Ordering Information:

Advanced Java 2 Platform How to Program

- View the complete **Table of Contents**
- Read the **Preface**
- Download the **Code Examples**

To view all the Deitel products and services available, visit the Deitel Kiosk on InformIT at **www.informIT.com/deitel**.

To follow the Deitel publishing program, sign-up now for the *DEITEL™ BUZZ ON-LINE* e-mail newsletter at **www.deitel.com/newsletter/subscribeinformIT.html**.

To learn more about our **Java and Advanced Java programming courses** or any other Deitel instructor-led corporate training courses that can be delivered at your location, visit **www.deitel.com/training**, contact our Director of Corporate Training Programs at (978) 461-5880 or e-mail: **christi.kelsey@deitel.com**.

Note from the Authors: This article is an excerpt from Chapter 13, Sections 13.2, 13.3 and 13.4 of *Advanced Java 2 Platform How to Program*. This article presents a case study for creating a distributed system with Remote Method Invocation (RMI). Readers should be familiar with Java and advanced Java topics. The code examples included in this article show readers programming examples using the DEITEL™ signature LIVE-CODE™ Approach, which presents all concepts in the context of complete working programs followed by the screen shots of the actual inputs and outputs.

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

```
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.

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 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

```
1  // WeatherServiceImpl.java
2  // WeatherServiceImpl implements the WeatherService remote
3  // interface to provide a WeatherService remote object.
4  package com.deitel.advjhtp1.rmi.weather;
5
6  // Java core packages
7  import java.io.*;
8  import java.net.URL;
9  import java.rmi.*;
10 import java.rmi.server.*;
11 import java.util.*;
12
13 public class WeatherServiceImpl extends UnicastRemoteObject
14    implements WeatherService {
15
16    private List weatherInformation;  // WeatherBean object List
17
18    // initialize server
19    public WeatherServiceImpl() throws RemoteException
20    {
21       super();
22       updateWeatherConditions();
23    }
24
25    // get weather information from NWS
26    private void updateWeatherConditions()
27    {
28       try {
29          System.out.println( "Update weather information..." );
30
31          // National Weather Service Traveler's Forecast page
32          URL url = new URL(
33             "http://iwin.nws.noaa.gov/iwin/us/traveler.html" );
34
35          // create BufferedReader for reading Web page contents
36          BufferedReader in = new BufferedReader(
37             new InputStreamReader( url.openStream() ) );
38
39          // separator for starting point of data on Web page
40          String separator = "TAV12";
41
42          // locate separator string in Web page
43          while ( !in.readLine().startsWith( separator ) )
44             ;    // do nothing
45
46          // strings representing headers on Traveler's Forecast
47          // Web page for daytime and nighttime weather
48          String dayHeader =
49             "CITY            WEA     HI/LO   WEA     HI/LO";
50          String nightHeader =
51             "CITY            WEA     LO/HI   WEA     LO/HI";
52
```

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

```
53          String inputLine = "";
54
55          // locate header that begins weather information
56          do {
57             inputLine = in.readLine();
58          } while ( !inputLine.equals( dayHeader ) &&
59                    !inputLine.equals( nightHeader ) );
60
61          weatherInformation = new ArrayList(); // create List
62
63          // create WeatherBeans containing weather data and
64          // store in weatherInformation List
65          inputLine = in.readLine();  // get first city's info
66
67          // The portion of inputLine containing relevant data is
68          // 28 characters long. If the line length is not at
69          // least 28 characters long, done processing data.
70          while ( inputLine.length() > 28 ) {
71
72             // Create WeatherBean object for city. First 16
73             // characters are city name. Next, six characters
74             // are weather description. Next six characters
75             // are HI/LO or LO/HI temperature.
76             WeatherBean weather = new WeatherBean(
77                inputLine.substring( 0, 16 ),
78                inputLine.substring( 16, 22 ),
79                inputLine.substring( 23, 29 ) );
80
81             // add WeatherBean to List
82             weatherInformation.add( weather );
83
84             inputLine = in.readLine();  // get next city's info
85          }
86
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   }
```

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

```
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).

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`**.]

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.

```
1  // WeatherBean.java
2  // WeatherBean maintains weather information for one city.
3  package com.deitel.advjhtp1.rmi.weather;
4
5  // Java core packages
6  import java.awt.*;
7  import java.io.*;
8  import java.net.*;
9  import java.util.*;
10
```

Fig. 13.3 **WeatherBean** stores weather forecast for one city (part 1 of 3).

```
11  // Java extension packages
12  import javax.swing.*;
13
14  public class WeatherBean implements Serializable {
15
16     private String cityName;         // name of city
17     private String temperature;      // city's temperature
18     private String description;      // weather description
19     private ImageIcon image;         // weather image
20
21     private static Properties imageNames;
22
23     // initialize imageNames when class WeatherBean
24     // is loaded into memory
25     static {
26        imageNames = new Properties();  // create properties table
27
28        // load weather descriptions and image names from
29        // properties file
30        try {
31
32           // obtain URL for properties file
33           URL url = WeatherBean.class.getResource(
34              "imagenames.properties" );
35
36           // load properties file contents
37           imageNames.load( new FileInputStream( url.getFile() ) );
38        }
39
40        // process exceptions from opening file
41        catch ( IOException ioException ) {
42           ioException.printStackTrace();
43        }
44
45     } // end static block
46
47     // WeatherBean constructor
48     public WeatherBean( String city, String weatherDescription,
49        String cityTemperature )
50     {
51        cityName = city;
52        temperature = cityTemperature;
53        description = weatherDescription.trim();
54
55        URL url = WeatherBean.class.getResource( "images/" +
56          imageNames.getProperty( description, "noinfo.jpg" ) );
57
58        // get weather image name or noinfo.jpg if weather
59        // description not found
60        image = new ImageIcon( url );
61     }
62
```

Fig. 13.3 **`WeatherBean`** stores weather forecast for one city (part 2 of 3).

```
63     // get city name
64     public String getCityName()
65     {
66        return cityName;
67     }
68
69     // get temperature
70     public String getTemperature()
71     {
72        return temperature;
73     }
74
75     // get weather description
76     public String getDescription()
77     {
78        return description;
79     }
80
81     // get weather image
82     public ImageIcon getImage()
83     {
84        return image;
85     }
86  }
```

Fig. 13.3 **`WeatherBean`** stores weather forecast for one city (part 3 of 3).

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.

```
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 {
```

Fig. 13.4 **`WeatherServiceClient`** client for **`WeatherService`** remote object (part 1 of 3).

```
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
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
```

Fig. 13.4 **WeatherServiceClient** client for **WeatherService** remote object (part 2 of 3).

```
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 3 of 3).

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`**.

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 **`WeatherService-`**

Client 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 **WeatherBean**s 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.

```
1  // WeatherListModel.java
2  // WeatherListModel extends AbstractListