who only stand and wait.
John Milton

Rule 1: The client is always right.
Rule 2: If you think the client is wrong, see Rule 1.
Sign seen in shops

I love being a writer. What I can't stand is the paperwork.
Peter De Vries



Outline

13.1 Introduction

In this chapter, we introduce Java's distributed computing capabilities with *Remote Method Invocation* (*RMI*). RMI allows Java objects running on separate computers or in separate processes to communicate with one another via *remote method calls*. Such method calls appear to the programmer the same as those operating on objects in the same program.

RMI is based on a similar, earlier technology for procedural programming called *remote procedure calls* (*RPCs*) developed in the 1980s. RPC allows a procedural program (i.e., a program written in C or another procedural programming language) to call a function residing on another computer as conveniently as if that function were part of the same program running on the same computer. A goal of RPC was to allow programmers to concentrate on the required tasks of an application by calling functions, while making the mechanism that allows the application's parts to communicate over a network transparent to the programmer. RPC performs all the networking and *marshaling of data* (i.e., packaging of function arguments and return values for transmission over a network). A disadvantage of RPC is that it supports a limited set of simple data types. Therefore, RPC is not suitable for passing and returning Java objects. Another disadvantage of RPC is that it requires the programmer to learn a special *interface definition language* (*IDL*) to describe the functions that can be invoked remotely.

RMI is Java's implementation of RPC for Java-object-to-Java-object distributed communication. Once a Java object registers as being remotely accessible (i.e., it is a *remote object*), a client can obtain a remote reference to that object, which allows the client to use that object remotely. The method call syntax is identical to the syntax for calling methods of other objects in the same program. As with RPC, RMI handles the marshaling of data across the network. However, RMI also enables Java programs to transfer complete Java objects using Java's object-serialization mechanism. The programmer need not be concerned with the transmission of the data over the network. RMI does not require the programmer to learn an IDL, because the J2SE SDK includes tools for generating all the networking code from the program's interface definitions. Also, because RMI supports only Java, no language-neutral IDL is required; Java's own interfaces are sufficient.

We present two substantial RMI examples and discuss the key concepts of RMI as we encounter them throughout the examples. After studying these examples, you should have an understanding of the RMI networking model and should be able to take advantage of advanced RMI features for building Java-to-Java distributed applications.

[*Note*: For Java-to-non-Java communication, you can use Java IDL (introduced in Java 1.2) or RMI-IIOP. Java IDL and RMI-IIOP enable applications and applets written in Java to communicate with objects written in any language that supports CORBA (Common Object Request Broker Architecture). Please see Chapter 26, CORBA: Part 1 and Chapter 27, CORBA: Part 2 for our discussion of CORBA and RMI-IIOP.]

13.2 Case Study: Creating a Distributed System with RMI

In the next several sections, we present an RMI example that downloads the *Traveler's Forecast* weather information from the National Weather Service Web site:

```
http://iwin.nws.noaa.gov/iwin/us/traveler.html
```

[*Note*: As we developed this example, the format of the *Traveler's Forecast* Web page changed several times (a common occurrence with today's dynamic Web pages). The information we use in this example depends directly on the format of the *Traveler's Forecast* Web page. If you have trouble running this example, please refer to the FAQ page on our Web site, **`www.deitel.com`**.]

We store the *Traveler's Forecast* information in an RMI remote object that accepts requests for weather information through remote method calls.

The four major steps in this example include:

1. Defining a *remote interface* that declares methods that clients can invoke on the remote object.
2. Defining the *remote object implementation* for the remote interface. [*Note*: By convention, the remote object implementation class has the same name as the remote interface and ends with ***`Impl`***.]
3. Defining the client application that uses a *remote reference* to interact with the interface implementation (i.e., an object of the class that implements the remote interface).
4. Compiling and executing the remote object and the client.

13.3 Defining the Remote Interface

The first step in creating a distributed application with RMI is to define the remote interface that describes the *remote methods* through which the client interacts with the remote object using RMI. To create a remote interface, define an interface that extends interface ***`java.rmi.Remote`***. Interface **`Remote`** is a *tagging interface*—it does not declare any methods, and therefore places no burden on the implementing class. An object of a class that implements interface **`Remote`** directly or indirectly is a *remote object* and can be accessed—with appropriate security permissions—from any Java virtual machine that has a connection to the computer on which the remote object executes.

Software Engineering Observation 13.1

*Every remote method must be declared in an interface that extends **`java.rmi.Remote`**.*

Software Engineering Observation 13.2

*An RMI distributed application must export an object of a class that implements the **`Remote`** interface to make that remote object available to receive remote method calls.*

Interface **`WeatherService`** (Fig. 13.1)—which extends interface **`Remote`** (line 10)—is the remote interface for our remote object. Line 13 declares method **`getWeatherInformation`**, which clients can invoke to retrieve weather information from the remote object. Note that although the **`WeatherService`** remote interface defines only one method, remote interfaces can declare multiple methods. A remote object must implement all methods declared in its remote interface.

When computers communicate over networks, there exists the potential for communication problems. For example, a server computer could malfunction, or a network resource could malfunction. If a communication problem occurs during a remote method call, the remote method throws a ***`RemoteException`***, which is a checked exception.

Software Engineering Observation 13.3

*Each method in a **`Remote`** interfaces must have a **`throws`** clause that indicates that the method can throw **`RemoteException`**s.*

Software Engineering Observation 13.4

*RMI uses Java's default serialization mechanism to transfer method arguments and return values across the network. Therefore, all method arguments and return values must be **`Serializable`** or primitive types.*

13.4 Implementing the Remote Interface

The next step is to define the remote object implementation. Class **`WeatherServiceImpl`** (Fig. 13.2) is the remote object class that implements the **`WeatherService`** remote interface. The client interacts with an object of class **`WeatherServiceImpl`** by invoking

```
1   // WeatherService.java
2   // WeatherService interface declares a method for obtaining
3   // wether information.
4   package com.deitel.advjhtp1.rmi.weather;
5
6   // Java core packages
7   import java.rmi.*;
8   import java.util.*;
9
10  public interface WeatherService extends Remote {
11
12     // obtain List of WeatherBean objects from server
13     public List getWeatherInformation() throws RemoteException;
14
15  }
```

Fig. 13.1 **`WeatherService`** interface.

method **getWeatherInformation** of interface **WeatherService** to obtain weather information. Class **WeatherServiceImpl** stores weather data in a **List** of **WeatherBean** (Fig. 13.3) objects. When a client invokes remote method **getWeatherInformation**, the **WeatherServiceImpl** returns a reference to the **List** of **WeatherBean**s. The RMI system returns a serialized copy of the **List** to the client. The RMI system then de-serializes the **List** on the receiving end and provides the caller with a reference to the **List**.

The National Weather Service updates the Web page from which we retrieve information twice a day. However, class **WeatherServiceImpl** downloads this information only once, when the server starts. The exercises ask you to modify the server to update the data twice a day. [*Note*: **WeatherServiceImpl** is the class affected if the National Weather Service changes the format of the *Traveler's Forecast* Web page. If you encounter problems with this example, visit the FAQ page at our Web site **www.deitel.com**.]

```java
// WeatherServiceImpl.java
// WeatherServiceImpl implements the WeatherService remote
// interface to provide a WeatherService remote object.
package com.deitel.advjhtp1.rmi.weather;

// Java core packages
import java.io.*;
import java.net.URL;
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public class WeatherServiceImpl extends UnicastRemoteObject
   implements WeatherService {

   private List weatherInformation;  // WeatherBean object List

   // initialize server
   public WeatherServiceImpl() throws RemoteException
   {
      super();
      updateWeatherConditions();
   }

   // get weather information from NWS
   private void updateWeatherConditions()
   {
      try {
         System.out.println( "Update weather information..." );

         // National Weather Service Traveler's Forecast page
         URL url = new URL(
            "http://iwin.nws.noaa.gov/iwin/us/traveler.html" );

```

Fig. 13.2 **WeatherServiceImpl** class implements remote interface **WeatherService** (part 1 of 3).

```
         // create BufferedReader for reading Web page contents
         BufferedReader in = new BufferedReader(
            new InputStreamReader( url.openStream() ) );

         // separator for starting point of data on Web page
         String separator = "TAV12";

         // locate separator string in Web page
         while ( !in.readLine().startsWith( separator ) )
            ;    // do nothing

         // strings representing headers on Traveler's Forecast
         // Web page for daytime and nighttime weather
         String dayHeader =
            "CITY            WEA     HI/LO   WEA     HI/LO";
         String nightHeader =
            "CITY            WEA     LO/HI   WEA     LO/HI";

         String inputLine = "";

         // locate header that begins weather information
         do {
            inputLine = in.readLine();
         } while ( !inputLine.equals( dayHeader ) &&
                   !inputLine.equals( nightHeader ) );

         weatherInformation = new ArrayList(); // create List

         // create WeatherBeans containing weather data and
         // store in weatherInformation List
         inputLine = in.readLine();  // get first city's info

         // The portion of inputLine containing relevant data is
         // 28 characters long. If the line length is not at
         // least 28 characters long, done processing data.
         while ( inputLine.length() > 28 ) {

            // Create WeatherBean object for city. First 16
            // characters are city name. Next, six characters
            // are weather description. Next six characters
            // are HI/LO or LO/HI temperature.
            WeatherBean weather = new WeatherBean(
               inputLine.substring( 0, 16 ),
               inputLine.substring( 16, 22 ),
               inputLine.substring( 23, 29 ) );

            // add WeatherBean to List
            weatherInformation.add( weather );

            inputLine = in.readLine();  // get next city's info
         }

```

Fig. 13.2 **WeatherServiceImpl** class implements remote interface **WeatherService** (part 2 of 3).

```
87           in.close();  // close connection to NWS Web server
88  
89           System.out.println( "Weather information updated." );
90  
91        } // end method updateWeatherConditions
92  
93        // handle exception connecting to National Weather Service
94        catch( java.net.ConnectException connectException ) {
95           connectException.printStackTrace();
96           System.exit( 1 );
97        }
98  
99        // process other exceptions
100       catch( Exception exception ) {
101          exception.printStackTrace();
102          System.exit( 1 );
103       }
104    }
105 
106    // implementation for WeatherService interface remote method
107    public List getWeatherInformation() throws RemoteException
108    {
109       return weatherInformation;
110    }
111 
112    // launch WeatherService remote object
113    public static void main( String args[] ) throws Exception
114    {
115       System.out.println( "Initializing WeatherService..." );
116 
117       // create remote object
118       WeatherService service = new WeatherServiceImpl();
119 
120       // specify remote object name
121       String serverObjectName = "rmi://localhost/WeatherService";
122 
123       // bind WeatherService remote object in RMI registry
124       Naming.rebind( serverObjectName, service );
125 
126       System.out.println( "WeatherService running." );
127    }
128 }
```

Fig. 13.2 **WeatherServiceImpl** class implements remote interface **WeatherService** (part 3 of 3).

Class **WeatherServiceImpl** extends class ***UnicastRemoteObject*** (package **java.rmi.server**) and implements **Remote** interface **WeatherService** (lines 13–14). Class **UnicastRemoteObject** provides the basic functionality required for all remote objects. In particular, its constructor *exports* the object to make it available to receive remote calls. Exporting the object enables the remote object to wait for client connections on an *anonymous port number* (i.e., one chosen by the computer on which the remote object executes). This enables the object to perform *unicast communication* (point-to-point communication between two objects via method calls) using standard streams-based socket connections. RMI abstracts away these communication details so the programmer can work with simple method calls. The **WeatherServiceImpl** constructor (lines 19–23) invokes the default constructor for class **UnicastRemoteObject** (line 21) and calls **private** method **updateWeatherConditions** (line 22). Overloaded constructors for class **UnicastRemoteObject** allow the programmer to specify additional information, such as an explicit port number on which to export the remote object. All **UnicastRemoteObject** constructors throw **RemoteException**s.

Software Engineering Observation 13.5

Class ***UnicastRemoteObject*** *constructors and methods throw checked* ***RemoteException****s, so* ***UnicastRemoteObject*** *subclasses must define constructors that also throw* ***RemoteException****s.*

Software Engineering Observation 13.6

Class ***UnicastRemoteObject*** *provides basic functionality that remote objects require to handle remote requests. Remote object classes need not extend this class if those remote object classes use* ***static*** *method* ***exportObject*** *of class* ***UnicastRemoteObject*** *to export remote objects.*

Method **updateWeatherConditions** (lines 26–91) reads weather information from the *Traveler's Forecast* Web page and stores this information in a **List** of **WeatherBean** objects. Lines 32–33 create a **URL** object for the *Traveler's Forecast* Web page. Lines 36–37 invoke method **openStream** of class **URL** to open a connection to the specified **URL** and wrap that connection with a **BufferedReader**.

Lines 40–87 perform *HTML scraping* (i.e., extracting data from a Web page) to retrieve the weather forecast information. Line 40 defines a separator **String**—**"TAV12"**—that determines the starting point from which to locate the appropriate weather information. Lines 43–44 read through the *Traveler's Forecast* Web page until reaching the sentinel. This process skips over information not needed for this application.

Lines 48–51 define two **String**s that represent the column heads for the weather information. Depending on the time of day, the column headers are either

```
"CITY                WEA     HI/LO   WEA     HI/LO"
```

after the morning update (normally around 10:30 AM Eastern Standard Time) or

```
"CITY                WEA     LO/HI   WEA     LO/HI"
```

after the evening update (normally around 10:30 PM Eastern Standard Time).

Lines 65–85 read each city's weather information and place this information in **WeatherBean** objects. Each **WeatherBean** contains the city's name, the temperature and a description of the weather. Line 61 creates a **List** for storing the **WeatherBean** objects. Lines 76–79 construct a **WeatherBean** object for the current city. The first 16 characters of **inputLine** are the city name, the next 6 characters of **inputLine** are the description (i.e., weather forecast) and the next 6 characters of **inputLine** are the high and low temperatures. The last two columns of data represent the next day's weather forecast, which we ignore in this example. Line 82 adds the **WeatherBean** object to the **List**. Line 87 closes the **BufferedReader** and its associated **InputStream**.

Method **getWeatherInformation** (lines 107–110) is the method from interface **WeatherService** that **WeatherServiceImpl** must implement to respond to remote requests. The method returns a serialized copy of the `weatherInformation List`. Clients invoke this remote method to obtain the weather information.

Method `main` (lines 113–127) creates the `WeatherServiceImpl` remote object. When the constructor executes, it exports the remote object so the object can listen for remote requests. Line 106 defines the URL that a client can use to obtain a *remote reference* to the server object. The client uses this remote reference to invoke methods on the remote object. The URL normally is of the form

`rmi://`*host*`:`*port*`/`*remoteObjectName*

where *host* represents the computer that is running the *registry for remote objects* (this also is the computer on which the remote object executes), *port* represents the port number on which the registry is running on the *host* and *remoteObjectName* is the name the client will supply when it attempts to locate the remote object in the registry. The ***`rmiregistry`*** utility program manages the registry for remote objects and is part of the J2SE SDK. The default port number for the RMI registry is `1099`.

Software Engineering Observation 13.7

RMI clients assume that they should connect to port `1099` when attempting to locate a remote object through the RMI registry (unless specified otherwise with an explicit port number in the URL for the remote object).

Software Engineering Observation 13.8

A client must specify a port number only if the RMI registry is running on a port other than the default port, `1099`.

In this program, the remote object URL is

```
rmi://localhost/WeatherService
```

indicating that the RMI registry is running on the `localhost` (i.e., the local computer) and that the name the client must use to locate the service is `WeatherService`. The name `localhost` is synonymous with the IP address `127.0.0.1`, so the preceding URL is equivalent to

```
rmi://127.0.0.1/WeatherService
```

Line 124 invokes `static` method ***`rebind`*** of class ***`Naming`*** (package `java.rmi`) to bind the remote `WeatherServiceImpl` object `service` to the RMI registry with the URL `rmi://localhost/WeatherService`. There also is a ***`bind`*** method for binding a remote object to the registry. Programmers use method `rebind` more commonly, because method `rebind` guarantees that if an object already has registered under the given name, the new remote object will replace the previously registered object. This could be important when registering a new version of an existing remote object.

Class `WeatherBean` (Fig. 13.3) stores data that class `WeatherServiceImpl` retrieves from the National Weather Service Web site. This class stores the city, temperature and weather descriptions as `String`s. Lines 64–85 provide *get* methods for each piece of information. Lines 25–45 load a property file that contains image names for displaying the weather information. This `static` block ensures that the image names are available as soon as the virtual machine loads the `WeatherBean` class into memory.

```java
// WeatherBean.java
// WeatherBean maintains weather information for one city.
package com.deitel.advjhtp1.rmi.weather;

// Java core packages
import java.awt.*;
import java.io.*;
import java.net.*;
import java.util.*;

// Java extension packages
import javax.swing.*;

public class WeatherBean implements Serializable {

   private String cityName;         // name of city
   private String temperature;      // city's temperature
   private String description;      // weather description
   private ImageIcon image;         // weather image

   private static Properties imageNames;

   // initialize imageNames when class WeatherBean
   // is loaded into memory
   static {
      imageNames = new Properties();  // create properties table

      // load weather descriptions and image names from
      // properties file
      try {

         // obtain URL for properties file
         URL url = WeatherBean.class.getResource(
            "imagenames.properties" );

         // load properties file contents
         imageNames.load( new FileInputStream( url.getFile() ) );
      }

      // process exceptions from opening file
      catch ( IOException ioException ) {
         ioException.printStackTrace();
      }

   } // end static block

   // WeatherBean constructor
   public WeatherBean( String city, String weatherDescription,
      String cityTemperature )
   {
      cityName = city;
      temperature = cityTemperature;
      description = weatherDescription.trim();
```

Fig. 13.3 **WeatherBean** stores weather forecast for one city (part 1 of 2).

```

      URL url = WeatherBean.class.getResource( "images/" +
        imageNames.getProperty( description, "noinfo.jpg" ) );

      // get weather image name or noinfo.jpg if weather
      // description not found
      image = new ImageIcon( url );
   }

   // get city name
   public String getCityName()
   {
      return cityName;
   }

   // get temperature
   public String getTemperature()
   {
      return temperature;
   }

   // get weather description
   public String getDescription()
   {
      return description;
   }

   // get weather image
   public ImageIcon getImage()
   {
      return image;
   }
}
```

Fig. 13.3 **WeatherBean** stores weather forecast for one city (part 2 of 2).

Next, we define the client application that will obtain weather information from the **WeatherServiceImpl**. Class **WeatherServiceClient** (Fig. 13.4) is the client application that invokes remote method **getWeatherInformation** of interface **WeatherService** to obtain weather information through RMI. Class **WeatherServiceClient** uses a **JList** with a custom **ListCellRenderer** to display the weather information for each city.

The **WeatherServiceClient** constructor (lines 16–58) takes as an argument the name of computer on which the **WeatherService** remote object is running. Line 24 creates a **String** that contains the URL for this remote object. Lines 27–28 invoke **Naming**'s **static** method ***lookup*** to obtain a remote reference to the **WeatherService** remote object at the specified URL. Method **lookup** connects to the RMI registry and returns a **Remote** reference to the remote object, so line 28 casts this reference to type **WeatherService**. Note that the **WeatherServiceClient** refers to the remote object only through interface **WeatherService**—the remote interface for the **WeatherServiceImpl** remote object implementation. The client can use this remote reference as if it referred to a local object running in the same virtual machine. This remote reference refers to a *stub* object on the client. Stubs allow clients to invoke remote objects' methods. Stub objects receive each remote method call and pass those calls to the RMI system, which performs the networking that allows clients to interact with the remote object. In this case, the **WeatherServiceImpl** stub will handle the communication between **WeatherServiceClient** and **WeatherServiceImpl**. The RMI layer is responsible for network connections to the remote object, so referencing remote objects is transparent to the client. RMI handles the underlying communication with the remote object and the transfer of arguments and return values between the objects.

Lines 31–32 invoke remote method **getWeatherInformation** on the **weatherService** remote reference. This method call returns a copy of the **List** of **WeatherBean**s, which contains information from the *Traveler's Forecast* Web page. It is important to note that RMI returns a copy of the **List**, because returning a reference from a remote method call is different from returning a reference from a local method call. RMI uses object serialization to send the **List** of **WeatherBean** objects to the client. Therefore, the argument and return types for remote methods must be **Serializable**.

```
1   // WeatherServiceClient.java
2   // WeatherServiceClient uses the WeatherService remote object
3   // to retrieve weather information.
4   package com.deitel.advjhtp1.rmi.weather;
5
6   // Java core packages
7   import java.rmi.*;
8   import java.util.*;
9
10  // Java extension packages
11  import javax.swing.*;
12
13  public class WeatherServiceClient extends JFrame
14  {
15     // WeatherServiceClient constructor
16     public WeatherServiceClient( String server )
17     {
18        super( "RMI WeatherService Client" );
19
20        // connect to server and get weather information
21        try {
22
23           // name of remote server object bound to rmi registry
24           String remoteName = "rmi://" + server + "/WeatherService";
25
26           // lookup WeatherServiceImpl remote object
27           WeatherService weatherService =
28              ( WeatherService ) Naming.lookup( remoteName );
29
30           // get weather information from server
31           List weatherInformation =
32              weatherService.getWeatherInformation();
33
```

Fig. 13.4 **WeatherServiceClient** client for **WeatherService** remote object (part 1 of 2).

```
34          // create WeatherListModel for weather information
35          ListModel weatherListModel =
36             new WeatherListModel( weatherInformation );
37 
38          // create JList, set ListCellRenderer and add to layout
39          JList weatherJList = new JList( weatherListModel );
40          weatherJList.setCellRenderer( new WeatherCellRenderer());
41          getContentPane().add( new JScrollPane( weatherJList ) );
42 
43       } // end try
44 
45       // handle exception connecting to remote server
46       catch ( ConnectException connectionException ) {
47          System.err.println( "Connection to server failed. " +
48             "Server may be temporarily unavailable." );
49 
50          connectionException.printStackTrace();
51       }
52 
53       // handle exceptions communicating with remote object
54       catch ( Exception exception ) {
55          exception.printStackTrace();
56       }
57 
58    } // end WeatherServiceClient constructor
59 
60    // execute WeatherServiceClient
61    public static void main( String args[] )
62    {
63       WeatherServiceClient client = null;
64 
65       // if no sever IP address or host name specified,
66       // use "localhost"; otherwise use specified host
67       if ( args.length == 0 )
68          client = new WeatherServiceClient( "localhost" );
69       else
70          client = new WeatherServiceClient( args[ 0 ] );
71 
72       // configure and display application window
73       client.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
74       client.pack();
75       client.setResizable( false );
76       client.setVisible( true );
77    }
78 }
```

Fig. 13.4 **WeatherServiceClient** client for **WeatherService** remote object (part 2 of 2).

Lines 35–36 create a **WeatherListModel** (Fig. 13.5) to facilitate displaying the weather information in a **JList** (line 39). Line 40 sets a **ListCellRenderer** for the **JList**. Class **WeatherCellRenderer** (Fig. 13.6) is a **ListCellRenderer** that uses **WeatherItem** objects to display weather information stored in **WeatherBean**s.

Method **main** (lines 61–77) checks the command-line arguments for a user-provided hostname. If the user did not provide a hostname, line 68 creates a new **WeatherServiceClient** that connects to an RMI registry running on **localhost**. If the user did provide a hostname, line 70 creates a **WeatherServiceClient** using the given hostname.

Class **WeatherListModel** (Fig. 13.5) is a **ListModel** that contains **WeatherBeans** to be displayed in a **JList**. This example continues our design patterns discussion by introducing the *Adapter design pattern*, which enables two objects with incompatible interfaces to communicate with each other.[1] The Adapter design pattern has many parallels in the real world. For example, the electrical plugs on appliances in the United States are not compatible with European electrical sockets. Using an American electrical appliance in Europe requires the user to place an adapter between the electrical plug and the electrical socket. On one side, this adapter provides an interface compatible with the American electrical plug. On the other side, this adapter provides an interface compatible with the European electrical socket. Class **WeatherListModel** plays the role of the *Adapter* in the Adapter design pattern. In Java, interface **List** is not compatible with class **JList**'s interface—a **JList** can retrieve elements only from a **ListModel**. Therefore, we provide class **WeatherListModel**, which adapts interface **List** to make it compatible with **JList**'s interface. When the **JList** invokes **WeatherListModel** method **getSize**, **WeatherListModel** invokes method **size** of interface **List**. When the **JList** invokes **WeatherListModel** method **getElementAt**, **WeatherListModel** invokes **JList** method **get**, etc. Class **WeatherListModel** also plays the role of the model in Swing's delegate-model architecture, as we discussed in Chapter 3, Model-View-Controller.

```java
// WeatherListModel.java
// WeatherListModel extends AbstractListModel to provide a
// ListModel for storing a List of WeatherBeans.
package com.deitel.advjhtp1.rmi.weather;

// Java core packages
import java.util.*;

// Java extension packages
import javax.swing.AbstractListModel;

public class WeatherListModel extends AbstractListModel {

   // List of elements in ListModel
   private List list;

   // no-argument WeatherListModel constructor
   public WeatherListModel()
   {
      // create new List for WeatherBeans
      list = new ArrayList();
   }
```

Fig. 13.5 **WeatherListModel** is a **ListModel** implementation for storing weather information (part 1 of 2).

[1] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns; Elements of Reusable Object-Oriented Software*. (Reading, MA: Addison-Wesley, 1995): p. 139.

```java

   // WeatherListModel constructor
   public WeatherListModel( List elementList )
   {
      list = elementList;
   }

   // get size of List
   public int getSize()
   {
      return list.size();
   }

   // get Object reference to element at given index
   public Object getElementAt( int index )
   {
      return list.get( index );
   }

   // add element to WeatherListModel
   public void add( Object element )
   {
      list.add( element );
      fireIntervalAdded( this, list.size(), list.size() );
   }

   // remove element from WeatherListModel
   public void remove( Object element )
   {
      int index = list.indexOf( element );

      if ( index != -1 ) {
         list.remove( element );
         fireIntervalRemoved( this, index, index );
      }

   } // end method remove

   // remove all elements from WeatherListModel
   public void clear()
   {
      // get original size of List
      int size = list.size();

      // clear all elements from List
      list.clear();

      // notify listeners that content changed
      fireContentsChanged( this, 0, size );
   }
}
```

Fig. 13.5 **`WeatherListModel`** is a **`ListModel`** implementation for storing weather information (part 2 of 2).

Class **JList** uses a **ListCellRenderer** to render each element in that **JList**'s **ListModel**. Class **WeatherCellRenderer** (Fig. 13.6) is a **DefaultListCellRenderer** subclass for rendering **WeatherBean**s in a **JList**. Method **getListCellRendererComponent** creates and returns a **WeatherItem** (Fig. 13.7) for the given **WeatherBean**.

Class **WeatherItem** (Fig. 13.7) is a **JPanel** subclass for displaying weather information stored in a **WeatherBean**. Class **WeatherCellRenderer** uses instances of class **WeatherItem** to display weather information in a **JList**. The **static** block (lines 22–29) loads the **ImageIcon backgroundImage** into memory when the virtual machine loads the **WeatherItem** class itself. This ensures that **backgroundImage** is available to all instances of class **WeatherItem**. Method **paintComponent** (lines 38–56) draws the **backgroundImage** (line 43), the city name (line 50), the temperature (line 51) and the **WeatherBean**'s **ImageIcon**, which describes the weather conditions (line 54).

```java
// WeatherCellRenderer.java
// WeatherCellRenderer is a custom ListCellRenderer for
// WeatherBeans in a JList.
package com.deitel.advjhtp1.rmi.weather;

// Java core packages
import java.awt.*;

// Java extension packages
import javax.swing.*;

public class WeatherCellRenderer extends DefaultListCellRenderer {

   // returns a WeatherItem object that displays city's weather
   public Component getListCellRendererComponent( JList list,
      Object value, int index, boolean isSelected, boolean focus )
   {
      return new WeatherItem( ( WeatherBean ) value );
   }
}
```

Fig. 13.6 **WeatherCellRenderer** is a custom **ListCellRenderer** for displaying **WeatherBean**s in a **JList**.

```java
 1  // WeatherItem.java
 2  // WeatherItem displays a city's weather information in a JPanel.
 3  package com.deitel.advjhtp1.rmi.weather;
 4
 5  // Java core packages
 6  import java.awt.*;
 7  import java.net.*;
 8  import java.util.*;
 9
10  // Java extension packages
11  import javax.swing.*;
12
```

Fig. 13.7 **WeatherItem** displays weather information for one city (part 1 of 2).

```java
public class WeatherItem extends JPanel {

   private WeatherBean weatherBean;  // weather information

   // background ImageIcon
   private static ImageIcon backgroundImage;

   // static initializer block loads background image when class
   // WeatherItem is loaded into memory
   static {

      // get URL for background image
      URL url = WeatherItem.class.getResource( "images/back.jpg" );

      // background image for each city's weather info
      backgroundImage = new ImageIcon( url );
   }

   // initialize a WeatherItem
   public WeatherItem( WeatherBean bean )
   {
      weatherBean = bean;
   }

   // display information for city's weather
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g );

      // draw background
      backgroundImage.paintIcon( this, g, 0, 0 );

      // set font and drawing color,
      // then display city name and temperature
      Font font = new Font( "SansSerif", Font.BOLD, 12 );
      g.setFont( font );
      g.setColor( Color.white );
      g.drawString( weatherBean.getCityName(), 10, 19 );
      g.drawString( weatherBean.getTemperature(), 130, 19 );

      // display weather image
      weatherBean.getImage().paintIcon( this, g, 253, 1 );

   } // end method paintComponent

   // make WeatherItem's preferred size the width and height of
   // the background image
   public Dimension getPreferredSize()
   {
      return new Dimension( backgroundImage.getIconWidth(),
         backgroundImage.getIconHeight() );
   }
}
```

Fig. 13.7 **`WeatherItem`** displays weather information for one city (part 2 of 2).

The images in this example are available with the example code from this text on the CD that accompanies the text and from our Web site (**www.deitel.com**). Click the **Downloads** link and download the examples for *Advanced Java 2 Platform How to Program.*

13.5 Compiling and Executing the Server and the Client

Now that the pieces are in place, we can build and execute our distributed application; this requires several steps. First, we must compile the classes. Next, we must compile the remote object class (**WeatherServiceImpl**), using the ***rmic** compiler* (a utility supplied with the J2SE SDK) to produce a *stub class*. As we discussed in Section 13.4, a stub class forwards method invocations to the RMI layer, which performs the network communication necessary to invoke the method call on the remote object. The command line

```
rmic -v1.2 com.deitel.advjhtp1.rmi.weather.WeatherServiceImpl
```

generates the file **WeatherServiceImpl_Stub.class**. This class must be available to the client (either locally or via download) to enable remote communication with the server object. Depending on the command line options passed to **rmic**, this may generate several files. In Java 1.1, **rmic** produced two classes—a stub class and a *skeleton class*. Java 2 no longer requires the skeleton class. The command-line option **-v1.2** indicates that **rmic** should create only the stub class.

The next step is to start the RMI registry with which the **WeatherServiceImpl** object will register. The command line

```
rmiregistry
```

launches the RMI registry on the local machine. The command line window (Fig. 13.8) will not show any text in response to this command.

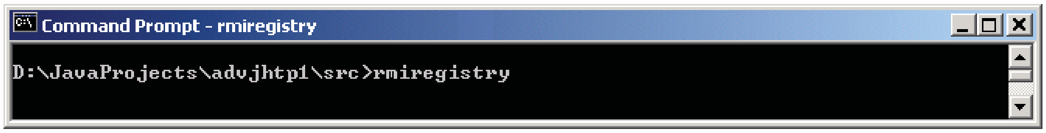
Common Programming Error 13.1

*Not starting the RMI registry before attempting to bind the remote object to the registry results in a **java.rmi.ConnectException**, which indicates that the program cannot connect to the registry.*

To make the remote object available to receive remote method calls, we bind the object to a name in the RMI registry. Run the **WeatherServiceImpl** application from the command line as follows:

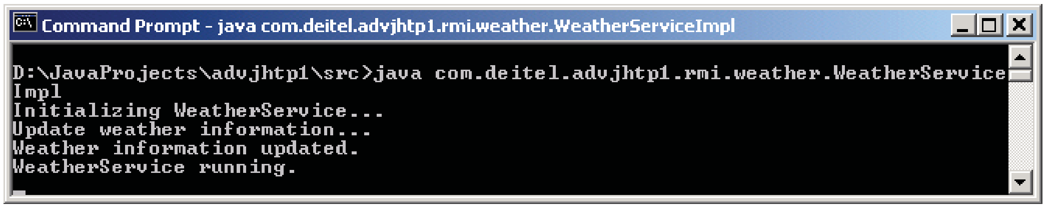
```
java com.deitel.advjhtp1.rmi.weather.WeatherServiceImpl
```

Figure 13.9 shows the **WeatherServiceImpl** application output. Class **WeatherServiceImpl** retrieves the data from the *Traveler's Forecast* Web page and displays a message indicating that the service is running.



```
D:\JavaProjects\advjhtp1\src>rmiregistry
```

Fig. 13.8 Running the **rmiregistry**.


Command Prompt - java com.deitel.advjhtp1.rmi.weather.WeatherServiceImpl

```
D:\JavaProjects\advjhtp1\src>java com.deitel.advjhtp1.rmi.weather.WeatherService
Impl
Initializing WeatherService...
Update weather information...
Weather information updated.
WeatherService running.
```

Fig. 13.9 Executing the **WeatherServiceImpl** remote object.

The **WeatherServiceClient** program now can connect with the **WeatherServiceImpl** running on **localhost** with the command

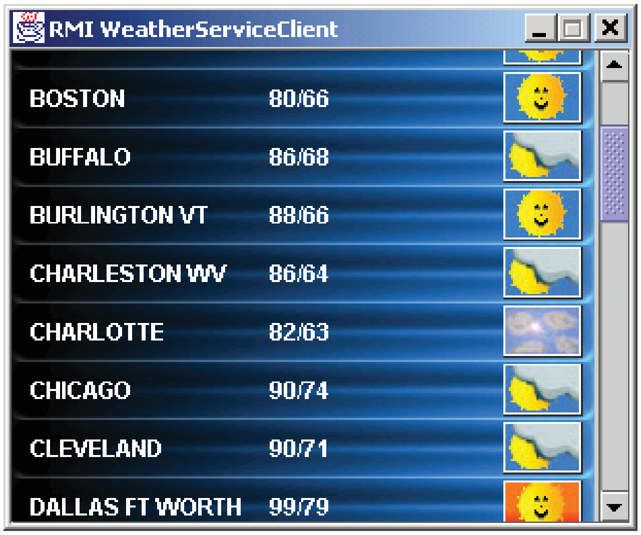
```
java com.deitel.advjhtp1.rmi.weather.WeatherServiceClient
```

Figure 13.10 shows the **WeatherServiceClient** application window. When the program executes, the **WeatherServiceClient** connects to the remote server object and displays the current weather information.

If the **WeatherServiceImpl** is running on a different machine from the client, you can specify the IP address or host name of the server computer as a command-line argument when executing the client. For example, to access a computer with IP address **192.168.0.150**, enter the command

```
java com.deitel.advjhtp1.rmi.weather.WeatherServiceClient
192.168.0.150
```

In the first part of this chapter, we built a simple distributed system that demonstrated the basics of RMI. In the following case study, we build a more sophisticated RMI distributed system that takes advantage of some advanced RMI features.



BOSTON	80/66
BUFFALO	86/68
BURLINGTON VT	88/66
CHARLESTON WV	86/64
CHARLOTTE	82/63
CHICAGO	90/74
CLEVELAND	90/71
DALLAS FT WORTH	99/79

Fig. 13.10 **WeatherServiceClient** application window.

13.6 Case Study: Deitel Messenger with **Activatable** Server

In this section, we present a case study that implements an online chat system using RMI and an *activatable* chat server. This case study—the Deitel Messenger—uses several advanced RMI features and a modular architecture that promotes reusability. Figure 13.11 lists the classes and interfaces that make up the case study and brief descriptions of each. Interfaces are shown in italic font.

Standard RMI objects exported as **UnicastRemoteObject**s must run continuously on the server to handle client requests. RMI objects that extend class **java.rmi.activation.Activatable** are able to *activate*, or start running, when a client invokes one of the remote object's methods. This can conserve resources on the server because a remote object's processes are put to sleep and release memory when there are no clients using that particular remote object. The *RMI activation daemon* (***rmid***) is a server process that enables activatable remote objects to become active when clients invoke remote methods on these objects.

Name	Role
ChatServer	Remote interface through which clients register for a chat, leave a chat, and post chat messages.
StoppableChatServer	Administrative remote interface for terminating the chat server.
ChatServerImpl	Implementation of the ChatServer remote interface that provides an RMI-based chat server.
ChatServerAdministrator	Utility program for launching and terminating the activatable **ChatServer**.
ChatClient	Remote interface through which the ChatServer communicates with clients.
ChatMessage	**Serializable** object for sending messages between **ChatServer** and **ChatClient**s.
MessageManager	Interface that defines methods for managing communication between the client's user interface and the **ChatServer**.
RMIMessageManager	**ChatClient** and **MessageManager** implementation for managing communication between the client and the **ChatServer**.
MessageListener	Interface for classes that wish to receive new chat messages.
DisconnectListener	Interface for classes that wish to receive notifications when the server disconnects.
ClientGUI	GUI for sending and receiving chat messages using a **MessageManager**.
DeitelMessenger	Application launcher for Deitel Messenger client.

Fig. 13.11 Participants of DeitelMessenger case study.

Activatable remote objects also are able to recover from server crashes, because remote references to activatable objects are persistent—when the server restarts, the RMI activation daemon maintains the remote reference, so clients can continue to use the remote object. We discuss the details of implementing **Activatable** remote objects when we present the chat server implementation.

13.6.1 Activatable Deitel Messenger ChatServer

Like every RMI remote object, an **Activatable** remote object must implement a remote interface. Interface **ChatServer** (Fig. 13.12) is the remote interface for the Deitel Messenger server. Clients interact with the Deitel Messenger server through the **ChatServer** remote interface. Remote interfaces for an **Activatable** RMI have the same requirements as standard RMI remote interfaces.

Line 13 declares that interface **ChatServer** extends interface **Remote**, which RMI requires for all remote interfaces. Method **registerClient** (lines 16–17) enables a **ChatClient** (Fig. 13.17) to register with the **ChatServer** and take part in the chat session. Method **registerClient** takes as an argument the **ChatClient** to register. Interface **ChatClient** is itself a remote interface, so both the server and client are remote objects in this application. This enables the server to communicate with clients by invoking remote methods on those clients. We discuss this communication—called an RMI callback—in more detail when we present the **ChatClient** implementation.

```
 1  // ChatServer.java
 2  // ChatServer is a remote interface that defines how a client
 3  // registers for a chat, leaves a chat and posts chat messages.
 4  package com.deitel.messenger.rmi.server;
 5
 6  // Java core packages
 7  import java.rmi.*;
 8
 9  // Deitel packages
10  import com.deitel.messenger.rmi.ChatMessage;
11  import com.deitel.messenger.rmi.client.ChatClient;
12
13  public interface ChatServer extends Remote {
14
15     // register new ChatClient with ChatServer
16     public void registerClient( ChatClient client )
17        throws RemoteException;
18
19     // unregister ChatClient with ChatServer
20     public void unregisterClient( ChatClient client )
21        throws RemoteException;
22
23     // post new message to ChatServer
24     public void postMessage( ChatMessage message )
25        throws RemoteException;
26  }
```

Fig. 13.12 **ChatServer** remote interface for Deitel Messenger chat server.

Method **unregisterClient** (lines 20–21) enables clients to remove themselves from the chat session. Method **postMessage** enables clients to post new messages to the chat session. Method **postMessage** takes as an argument a reference to a **ChatMessage**. A **ChatMessage** (Fig. 13.18) is a **Serializable** object that contains the name of the sender and the message body. We discuss this class in more detail shortly.

The server side of the Deitel Messenger application includes a program for managing the **Activatable** remote object. Interface **StoppableChatServer** (Fig. 13.13) declares method **stopServer**. The program that manages the Deitel Messenger server invokes method **stopServer** to terminate the server.

Class **ChatServerImpl** (Fig. 13.14) is an **Activatable** RMI object that implements the **ChatServer** and **StoppableChatServer** remote interfaces. Line 23 creates a **Set** for maintaining remote references to registered **ChatClient**s. The **ChatServerImpl** constructor (lines 29–34) takes as arguments an **ActivationID** and a **MarshalledObject**. The RMI activation mechanism requires that **Activatable** objects provide this constructor. When the activation daemon activates a remote object of this class, it invokes this *activation constructor*. The **ActivationID** argument specifies a unique identifier for the remote object. Class **MarshalledObject** is a wrapper class that contains a serialized object for transmission over RMI. In this case, the **MarshalledObject** argument contains application-specific initialization information, such as the name under which the activation daemon registered the remote object. Line 33 invokes the superclass constructor to complete activation. The second argument to the superclass constructor (**0**) specifies that the activation daemon should export the object on an anonymous port.

```java
// StoppableChatServer.java
// StoppableChatServer is a remote interface that provides a
// mechansim to terminate the chat server.
package com.deitel.messenger.rmi.server;

// Java core packages
import java.rmi.*;

public interface StoppableChatServer extends Remote {

   // stop ChatServer
   public void stopServer() throws RemoteException;
}
```

Fig. 13.13 **StoppableChatServer** remote interface for stopping a **ChatServer** remote object.

```java
1  // ChatServerImpl.java
2  // ChatServerImpl implements the ChatServer remote interface
3  // to provide an RMI-based chat server.
4  package com.deitel.messenger.rmi.server;
5
```

Fig. 13.14 **ChatServerImpl** implementation of remote interfaces **ChatServer** and **StoppableChatServer** as **Activatable** remote objects (part 1 of 5).

```java
// Java core packages
import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.activation.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.util.*;

// Deitel packages
import com.deitel.messenger.rmi.ChatMessage;
import com.deitel.messenger.rmi.client.ChatClient;

public class ChatServerImpl extends Activatable
   implements ChatServer, StoppableChatServer {

   // Set of ChatClient references
   private Set clients = new HashSet();

   // server object's name
   private String serverObjectName;

   // ChatServerImpl constructor
   public ChatServerImpl( ActivationID id, MarshalledObject data )
      throws RemoteException {

       // register activatable object and export on anonymous port
       super( id, 0 );
   }

   // register ChatServerImpl object with RMI registry.
   public void register( String rmiName ) throws RemoteException,
      IllegalArgumentException, MalformedURLException
   {
      // ensure registration name was provided
      if ( rmiName == null )
         throw new IllegalArgumentException(
            "Registration name cannot be null" );

      serverObjectName = rmiName;

      // bind ChatServerImpl object to RMI registry
      try {

         // create RMI registry
         System.out.println( "Creating registry ..." );
         Registry registry  =
            LocateRegistry.createRegistry( 1099 );

         // bind RMI object to default RMI registry
         System.out.println( "Binding server to registry ..." );
```

Fig. 13.14 **ChatServerImpl** implementation of remote interfaces **ChatServer** and **StoppableChatServer** as **Activatable** remote objects (part 2 of 5).

```
         registry.rebind( serverObjectName, this );
      }

      // if registry already exists, bind to existing registry
      catch ( RemoteException remoteException ) {
         System.err.println( "Registry already exists. " +
            "Binding to existing registry ..." );
         Naming.rebind( serverObjectName, this );
      }

      System.out.println( "Server bound to registry" );

   } // end method register

   // register new ChatClient with ChatServer
   public void registerClient( ChatClient client )
      throws RemoteException
   {
      // add client to Set of registered clients
      synchronized ( clients ) {
         clients.add( client );
      }

      System.out.println( "Registered Client: " + client );

   } // end method registerClient

   // unregister client with ChatServer
   public void unregisterClient( ChatClient client )
      throws RemoteException
   {
      // remove client from Set of registered clients
      synchronized( clients ) {
         clients.remove( client );
      }

      System.out.println( "Unregistered Client: " + client );

   } // end method unregisterClient

   // post new message to chat server
   public void postMessage( ChatMessage message )
      throws RemoteException
   {
      Iterator iterator = null;

      // get Iterator for Set of registered clients
      synchronized( clients ) {
         iterator = new HashSet( clients ).iterator();
      }

```

Fig. 13.14 **ChatServerImpl** implementation of remote interfaces **ChatServer** and **StoppableChatServer** as **Activatable** remote objects (part 3 of 5).

```
      // send message to every ChatClient
      while ( iterator.hasNext() ) {

         // attempt to send message to client
         ChatClient client = ( ChatClient ) iterator.next();

         try {
            client.deliverMessage( message );
         }

         // unregister client if exception is thrown
         catch( Exception exception ) {
            System.err.println( "Unregistering absent client." );
            unregisterClient( client );
         }

      } // end while loop

   } // end method postMessage

   // notify each client that server is shutting down and
   // terminate server application
   public void stopServer() throws RemoteException
   {
      System.out.println( "Terminating server ..." );

      Iterator iterator = null;

      // get Iterator for Set of registered clients
      synchronized( clients ) {
         iterator = new HashSet( clients ).iterator();
      }

      // send message to every ChatClient
      while ( iterator.hasNext() ) {
         ChatClient client = ( ChatClient ) iterator.next();
         client.serverStopping();
      }

      // create Thread to terminate application after
      // stopServer method returns to caller
      Thread terminator = new Thread(
         new Runnable() {

            // sleep for 5 seconds, print message and terminate
            public void run()
            {
               // sleep
               try {
                  Thread.sleep( 5000 );
               }
```

Fig. 13.14 **ChatServerImpl** implementation of remote interfaces **ChatServer** and **StoppableChatServer** as **Activatable** remote objects (part 4 of 5).

```
159
160                     // ignore InterruptedExceptions
161                     catch ( InterruptedException exception ) {
162                     }
163
164                     System.err.println( "Server terminated" );
165                     System.exit( 0 );
166                 }
167             }
168         );
169
170         terminator.start(); // start termination thread
171
172     } // end method stopServer
173 }
```

Fig. 13.14 **ChatServerImpl** implementation of remote interfaces **ChatServer** and **StoppableChatServer** as **Activatable** remote objects (part 5 of 5).

Method **register** (lines 37–69) registers a **ChatServerImpl** remote object with the RMI registry. If the provided name for the remote object is **null**, lines 42–43 throw an **IllegalArgumentException**, indicating that the caller must specify a name for the remote object. Lines 52–53 use **static** method **createRegistry** of class **LocateRegistry** to create a new **Registry** on the local machine at port **1099**, which is the default port. This is equivalent to executing the **rmiregistry** utility to start a new RMI registry. Line 57 invokes method **rebind** of class **Registry** to bind the activatable object to the **Registry**. If creating or binding to the **Registry** fails, we assume that an RMI registry already is running on the local machine. Line 64 invokes **static** method **rebind** of class **Naming** to bind the remote object to the existing RMI registry.

Method **registerClient** (lines 72–82) enables **ChatClient** remote objects to register with the **ChatServer** to participate in the chat session. The **ChatClient** argument to method **registerClient** is a remote reference to the registering client, which is itself a remote object. Line 77 adds the **ChatClient** remote reference to the **Set** of **ChatClient**s participating in the chat session. Method **unregisterClient** (lines 85–95) enables **ChatClient**s to leave the chat session. Line 90 removes the given **ChatClient** remote reference from the **Set** of **ChatClient** references.

ChatClients invoke method **postMessage** (lines 98–124) to post new **ChatMessage**s to the chat session. Each **ChatMessage** (Fig. 13.18) instance is a **Serializable** object that contains as properties the message sender and the message body. Lines 109–123 iterate through the **Set** of **ChatClient** references and invoke remote method **deliverMessage** of interface **ChatClient** to deliver the new **ChatMessage** to each client. If delivering a message to a client throws an exception, we assume that the client is no longer available. Line 121 therefore unregisters the absent client from the server.

Interface **StoppableChatServer** requires that class **ChatServerImpl** implements method **stopServer** (lines 128–170). Lines 140–143 iterate through the **Set** of **ChatClient** references and invoke method **serverStopping** of interface **ChatClient** to notify each **ChatClient** that the server is shutting down. Lines 147–168 create and start a new **Thread** to ensure that the **ChatServerAdministrator** (Fig. 13.15) can unbind the remote object from the RMI **Registry** before the remote object terminates.

Class **ChatServerAdministrator** (Fig. 13.15) is a utility program for registering and unregistering the activatable **ChatServer** remote object. Method **startServer** (lines 14–52) launches the activatable **ChatServer**. Activatable RMI objects execute as part of an ***ActivationGroup*** (package **java.rmi.activation**). The RMI activation daemon starts a new virtual machine for each **ActivationGroup**. Lines 21–22 create a **Properties** object and add a property that specifies the policy file under which the **ActivationGroup**'s JVM should run. This policy file (Fig. 13.16) allows **Activatable** objects in this **ActivationGroup** to terminate the virtual machine for this activation group. Recall that **ChatServerImpl** invokes **static** method **exit** of class **System** in method **stopServer**, which terminates the **ActivationGroup**'s virtual machine along with all of its executing remote objects.

```
 1  // ChatServerAdministrator.java
 2  // ChatServerAdministrator is a utility program for launching
 3  // and terminating the Activatable ChatServer.
 4  package com.deitel.messenger.rmi.server;
 5
 6  // Java core packages
 7  import java.rmi.*;
 8  import java.rmi.activation.*;
 9  import java.util.*;
10
11  public class ChatServerAdministrator {
12
13     // set up activatable server object
14     private static void startServer( String policy,
15        String codebase ) throws Exception
16     {
17        // set up RMI security manager
18        System.setSecurityManager( new RMISecurityManager() );
19
20        // set security policy for ActivatableGroup JVM
21        Properties properties = new Properties();
22        properties.put( "java.security.policy", policy );
23
24        // create ActivationGroupDesc for activatable object
25        ActivationGroupDesc groupDesc =
26           new ActivationGroupDesc( properties, null );
27
28        // register activation group with RMI activation system
29        ActivationGroupID groupID =
30           ActivationGroup.getSystem().registerGroup( groupDesc );
31
32        // create activation group
33        ActivationGroup.createGroup( groupID, groupDesc , 0 );
34
```

Fig. 13.15 **ChatServerAdministrator** application for starting and stopping the **ChatServer** remote object (part 1 of 3).

```
      // activation description for ChatServerImpl
      ActivationDesc description = new ActivationDesc(
         "com.deitel.messenger.rmi.server.ChatServerImpl",
         codebase, null );

      // register description with rmid
      ChatServer server =
         ( ChatServer ) Activatable.register( description );
      System.out.println( "Obtained ChatServerImpl stub" );

      // bind ChatServer in registry
      Naming.rebind( "ChatServer", server );
      System.out.println( "Bound object to registry" );

      // terminate setup program
      System.exit( 0 );

   } // end method startServer

   // terminate server
   private static void terminateServer( String hostname )
      throws Exception
   {
      // lookup ChatServer in RMI registry
      System.out.println( "Locating server ..." );
      StoppableChatServer server = ( StoppableChatServer )
         Naming.lookup( "rmi://" + hostname + "/ChatServer" );

      // terminate server
      System.out.println( "Stopping server ..." );
      server.stopServer();

      // remove ChatServer from RMI registry
      System.out.println( "Server stopped" );
      Naming.unbind( "rmi://" + hostname + "/ChatServer" );

   } // end method terminateServer

   // launch ChatServerAdministrator application
   public static void main( String args[] ) throws Exception
   {
      // check for stop server argument
      if ( args.length == 2 ) {

         if ( args[ 0 ].equals( "stop" ) )
            terminateServer( args[ 1 ] );

         else printUsageInstructions();
      }

```

Fig. 13.15 **ChatServerAdministrator** application for starting and stopping the **ChatServer** remote object (part 2 of 3).

```
85       // check for start server argument
86       else if ( args.length == 3 ) {
87
88          // start server with user-provided policy, codebase
89          // and Registry hostname
90          if ( args[ 0 ].equals( "start" ) )
91             startServer( args[ 1 ], args[ 2 ] );
92
93          else printUsageInstructions();
94       }
95
96       // wrong number of arguments provided, so print instructions
97       else printUsageInstructions();
98
99    } // end method main
100
101   // print instructions for running ChatServerAdministrator
102   private static void printUsageInstructions()
103   {
104      System.err.println( "\nUsage:\n" +
105         "\tjava com.deitel.messenger.rmi.server." +
106         "ChatServerAdministrator start <policy> <codebase>\n" +
107         "\tjava com.deitel.messenger.rmi.server." +
108         "ChatServerAdministrator stop <registry hostname>" );
109   }
110 }
```

Fig. 13.15 **ChatServerAdministrator** application for starting and stopping the **ChatServer** remote object (part 3 of 3).

```
// allow ActivationGroup to terminate the virtual machine
grant {
   permission java.lang.RuntimePermission "exitVM";
};
```

Fig. 13.16 Policy file for **ChatServer**'s **ActivationGroup**.

Lines 25–26 create an **ActivationGroupDesc** object, which is an *activation group descriptor*. The activation group descriptor specifies configuration information for the **ActivationGroup**. The first argument to the **ActivationGroupDesc** constructor is a **Properties** reference that contains replacement values for system properties in the **ActivationGroup**'s virtual machine. In this example, we override the **java.security.policy** system property to provide an appropriate security policy for the **ActivationGroup**'s virtual machine. The second argument is a reference to an **ActivationGroupDesc.CommandEnvironment** object. This object enables the **ActivationGroup** to customize the commands that the activation daemon executes when starting the **ActivationGroup**'s virtual machine. This example requires no such customization, so we pass a **null** reference for the second argument.

Lines 29–30 obtain an **ActivationSystem** by invoking **static** method **getSystem** of class **ActivationGroup**. Line 30 invokes method **registerGroup** of interface **ActivationSystem** and passes as an argument the **groupDesc** activation group descriptor. Method **registerGroup** returns the **ActivationGroupID** for the newly registered **ActivationGroup**. Line 33 invokes **static** method **createGroup** of class **ActivationGroup** to create the **ActivationGroup**. This method takes as arguments the **ActivationGroupID**, the **ActivationGroupDesc** and the *incarnation number* for the **ActivationGroup**. The incarnation number identifies different instances of the same **ActivationGroup**. Each time the activation daemon activates the **ActivationGroup**, the daemon increments the incarnation number.

Lines 36–38 create an **ActivationDesc** object for the **ChatServer** remote object. This *activation descriptor* specifies configuration information for a particular **Activatable** remote object. The first argument to the **ActivationDesc** constructor specifies the name of the class that implements the **Activatable** remote object. The second argument specifies the codebase that contains the remote object's class files. The final argument is a **MarshalledObject** reference, whose object specifies initialization information for the remote object. Recall that the **ChatServerImpl** activation constructor takes as its second argument a **MarshalledObject** reference. Our **ChatServer** remote object requires no special initialization information, so line 38 passes a **null** reference for the **MarshalledObject** argument.

Line 42 invokes **static** method **register** of class **Activatable** to register the **Activatable** remote object. Method **register** takes as an argument the **ActivationDesc** for the **Activatable** object and returns a reference to the remote object's stub. Line 46 invokes **static** method **rebind** of class **Naming** to bind the **ChatServer** in the RMI **Registry**.

Method **terminateServer** (lines 55–71) provides a means to shut down the activatable **ChatServer** remote object. Line 61 invokes **static** method **lookup** of class **Naming** to obtain a remote reference to the **ChatServer**. Line 60 casts the reference to type **StoppableChatServer**, which declares method **stopServer**. Line 65 invokes method **stopServer** to notify clients that the **ChatServer** is shutting down. Recall that method **stopServer** of class **ChatServerImpl** starts a **Thread** that waits five seconds before invoking **static** method **exit** of class **System**. This **Thread** keeps the **ChatServer** remote object running after method **stopServer** returns, allowing the **ChatServerAdministrator** to remove the remote object from the RMI **Registry**. Line 69 invokes **static** method **unbind** of class **Naming** to remove the **ChatServer** remote object from the RMI **Registry**. The **Thread** in class **ChatServerImpl** then terminates the virtual machine in which the **ChatServer** remote object ran.

Method **main** (lines 74–99) checks the command-line arguments to determine whether to start or stop the **ChatServer** remote object. When stopping the server, the user must provide as the second argument the hostname of the computer on which the server is running. When starting the server, the user must provide as arguments the location of the policy file for the **ActivationGroup** and the codebase for the remote object. If the user passes argument **"stop"**, line 80 invokes method **terminateServer** to shut down the **ChatServer** on the specified host. If the user passes argument **"start"**, line 91 invokes method **startServer** with the given policy file location and codebase. If the user provides an invalid number or type of arguments, lines 82, 93 and 97 invoke method **printUsageInstructions** (lines 102–109) to display information about the required command-line arguments.

13.6.2 Deitel Messenger Client Architecture and Implementation

Throughout this book, we present several versions of the Deitel Messenger case study. Each version implements the underlying communications using a different technology. For example, in Chapter 26, Common Object Request Broker Architecture (CORBA): Part 1, we present an implementation that uses CORBA as the underlying communication mechanism. The client for the Deitel Messenger application uses a modularized architecture to optimize code reuse in the several versions of this case study.

Communication Interfaces and Implementation

The client for the Deitel Messenger system separates the application GUI and the network communication into separate objects that interact through a set of interfaces. This enables us to use the same client-side GUI for different versions of the Deitel Messenger application. In this section, we present these interfaces and implementations with RMI.

Interface **ChatClient** (Fig. 13.17) is an RMI remote interface that enables the **ChatServer** to communicate with the **ChatClient** through *RMI callbacks*—remote method calls from the **ChatServer** back to the client. Recall that when a client connects to the **ChatServer**, the client invokes **ChatServer** method **registerClient** and passes as an argument a **ChatClient** remote reference. The server then uses this **ChatClient** remote reference to invoke RMI callbacks on the **ChatClient** (e.g., to deliver **ChatMessage**s to that client). Method **deliverMessage** (lines 16–17) enables the **ChatServer** to send new **ChatMessages** to the **ChatClient**. Method **serverStopping** (line 20) enables the **ChatServer** to notify the **ChatClient** when the **ChatServer** is shutting down.

```
 1  // ChatClient.java
 2  // ChatClient is a remote interface that defines methods for a
 3  // chat client to receive messages and status information from
 4  // a ChatServer.
 5  package com.deitel.messenger.rmi.client;
 6
 7  // Java core packages
 8  import java.rmi.*;
 9
10  // Deitel packages
11  import com.deitel.messenger.rmi.ChatMessage;
12
13  public interface ChatClient extends Remote {
14
15     // method called by server to deliver message to client
16     public void deliverMessage( ChatMessage message )
17        throws RemoteException;
18
19     // method called when server shuting down
20     public void serverStopping() throws RemoteException;
21  }
```

Fig. 13.17 **ChatClient** remote interface to enable RMI callbacks.

Class **ChatMessage** (Fig. 13.18) is a **Serializable** class that represents a message in the Deitel Messenger system. Instance variables **sender** and **message** contain the name of the person who sent the message and the message body, respectively. Class **ChatMessage** provides *set* and *get* methods for the **sender** and **message** and method **toString** for producing a **String** representation of a **ChatMessage**.

Interface **MessageManager** (Fig. 13.19) declares methods for classes that implement communication logic for a **ChatClient**. The methods that this interface declares are not specific to any underlying communication implementation. The chat client GUI uses a **MessageManager** implementation to connect to and disconnect from the **ChatServer**, and to send messages. Method **connect** (lines 10–11) connects to the **ChatServer** and takes as an argument the **MessageListener** to which the **MessageManager** should deliver new incoming messages. We discuss interface **MessageListener** in detail when we present the client user interface. Method **disconnect** (lines 15–16) disconnects the **MessageManager** from the **ChatServer** and stops routing messages to the given **MessageListener**. Method **sendMessage** (lines 19–20) takes as **String** arguments a user name (**from**) and a **message** to send to the **ChatServer**. Method **setDisconnectListener** registers a **DisconnectListener** to be notified when the **ChatServer** disconnects the client. We discuss interface **DisconnectListener** in detail when we present the client user interface.

```java
// ChatMessage.java
// ChatMessage is a Serializable object for messages in the RMI
// ChatClient and ChatServer.
package com.deitel.messenger.rmi;

// Java core packages
import java.io.*;

public class ChatMessage implements Serializable {

   private String sender;   // person sending message
   private String message;  // message being sent

   // construct empty ChatMessage
   public ChatMessage()
   {
      this( "", "" );
   }

   // construct ChatMessage with sender and message values
   public ChatMessage( String sender, String message )
   {
      setSender( sender );
      setMessage( message );
   }

```

Fig. 13.18 **ChatMessage** is a serializable class for transmitting messages over RMI (part 1 of 2).

```
   // set name of person sending message
   public void setSender( String name )
   {
      sender = name;
   }

   // get name of person sending message
   public String getSender()
   {
      return sender;
   }

   // set message being sent
   public void setMessage( String messageBody )
   {
      message = messageBody;
   }

   // get message being sent
   public String getMessage()
   {
      return message;
   }

   // String representation of ChatMessage
   public String toString()
   {
      return getSender() + "> " + getMessage();
   }
}
```

Fig. 13.18 **ChatMessage** is a serializable class for transmitting messages over RMI (part 2 of 2).

```
// MessageManager.java
// MessageManger is an interface for objects capable of managing
// communications with a message server.
package com.deitel.messenger;

public interface MessageManager {

   // connect to message server and route incoming messages
   // to given MessageListener
   public void connect( MessageListener listener )
      throws Exception;

   // disconnect from message server and stop routing
   // incoming messages to given MessageListener
   public void disconnect( MessageListener listener )
      throws Exception;

```

Fig. 13.19 **MessageManager** interface for classes that implement communication logic for a **ChatClient** (part 1 of 2).

```
   // send message to message server
   public void sendMessage( String from, String message )
      throws Exception;

   // set listener for disconnect notifications
   public void setDisconnectListener(
      DisconnectListener listener );
}
```

Fig. 13.19 **MessageManager** interface for classes that implement communication logic for a **ChatClient** (part 2 of 2).

Class **RMIMessageManager** (Fig. 13.20) handles all communication between the client and the **ChatServer**. Class **RMIMessageManager** is an RMI remote object that extends class **UnicastRemoteObject** and implements the **ChatClient** remote interface (lines 18–19). Class **RMIMessageManager** also implements interface **MessageManager**, enabling the client user interface to use an **RMIMessageManager** object to communicate with the **ChatServer**.

The **RMIMessageManager** constructor takes as a **String** argument the hostname of the computer running the RMI registry with which the **ChatServer** has registered. Note that because class **RMIMessenger** is itself an RMI remote object, the **RMIMessageManager** constructor throws **RemoteException**, which RMI requires of all **UnicastRemoteObject** subclasses. Line 31 assigns the given server name to instance variable **serverAddress**.

```
1  // RMIMessageManager.java
2  // RMIMessageManager implements the ChatClient remote interface
3  // and manages incoming and outgoing chat messages using RMI.
4  package com.deitel.messenger.rmi.client;
5
6  // Java core packages
7  import java.awt.*;
8  import java.awt.event.*;
9  import java.rmi.*;
10 import java.rmi.server.*;
11 import java.util.*;
12
13 // Deitel packages
14 import com.deitel.messenger.*;
15 import com.deitel.messenger.rmi.*;
16 import com.deitel.messenger.rmi.server.ChatServer;
17
18 public class RMIMessageManager extends UnicastRemoteObject
19    implements ChatClient, MessageManager {
20
21    // listeners for incoming messages and disconnect notifications
22    private MessageListener messageListener;
23    private DisconnectListener disconnectListener;
24
```

Fig. 13.20 **RMIMessageManager** remote object and **MessageManager** implementation for managing **ChatClient** communication (part 1 of 3).

```java
   private String serverAddress;
   private ChatServer chatServer;

   // RMIMessageManager constructor
   public RMIMessageManager( String server ) throws RemoteException
   {
      serverAddress = server;
   }

   // connect to ChatServer
   public void connect( MessageListener listener )
      throws Exception
   {
      // look up ChatServer remote object
      chatServer = ( ChatServer ) Naming.lookup(
         "//" + serverAddress + "/ChatServer" );

      // register with ChatServer to receive messages
      chatServer.registerClient( this );

      // set listener for incoming messages
      messageListener = listener;

   } // end method connect

   // disconnect from ChatServer
   public void disconnect( MessageListener listener )
      throws Exception
   {
      if ( chatServer == null )
         return;

      // unregister with ChatServer
      chatServer.unregisterClient( this );

      // remove references to ChatServer and MessageListener
      chatServer = null;
      messageListener = null;

   } // end method disconnect

   // send ChatMessage to ChatServer
   public void sendMessage( String fromUser, String message )
      throws Exception
   {
      if ( chatServer == null )
         return;

      // create ChatMessage with message text and userName
      ChatMessage chatMessage =
         new ChatMessage( fromUser, message );

```

Fig. 13.20 **RMIMessageManager** remote object and **MessageManager** implementation for managing **ChatClient** communication (part 2 of 3).

```
77          // post message to ChatServer
78          chatServer.postMessage( chatMessage );
79
80       }  // end method sendMessage
81
82       // process delivery of ChatMessage from ChatServer
83       public void deliverMessage( ChatMessage message )
84          throws RemoteException
85       {
86          if ( messageListener != null )
87             messageListener.messageReceived( message.getSender(),
88                message.getMessage() );
89       }
90
91       // method called when server shutting down
92       public void serverStopping() throws RemoteException
93       {
94          chatServer = null;
95          fireServerDisconnected( "Server shut down." );
96       }
97
98       // register listener for disconnect notifications
99       public void setDisconnectListener(
100         DisconnectListener listener )
101      {
102         disconnectListener = listener;
103      }
104
105      // send disconnect notification
106      private void fireServerDisconnected( String message )
107      {
108         if ( disconnectListener != null )
109            disconnectListener.serverDisconnected( message );
110      }
111   }
```

Fig. 13.20 **RMIMessageManager** remote object and **MessageManager** implementation for managing **ChatClient** communication (part 3 of 3).

Method **connect** (lines 35–48)—declared in interface **MessageManager**—connects the **RMIMessageManager** to the **ChatServer**. Lines 39–40 invoke **static** method **lookup** of class **Naming** to retrieve a remote reference to the **ChatServer**. Line 43 invokes method **registerClient** of interface **ChatServer** to register the **RMIMessageManager** for RMI callbacks from the **ChatServer**. Note that line 43 passes the **this** reference as the argument to method **registerClient**. Recall that class **RMIMessageManager** is a remote object, therefore the **this** reference can serve as a remote **ChatClient** reference to the **RMIMessageManager** remote object.

Method **disconnect** (lines 51–64) disconnects the **RMIMessageManager** from the **ChatServer**. If remote **ChatServer** reference **chatServer** is **null**, line 55 returns immediately, because the **RMIMessageManager** is disconnected already. Line 58 invokes method **unregisterClient** of remote interface **ChatServer** to unregister the **RMIMessageManager** from the **ChatServer**. Line 58 passes the **this** reference as an argument to method **unregisterClient**, specifying that the **ChatServer** should **unregister** this **RMIMessageManager** remote object. Line 62 sets **MessageListener** reference **messageListener** to **null**.

Method **sendMessage** (lines 67–80) delivers a message from the client to the **ChatServer**. Line 71 returns immediately if the **chatServer** remote reference is **null**. Lines 74–75 create a new **ChatMessage** object to contain the user name from whom the message came and the message body. Line 78 invokes method **postMessage** of remote interface **ChatServer** to post the new **ChatMessage** to the **ChatServer**. The **ChatServer** will use RMI callbacks to deliver this message to each registered **ChatClient**.

Method **deliverMessage** (lines 83–89)—defined in remote interface **ChatClient**—enables the **ChatServer** to use RMI callbacks to deliver incoming **ChatMessages** to the **ChatClient**. If there is a **MessageListener** registered with the **RMIMessageManager** (line 86), lines 87–88 invoke method **messageReceived** of interface **MessageListener** to notify the **MessageListener** of the incoming **ChatMessage**. Lines 87–88 invoke methods **getSender** and **getMessage** of class **ChatMessage** to retrieve the message sender and message body, respectively.

Method **serverStopping** (lines 92–96)—defined in remote interface **ChatClient**—enables the **ChatServer** to use RMI callbacks to notify the **ChatClient** that the **ChatServer** is shutting down so the **ChatClient** can disconnect and notify the **DisconnectListener**. Line 95 invokes method **fireServerDisconnected** of class **RMIMessageManager** to notify the registered **DisconnectListener** that the **ChatServer** has disconnected the **ChatClient**.

Method **setDisconnectListener** (lines 99–103)—defined in interface **MessageManager**—enables a **DisconnectListener** to register for notifications when the **ChatServer** disconnects the client. For example, the client user interface could register for these notifications to notify the user that the server has disconnected. Method **fireServerDisconnected** (lines 106–110) is a utility method for sending **serverDisconnected** messages to the **DisconnectListener**. If there is a registered **DisconnectListener**, line 109 invokes method **serverDisconnected** of interface **DisconnectListener** to notify the listener that the server disconnected. We discuss interface **DisconnectListener** in detail when we present the client user interface.

Client GUI Interfaces and Implementation

We uncouple the client user interface from the **MessageManager** implementation through interfaces **MessageListener** and **DisconnectListener** (Fig. 13.19 and 13.20). Class **ClientGUI** uses implementations of interfaces **MessageListener** and **DisconnectListener** to interact with the **MessageManager** and provides a graphical user interface for the client.

Interface **MessageListener** (Fig. 13.21) enables objects of an implementing class to receive incoming messages from a **MessageManager**. Line 9 defines method **messageReceived**, which takes as arguments the user name **from** whom the message came and the **message** body.

Interface **DisconnectListener** (Fig. 13.22) enables implementing objects to receive notifications when the server disconnects the **MessageManager**. Line 9 defines method **serverDisconnected**, which takes as a **String** argument a **message** that indicates why the server disconnected.

```
// MessageListener.java
// MessageListener is an interface for classes that wish to
// receive new chat messages.
package com.deitel.messenger;

public interface MessageListener {

   // receive new chat message
   public void messageReceived( String from, String message );
}
```

Fig. 13.21 **MessageListener** interface for receiving new messages.

```
// DisconnectListener.java
// DisconnectListener defines method serverDisconnected, which
// indicates that the server has disconnected the client.
package com.deitel.messenger;

public interface DisconnectListener {

   // receive notification that server disconnected
   public void serverDisconnected( String message );
}
```

Fig. 13.22 **DisconnectListener** interface for receiving server disconnect notifications.

Class **ClientGUI** (Fig. 13.23) provides a user interface for the Deitel Messenger client. The GUI consists of a menu and a toolbar with **Action**s for connecting to and disconnecting from a **ChatServer**, a **JTextArea** for displaying incoming **ChatMessages** and a **JTextArea** and **JButton** for sending new messages to the **ChatServer**. Lines 27–29 declare **Action** references for connecting to and disconnecting from the **ChatServer** and for sending **ChatMessage**s. Line 35 declares a **MessageManager** reference for the **MessageManager** implementation that provides the network communication. Line 38 declares a **MessageListener** reference for receiving new **ChatMessage**s from the **ChatServer** through the **MessageManager**.

```
1  // ClientGUI.java
2  // ClientGUI provides a GUI for sending and receiving
3  // chat messages using a MessageManager.
4  package com.deitel.messenger;
5
6  // Java core packages
7  import java.awt.*;
8  import java.awt.event.*;
9  import java.util.*;
```

Fig. 13.23 **ClientGUI** provides a graphical user interface for the Deitel Messenger client (part 1 of 9).

```java

// Java extension packages
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.text.*;

public class ClientGUI extends JFrame {

   // JLabel for displaying connection status
   private JLabel statusBar;

   // JTextAreas for displaying and inputting messages
   private JTextArea messageArea;
   private JTextArea inputArea;

   // Actions for connecting and disconnecting MessageManager
   // and sending messages
   private Action connectAction;
   private Action disconnectAction;
   private Action sendAction;

   // userName to add to outgoing messages
   private String userName = "";

   // MessageManager for communicating with server
   MessageManager messageManager;

   // MessageListener for receiving new messages
   MessageListener messageListener;

   // ClientGUI constructor
   public ClientGUI( MessageManager manager )
   {
      super( "Deitel Messenger" );

      messageManager = manager;

      messageListener = new MyMessageListener();

      // create Actions
      connectAction = new ConnectAction();
      disconnectAction = new DisconnectAction();
      disconnectAction.setEnabled( false );
      sendAction = new SendAction();
      sendAction.setEnabled( false );

      // set up File menu
      JMenu fileMenu = new JMenu ( "File" );
      fileMenu.setMnemonic( 'F' );
      fileMenu.add( connectAction );
      fileMenu.add( disconnectAction );

```

Fig. 13.23 **ClientGUI** provides a graphical user interface for the Deitel Messenger client (part 2 of 9).

```
      // set up JMenuBar and attach File menu
      JMenuBar menuBar = new JMenuBar();
      menuBar.add ( fileMenu );
      setJMenuBar( menuBar );

      // set up JToolBar
      JToolBar toolBar = new JToolBar();
      toolBar.add( connectAction );
      toolBar.add( disconnectAction );

      // create JTextArea for displaying messages
      messageArea = new JTextArea( 15, 15 );

      // disable editing and wrap words at end of line
      messageArea.setEditable( false );
      messageArea.setLineWrap( true );
      messageArea.setWrapStyleWord( true );

      JPanel panel = new JPanel();
      panel.setLayout( new BorderLayout( 5, 5 ) );
      panel.add( new JScrollPane( messageArea ),
         BorderLayout.CENTER );

      // create JTextArea for entering new messages
      inputArea = new JTextArea( 3, 15 );
      inputArea.setLineWrap( true );
      inputArea.setWrapStyleWord( true );
      inputArea.setEditable( false );

      // map Enter key in inputArea to sendAction
      Keymap keyMap = inputArea.getKeymap();
      KeyStroke enterKey = KeyStroke.getKeyStroke(
         KeyEvent.VK_ENTER, 0 );
      keyMap.addActionForKeyStroke( enterKey, sendAction );

      // lay out inputArea and sendAction JButton in BoxLayout
      // and add Box to messagePanel
      Box box = new Box( BoxLayout.X_AXIS );
      box.add( new JScrollPane( inputArea ) );
      box.add( new JButton( sendAction ) );

      panel.add( box, BorderLayout.SOUTH );

      // create statusBar JLabel with recessed border
      statusBar = new JLabel( "Not Connected" );
      statusBar.setBorder(
         new BevelBorder( BevelBorder.LOWERED ) );

      // lay out components
      Container container = getContentPane();
      container.add( toolBar, BorderLayout.NORTH );
      container.add( panel, BorderLayout.CENTER );
```

Fig. 13.23 **ClientGUI** provides a graphical user interface for the Deitel Messenger client (part 3 of 9).

```
      container.add( statusBar, BorderLayout.SOUTH );

      // disconnect and exit if user closes window
      addWindowListener(

         new WindowAdapter() {

            // disconnect MessageManager when window closes
            public void windowClosing( WindowEvent event )
            {
               // disconnect from chat server
               try {
                  messageManager.disconnect( messageListener );
               }

               // handle exception disconnecting from server
               catch ( Exception exception ) {
                  exception.printStackTrace();
               }

               System.exit( 0 );

            } // end method windowClosing

         } // end WindowAdapter inner class
      );

   } // end ClientGUI constructor

   // Action for connecting to server
   private class ConnectAction extends AbstractAction {

      // configure ConnectAction
      public ConnectAction()
      {
         putValue( Action.NAME, "Connect" );
         putValue( Action.SMALL_ICON, new ImageIcon(
            ClientGUI.class.getResource(
               "images/Connect.gif" ) ) );
         putValue( Action.SHORT_DESCRIPTION,
            "Connect to Server" );
         putValue( Action.LONG_DESCRIPTION,
            "Connect to server to send Instant Messages" );
         putValue( Action.MNEMONIC_KEY, new Integer( 'C' ) );
      }

      // connect to server
      public void actionPerformed( ActionEvent event )
      {
         // connect MessageManager to server
         try {

```

Fig. 13.23 **ClientGUI** provides a graphical user interface for the Deitel Messenger client (part 4 of 9).

```
            // clear messageArea
            messageArea.setText( "" );

            // connect MessageManager and register MessageListener
            messageManager.connect( messageListener );

            // listen for disconnect notifications
            messageManager.setDisconnectListener(
               new DisconnectHandler() );

            // get desired userName
            userName = JOptionPane.showInputDialog(
               ClientGUI.this, "Please enter your name: " );

            // update Actions, inputArea and statusBar
            connectAction.setEnabled( false );
            disconnectAction.setEnabled( true );
            sendAction.setEnabled( true );
            inputArea.setEditable( true );
            inputArea.requestFocus();
            statusBar.setText( "Connected: " + userName );

            // send message indicating user connected
            messageManager.sendMessage( userName, userName +
               " joined chat" );

         } // end try

         // handle exception connecting to server
         catch ( Exception exception ) {
            JOptionPane.showMessageDialog( ClientGUI.this,
               "Unable to connect to server.", "Error Connecting",
               JOptionPane.ERROR_MESSAGE );

            exception.printStackTrace();
         }

      }  // end method actionPerformed

   } // end ConnectAction inner class

   // Action for disconnecting from server
   private class DisconnectAction extends AbstractAction {

      // configure DisconnectAction
      public DisconnectAction()
      {
         putValue( Action.NAME, "Disconnect" );
         putValue( Action.SMALL_ICON, new ImageIcon(
            ClientGUI.class.getResource(
               "images/Disconnect.gif" ) ) );
```

Fig. 13.23 **ClientGUI** provides a graphical user interface for the Deitel Messenger client (part 5 of 9).

```
         putValue( Action.SHORT_DESCRIPTION,
            "Disconnect from Server" );
         putValue( Action.LONG_DESCRIPTION,
            "Disconnect to end Instant Messaging session" );
         putValue( Action.MNEMONIC_KEY, new Integer( 'D' ) );
      }

      // disconnect from server
      public void actionPerformed( ActionEvent event )
      {
         // disconnect MessageManager from server
         try {

            // send message indicating user disconnected
            messageManager.sendMessage( userName, userName +
               " exited chat" );

            // disconnect from server and unregister
            // MessageListener
            messageManager.disconnect( messageListener );

            // update Actions, inputArea and statusBar
            sendAction.setEnabled( false );
            disconnectAction.setEnabled( false );
            inputArea.setEditable( false );
            connectAction.setEnabled( true );
            statusBar.setText( "Not Connected" );

         } // end try

         // handle exception disconnecting from server
         catch ( Exception exception ) {
            JOptionPane.showMessageDialog( ClientGUI.this,
               "Unable to disconnect from server.",
               "Error Disconnecting", JOptionPane.ERROR_MESSAGE );

            exception.printStackTrace();
         }

      } // end method actionPerformed

   } // end DisconnectAction inner class

   // Action for sending messages
   private class SendAction extends AbstractAction {

      // configure SendAction
      public SendAction()
      {
         putValue( Action.NAME, "Send" ) ;
         putValue( Action.SMALL_ICON, new ImageIcon(
            ClientGUI.class.getResource( "images/Send.gif" ) ) );
```

Fig. 13.23 **ClientGUI** provides a graphical user interface for the Deitel Messenger client (part 6 of 9).

```
         putValue( Action.SHORT_DESCRIPTION, "Send Message" );
         putValue( Action.LONG_DESCRIPTION,
            "Send an Instant Message" );
         putValue( Action.MNEMONIC_KEY, new Integer( 'S' ) );
      }

      // send message and clear inputArea
      public void actionPerformed( ActionEvent event )
      {
         // send message to server
         try {

            // send userName and text in inputArea
            messageManager.sendMessage( userName,
               inputArea.getText() );

            inputArea.setText( "" );
         }

         // handle exception sending message
         catch ( Exception exception ) {
            JOptionPane.showMessageDialog( ClientGUI.this,
               "Unable to send message.", "Error Sending Message",
               JOptionPane.ERROR_MESSAGE );

            exception.printStackTrace();
         }

      } // end method actionPerformed

   } // end SendAction inner class

   // MyMessageListener listens for new messages from the
   // MessageManager and displays the messages in messageArea
   // using a MessageDisplayer.
   private class MyMessageListener implements MessageListener {

      // when new message received, display in messageArea
      public void messageReceived( String from, String message )
      {
         // append message using MessageDisplayer and invokeLater
         // to ensure thread-safe access to messageArea
         SwingUtilities.invokeLater(
            new MessageDisplayer( from, message ) );
      }

   }  // end MyMessageListener inner class

   // MessageDisplayer displays a new messaage by appending
   // the message to the messageArea JTextArea. This Runnable
   // object should be executed only on the event-dispatch
   // thread, as it modifies a live Swing component.
```

Fig. 13.23 **ClientGUI** provides a graphical user interface for the Deitel Messenger client (part 7 of 9).

```java
   private class MessageDisplayer implements Runnable {

      private String fromUser;
      private String messageBody;

      // MessageDisplayer constructor
      public MessageDisplayer( String from, String body )
      {
         fromUser = from;
         messageBody = body;
      }

      // display new message in messageArea
      public void run()
      {
         // append new message
         messageArea.append( "\n" + fromUser + "> " +
            messageBody );

         // move caret to end of messageArea to ensure new
         // message is visible on screen
         messageArea.setCaretPosition(
           messageArea.getText().length() );
      }

   } // end MessageDisplayer inner class

   // DisconnectHandler listens for serverDisconnected messages
   // from the MessageManager and updates the user interface.
   private class DisconnectHandler implements DisconnectListener {

      // receive disconnect notifcation
      public void serverDisconnected( final String message )
      {
         // update GUI in thread-safe manner
         SwingUtilities.invokeLater(

            new Runnable() {

               // update Actions, inputs and status bar
               public void run()
               {
                  sendAction.setEnabled( false );
                  disconnectAction.setEnabled( false );
                  inputArea.setEditable( false );
                  connectAction.setEnabled( true );
                  statusBar.setText( message );
               }

            } // end Runnable inner class
         );

```

Fig. 13.23 **ClientGUI** provides a graphical user interface for the Deitel Messenger client (part 8 of 9).

```
373          } // end method serverDisconnected
374
375       } // end DisconnectHandler inner class
376    }
```

Fig. 13.23 **ClientGUI** provides a graphical user interface for the Deitel Messenger client (part 9 of 9).

The **ClientGUI** constructor (lines 41–141) creates and lays out the various user-interface components. The constructor takes as an argument the **MessageManager** that implements the underlying network communications. A **WindowAdapter** inner class (lines 119–138) ensures that the **MessageManager** disconnects from the **ChatServer** (line 126) when the user closes the application window.

The **ConnectAction** inner class (lines 144–205) is an **Action** implementation for connecting to the Deitel Messenger server. Lines 170–174 invoke method **connect** of interface **MessageManager** and register a **DisconnectListener** for receiving **serverDisconnected** notifications. Lines 177–186 prompt the user for a name to use in the chat session and update the user-interface components to allow the user to send messages and disconnect from the Deitel Messenger server. Lines 188–189 invoke method **sendMessage** of interface **MessageManager** to send a **ChatMessage** that announces the user's arrival in the chat session.

The **DisconnectAction** inner class (lines 211–258) is an **Action** implementation for disconnecting the **MessageManager** from the Deitel Messenger server. Lines 231–232 send a **ChatMessage** to announce the user's departure from the chat session. Line 236 invokes method **disconnect** of interface **MessageManager** to disconnect from the server. Lines 239–243 update the user-interface components to disable the message **inputArea** and display a message in the status bar.

The **SendAction** inner class (lines 261–299) is an **Action** implementation for sending messages to the server. Lines 282–283 invoke method **sendMessage** of interface **MessageManager** and pass the contents of **inputArea** and the user's **userName** as arguments.

An instance of inner class **MyMessageListener** (lines 304–315) listens for incoming **ChatMessage**s. When the **MessageManager** receives a new **ChatMessage** from the server, the **MessageManager** invokes method **messageReceived** (lines 307–313). Lines 311–312 invoke **static** method **invokeLater** of class **SwingUtilities** with a **MessageDisplayer** argument to display the new message.

Inner class **MessageDisplayer** (lines 321–346) is a **Runnable** implementation that appends a new message to the **messageArea JTextArea** to display that message to the user. Lines 337–338 append the message text and sender's user name to **messageArea**, and lines 342–343 move the cursor to the end of **messageArea**.

An instance of inner class **DisconnectHandler** (lines 350–375) receives **serverDisconnected** notifications from the **MessageManager** when the server disconnects. Lines 356–371 update the user-interface components to indicate that the server disconnected.

Class **DeitelMessenger** (Fig. 13.24) launches the client application using a **ClientGUI** and **RMIMessageManager**. Line 18 invokes method **setSecurityManager** of class **System** to install an **RMISecurityManager** for the client application.

The client requires this **SecurityManager** for downloading the **ChatServer**'s stub dynamically. We discuss dynamic class downloading in Section 13.6.3. If the user does not specify a hostname for the **ChatServer**, line 24 creates an **RMIMessageManager** that connects to the server running on **localhost**. Line 26 creates an **RMIMessageManager** that connects to the user-provided hostname. Lines 29–32 create a **ClientGUI** for the **RMIMessageManager** and display that GUI to the user.

13.6.3 Running the Deitel Messenger Server and Client Applications

Running the Deitel Messenger case study server and clients requires several steps. In addition to the RMI registry, RMI applications that use **Activatable** objects require the RMI activation daemon (**rmid**). The RMI activation daemon is a server process that manages the registration, activation and deactivation of **Activatable** remote objects.\

```
 1  // DeitelMessenger.java
 2  // DeitelMessenger uses a ClientGUI and RMIMessageManager to
 3  // implement an RMI-based chat client.
 4  package com.deitel.messenger.rmi.client;
 5
 6  // Java core packages
 7  import java.rmi.RMISecurityManager;
 8
 9  // Deitel packages
10  import com.deitel.messenger.*;
11
12  public class DeitelMessenger {
13
14     // launch DeitelMessenger application
15     public static void main ( String args[] ) throws Exception
16     {
17        // install RMISecurityManager
18        System.setSecurityManager( new RMISecurityManager() );
19
20        MessageManager messageManager;
21
22        // create new DeitelMessenger
23        if ( args.length == 0 )
24           messageManager = new RMIMessageManager( "localhost" );
25        else
26           messageManager = new RMIMessageManager( args[ 0 ] );
27
28        // finish configuring window and display it
29        ClientGUI clientGUI = new ClientGUI( messageManager );
30        clientGUI.pack();
31        clientGUI.setResizable( false );
32        clientGUI.setVisible( true );
33     }
34  }
```

Fig. 13.24 **DeitelMessenger** launches a chat client using classes **ClientGUI** and **RMIMessageManager**.

To begin, start the RMI registry by executing the command

```
rmiregistry
```

at a command prompt. Be sure that the stub file for the **ChatServer** remote object (**ChatServerImpl_Stub.class**) is not in the RMI registry's **CLASSPATH**, as this will disable dynamic class downloading. Next, start the RMI activation daemon by executing the command

```
rmid -J-Djava.security.policy=rmid.policy
```

where **rmid.policy** is the complete path to the policy file of Fig. 13.25. This policy file allows the **ActivationGroup** in which the **ChatServer** runs to specify **C:\activationGroup.policy** as the policy file for the **ActivationGroup**'s virtual machine. If you place **activationGroup.policy** in a location other than the **C:\** directory, be sure to modify **rmid.policy** to specify the appropriate location.

Dynamic class downloading enables Java programs to download classes not available in the local **CLASSPATH**. This is particularly useful in RMI applications for enabling clients to download stub files dynamically. When an RMI object specifies the **java.rmi.server.codebase** system property, the RMI registry adds an *annotation* to that object's remote references. This annotation specifies the codebase from which clients can download any necessary classes. These classes might include the stub for the remote object and other classes. These **.class** files must be available for download from an HTTP server. Sun provides a basic HTTP server suitable for testing purposes, which is downloadable from

```
java.sun.com/products/jdk/rmi/class-server.zip
```

Extract the files from **class-server.zip** and read the included instructions for running the HTTP server. Figure 13.26 lists the files to include in the HTTP server's download directory. For example, if the HTTP server's download directory is **C:\classes**, copy the directory structure and **.class** files listed in Fig. 13.26 to **C:\classes**. Be sure to start the HTTP server before continuing.

Next, run the **ChatServerAdministrator** application to launch the **Activatable** remote object by using the command

```
java -Djava.security.policy=administrator.policy
   -Djava.rmi.server.codebase=http://hostname:port/
   com.deitel.messenger.rmi.server.ChatServerAdministrator
   start
```

```
// allow ActivationGroup to specify C:\activationGroup.policy
// as its VM's security policy
grant {
   permission com.sun.rmi.rmid.ExecOptionPermission
      "-Djava.security.policy=file:///C:/activationGroup.policy";
};
```

Fig. 13.25 Policy file for the RMI activation daemon.

Directory	File Name
com\deitel\messenger\rmi\server\	
	ChatServer.class
	ChatServerImpl.class
	ChatServerImpl$1.class
	ChatServerImpl_Stub.class
	StoppableChatServer.class
com\deitel\messenger\rmi\client\	
	ChatClient.class
	RMIMessageManager_Stub.class
com\deitel\messenger\rmi\	
	ChatMessage.class

Fig. 13.26 File listing for the HTTP server's download directory.

where **administrator.policy** is the complete path to the policy file of Fig. 13.27, *hostname* is the name of the computer running the HTTP server and *port* is the port number on which that HTTP server is running. The RMI registry will annotate each remote reference it returns with this codebase. The policy file must permit **ChatServerAdministrator** to connect to port 1098 on the local machine, which is the port for the RMI activation daemon. The policy file also must allow the **ChatServerAdministrator** to access the port on which the Web server is running. Lines 4–5 of Fig. 13.27 specify that the **ChatServerAdministrator** can access all ports above and including **1024** on *hostname*. Be sure to replace *hostname* with the appropriate name or IP address of the machine running the Web server and RMI activation daemon. The **ChatServerAdministrator** also requires the permission **setFactory** of type **java.lang.RuntimePermission**, which permits the **ActivationGroup** to set a **SecurityManager**.

The **ChatServerAdministrator** application registers the **ActivationGroup** for the **Activatable ChatServer**, then exits. Clients then may access the **ChatServer** by obtaining a remote reference to the **ChatServer** from the RMI registry and invoking methods on that remote reference. Note that the **ChatServer** does not begin executing until the first client invokes a method on the **ChatServer** remote object.

```
1  // allow ChatServerAdministrator to connect to
2  // activation daemon
3  grant {
4     permission java.net.SocketPermission "hostname:1024-",
5        "connect, accept, resolve";
6
7     permission java.lang.RuntimePermission "setFactory";
8  };
```

Fig. 13.27 Policy file for **ChatServerAdministrator**.

At that time, the activation system activates the **ChatServer**'s **ActivationGroup**. To launch a client for the **ChatServer**, type the following at a command prompt:

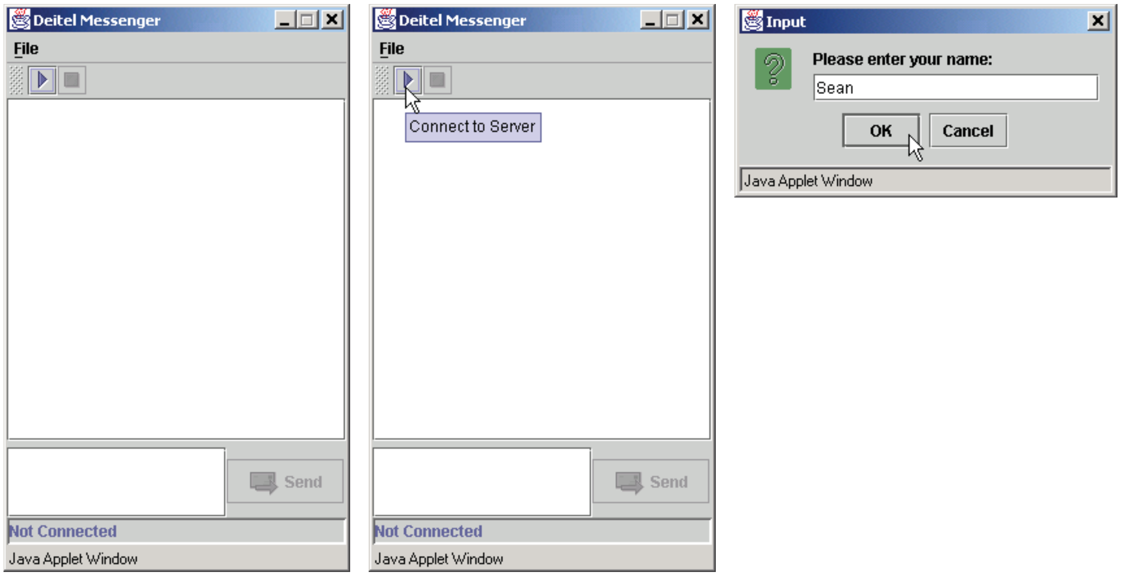
```
java -Djava.security.policy=client.policy
   com.deitel.messenger.rmi.client.DeitelMessenger
```

where **client.policy** is the policy file of Fig. 13.28. This policy file enables the client to connect, accept and resolve connnections to the specified hostname on ports above and including **1024**. Recall that the client is itself a remote object, so the client must be able to accept incoming network connections from the **ChatServer**. Be sure to replace **hostname** with the hostname or IP address of the computer on which the **ChatServer** is running.

Figure 13.29 shows a sample conversation in Deitel Messenger. Notice that the GUI elements properly reflect the current connection state—when the client is disconnected, only the **ConnectAction** is enabled. After the client connects, the **DisconnectAction**, input **JTextArea** and **SendAction** become enabled. Note also that the bottom of each window displays the message **Java Applet Window**. The virtual machine places this message in the windows because the application is running under security restrictions.

```
// allow client to connect to network resources on hostname
// at ports above 1024
grant {
   permission java.net.SocketPermission "hostname:1024-",
      "connect, accept, resolve";
};
```

Fig. 13.28 Policy file for the **DeitelMessenger** client.

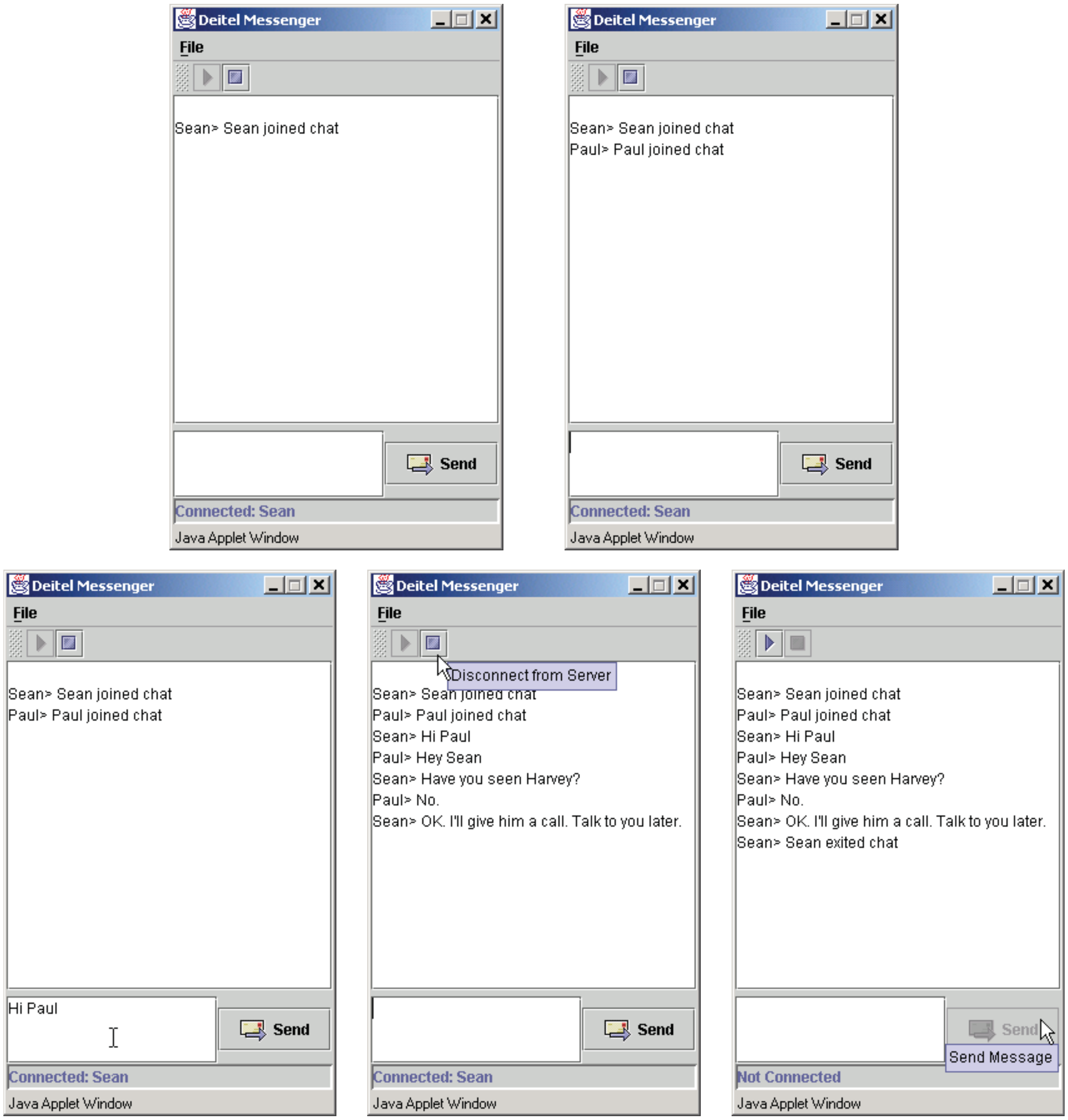


Not Connected
Java Applet Window

Not Connected
Java Applet Window

Please enter your name:
Sean
Java Applet Window

Fig. 13.29 Sample conversation using Deitel Messenger.



```
Sean> Sean joined chat
```

```
Sean> Sean joined chat
Paul> Paul joined chat
```

```
Sean> Sean joined chat
Paul> Paul joined chat
```

```
Hi Paul
```

```
Sean> Sean joined chat
Paul> Paul joined chat
Sean> Hi Paul
Paul> Hey Sean
Sean> Have you seen Harvey?
Paul> No.
Sean> OK. I'll give him a call. Talk to you later.
```

```
Sean> Sean joined chat
Paul> Paul joined chat
Sean> Hi Paul
Paul> Hey Sean
Sean> Have you seen Harvey?
Paul> No.
Sean> OK. I'll give him a call. Talk to you later.
Sean> Sean exited chat
```

Fig. 13.29 Sample conversation using Deitel Messenger.

13.7 Internet and World Wide Web Resources

`java.sun.com/products/jdk/rmi/index.html`
Sun's Remote Method Invocation (RMI) home page, which provides links to technical articles, documentation and other resources.

`java.sun.com/j2se/1.3/docs/guide/rmi/index.html`
Sun's RMI guide, which includes links to tutorials on building activatable remote objects and other useful resources.

`www.jguru.com/faq/home.jsp?topic=RMI`
jGuru's RMI Frequently Asked Questions with answers, which provides tips and answers to many common questions that developer's ask about RMI.

`www.javaworld.com/javaworld/topicalindex/jw-ti-rmi.html`
JavaWorld's list of articles related to RMI. Articles include discussions of activatable RMI objects, integrating RMI with CORBA and RMI-related technologies, such as Jini.

SUMMARY

- RMI allows Java objects running on separate computers or in separate processes to communicate with one another via remote method calls. Such method calls appear to the programmer the same as those operating on objects in the same program.
- RMI is based on a similar, earlier technology for procedural programming called remote procedure calls (RPCs) developed in the 1980s.
- RMI enables Java programs to transfer complete Java objects using Java's object-serialization mechanism. The programmer need not be concerned with the transmission of the data over the network.
- For Java-to-non-Java communication, you can use Java IDL (introduced in Java 1.2) or RMI-IIOP. Java IDL and RMI-IIOP enable applications and applets written in Java to communicate with objects written in any language that supports CORBA (Common Object Request Broker Architecture).
- The four major steps for building an RMI distributed system are 1) defining the remote interface, 2) defining the remote object implementation, 3) defining the client application that uses the remote object and 4) compiling and executing the remote object and the client.
- To create a remote interface, define an interface that extends interface ***java.rmi.Remote***. Interface **Remote** is a tagging interface—it does not declare any methods, and therefore places no burden on the implementing class.
- An object of a class that implements interface **Remote** directly or indirectly is a remote object and can be accessed—with appropriate security permissions—from any Java virtual machine that has a connection to the computer on which the remote object executes.
- Every remote method must be declared in an interface that extends ***java.rmi.Remote***. A remote object must implement all methods declared in its remote interface.
- An RMI distributed application must export an object of a class that implements the ***Remote*** interface to make that remote object available to receive remote method calls.
- Each method in a **Remote** interface must have a **throws** clause that indicates that the method can throw **RemoteException**s. A **RemoteException** indicates a problem communicating with the remote object.
- RMI uses Java's default serialization mechanism to transfer method arguments and return values across the network. Therefore, all method arguments and return values must be **Serializable** or primitive types.
- Class **UnicastRemoteObject** provides the basic functionality required for all remote objects. In particular, its constructor exports the object to make it available to receive remote calls.
- Exporting a remote object enables that object to wait for client connections on an anonymous port number (i.e., one chosen by the computer on which the remote object executes). RMI abstracts away communication details so the programmer can work with simple method calls.
- Constructors for class **UnicastRemoteObject** allow the programmer to specify information about the remote object, such as an explicit port number on which to export the remote object. All **UnicastRemoteObject** constructors throw **RemoteException**s.
- The **rmiregistry** utility program manages the registry for remote objects and is part of the J2SE SDK. The default port number for the RMI registry is **1099**.
- Method **lookup** connects to the RMI registry and returns a **Remote** reference to the remote object. Note that clients refer to remote objects only through those object's remote interfaces.
- A remote reference refers to a stub object on the client. Stubs allow clients to invoke remote objects' methods. Stub objects receive each remote method call and pass those calls to the RMI system, which performs the networking that allows clients to interact with the remote object.

- The RMI layer is responsible for network connections to the remote object, so referencing remote objects is transparent to the client. RMI handles the underlying communication with the remote object and the transfer of arguments and return values between the objects. Argument and return types for remote methods must be **Serializable**.
- The **rmic** utility compiles the remote object class to produce a stub class. A stub class forwards method invocations to the RMI layer, which performs the network communication necessary to invoke the method call on the remote object.
- Standard RMI objects exported as **UnicastRemoteObject**s must run continuously on the server to handle client requests. RMI objects that extend class **java.rmi.activation.Activatable** are able to activate, or start running, when a client invokes one of the remote object's methods.
- The RMI activation daemon (**rmid**) is a server process that enables activatable remote objects to become active when clients invoke remote methods on these objects.
- Activatable remote objects also are able to recover from server crashes, because remote references to activatable objects are persistent—when the server restarts, the RMI activation daemon maintains the remote reference, so clients can continue to use the remote object.
- The RMI activation mechanism requires that **Activatable** objects provide a constructor that takes as arguments an **ActivationID** and a **MarshalledObject**. When the activation daemon activates a remote object of this class, it invokes this activation constructor. The **ActivationID** argument specifies a unique identifier for the remote object.
- Class **MarshalledObject** is a wrapper class that contains a serialized object for transmission over RMI. The **MarshalledObject** passed to the activation constructor can contain application-specific initialization information, such as the name under which the activation daemon registered the remote object.
- Activatable RMI objects execute as part of an ***ActivationGroup*** (package **java.rmi.activation**). The RMI activation daemon—a server-side process that manages activatable objects—starts a new virtual machine for each **ActivationGroup**.
- Class **ActivationGroupDesc** specifies configuration information for an **ActivationGroup**. The first argument to the **ActivationGroupDesc** constructor is a **Properties** reference that contains replacement values for system properties in the **ActivationGroup**'s virtual machine. The second argument is a reference to an **ActivationGroupDesc.CommandEnvironment** object, which enables the **ActivationGroup** to customize the commands that the activation daemon executes when starting the **ActivationGroup**'s virtual machine.
- The incarnation number of an **ActivationGroup** identifies different instances of the same **ActivationGroup**. Each time the activation daemon activates the **ActivationGroup**, the daemon increments the incarnation number.
- Class **ActivationDesc** specifies configuration information for a particular **Activatable** remote object. The first argument to the **ActivationDesc** constructor specifies the name of the class that implements the **Activatable** remote object. The second argument specifies the codebase that contains the remote object's class files. The final argument is a **MarshalledObject** reference, whose object specifies initialization information for the remote object.
- Method **register** of class **Activatable** takes as an argument the **ActivationDesc** for the **Activatable** object and returns a reference to the remote object's stub.
- Dynamic class downloading enables Java programs to download classes not available in the local **CLASSPATH**. This is particularly useful in RMI applications for enabling clients to download stub files dynamically.
- When an RMI object specifies the **java.rmi.server.codebase** system property, the RMI registry adds an annotation to that object's remote references, which specifies the codebase from

which clients can download necessary classes. Downloadable **.class** files must be available from an HTTP server.

TERMINOLOGY

Activatable class (package **java.rmi.activation**)
activatable remote object
activation daemon
activation descriptor
activation group descriptor
ActivationGroup class
ActivationGroupDesc class
ActivationGroupDescd.Command-Environment class
ActivationID class
ActivationSystem interface
Adapter design pattern
anonymous port number
bind method of class **Naming**
createRegistry method of class **LocateRegistry**
distributed computing
export
exportObject method of class **UnicastRemoteObject**
HTML scraping
Interface Definition Language (IDL)
ListCellRenderer interface
LocateRegistry class
marshaling of data
MarshalledObject class
rebind method of class **Naming**
Registry class
remote interface
Remote interface (package **java.rmi**)
remote method
remote method call
Remote Method Invocation (RMI)
remote object
remote object implementation
Remote Procedure Call (RPC)
remote reference
RemoteException class (package **java.rmi**)
RMI registry
rmic compiler
rmid utility
rmiregistry utility
RMISecurityManager class
stub class
tagging interface
UnicastRemoteObject class (package **java.rmi.server**)

SELF-REVIEW EXERCISES

13.1 Fill in the blanks in each of the following statements:

a) The remote object class must be compiled using the __________ to produce a stub class.
b) RMI is based on a similar technology for procedural programming called __________.
c) Clients use method __________ of class **Naming** to obtain a remote reference to a remote object.
d) To create a remote interface, define an interface that extends interface __________ of package __________.
e) Method __________ or __________ of class ***Naming*** binds a remote object to the RMI registry.
f) Remote objects normally extend class __________, which provides the basic functionality required for all remote objects.
g) Remote objects use the __________ and __________ to locate the RMI registry so they can register themselves as remote services. Clients use these to locate a service.
h) The default port number for the RMI registry is __________.
i) Interface **Remote** is a __________.
j) __________ allows Java objects running on separate computers (or possibly the same computer) to communicate with one another via remote method calls.

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

a) Not starting the RMI registry before attempting to bind the remote object to the registry results in a **RuntimeException** refusing connection to the registry.

b) Every remote method must be part of an interface that extends **java.rmi.Remote**.

c) The **stubcompiler** creates a stub class that performs the networking which allows the client to connect to the server and use the remote object's methods.

d) Class **UnicastRemoteObject** provides basic functionality required by remote objects.

e) An object of a class that implements interface **Serializable** can be registered as a remote object and receive a remote method call.

f) All methods in a ***Remote*** interface must have a **throws** clause indicating the potential for a **RemoteException**.

g) RMI clients assume that they should connect to port 80 on a server computer when attempting to locate a remote object through the RMI registry.

h) Once a remote object is bound to the RMI registry with method **bind** or **rebind** of class **Naming**, the client can look up the remote object with **Naming** method **lookup**.

i) Method **find** of class **Naming** interacts with the RMI registry to help the client obtain a reference to a remote object so the client can use the remote object's services.

ANSWERS TO SELF-REVIEW EXERCISES

13.1 a) **rmic** compiler. b) RPC. c) **lookup**. d) **Remote**, **java.rmi**. e) **bind**, **rebind**. f) **UnicastRemoteObject**. g) host, port. h) 1099. i) tagging interface. j) RMI.

13.2 a) False. This results in a **java.rmi.ConnectException**.

b) True.

c) False. The **rmic** compiler creates a stub class.

d) True.

e) False. An object of a class that implements a subinterface of **java.rmi.Remote** can be registered as a remote object and receive remote method calls.

f) True.

g) False. RMI clients assume port 1099 by default. Web browser clients assume port 80.

h) True.

i) False. Method **lookup** interacts with the RMI registry to help the client obtain a reference to a remote object.

EXERCISES

13.3 The current implementation of class **WeatherServiceImpl** downloads the weather information only once. Modify class **WeatherServiceImpl** to obtain weather information from the National Weather Service twice a day.

13.4 Modify interface **WeatherService** to include support for obtaining the current day's forecast and the next day's forecast. Study the Traveler's Forecast Web page

http://iwin.nws.noaa.gov/iwin/us/traveler.html

13.5 Visit the NWS Web site for the format of each line of information. Next, modify class **WeatherServiceImpl** to implement the new features of the interface. Finally, modify class **WeatherServiceClient** to allow the user to select the weather forecast for either day. Modify the support classes **WeatherBean** and **WeatherItem** as necessary to support the changes to classes **WeatherServiceImpl** and **WeatherServiceClient**.

13.6 (Project: Weather for Your State) There is a wealth of weather information on the National Weather Service Web site. Study the following Web pages:

```
http://iwin.nws.noaa.gov/
http://iwin.nws.noaa.gov/iwin/textversion/main.html
```

and create a complete weather forecast server for your state. Design your classes for reusability.

13.7 (Project: Weather for Your State) Modify the Exercise 13.6 project solution to allow the user to select the weather forecast for any state. [*Note*: For some states, the format of the weather forecast differs from the standard format. Your solution should allow the user to select only from those states whose forecasts are in the standard format.]

13.8 (*For International Readers*) If there is a similar World Wide Web-based weather service in your own country, provide a different **WeatherServiceImpl** implementation with the same remote interface **WeatherService** (Fig. 13.1). The server should return weather information for major cities in your country.

13.9 (*Remote Phone Book Server*) Create a remote phone book server that maintains a file of names and phone numbers. Define interface **PhoneBookServer** with the following methods:

```
public PhoneBookEntry[] getPhoneBook()
public void addEntry( PhoneBookEntry entry )
public void modifyEntry( PhoneBookEntry entry )
public void deleteEntry( PhoneBookEntry entry )
```

Create **Activatable** remote object class **PhoneBookServerImpl**, which implements interface **PhoneBookServer**. Class **PhoneBookEntry** should contain **String** instance variables that represent the first name, last name and phone number for one person. The class should also provide appropriate *set*/*get* methods and perform validation on the phone number format. Remember that class **PhoneBookEntry** also must implement **Serializable**, so that RMI can serialize objects of this class.

13.10 Class **PhoneBookClient** should provide a user interface that allows the user to scroll through entries, add a new entry, modify an existing entry and delete an existing entry. The client and the server should provide proper error handling (e.g., the client cannot modify an entry that does not exist).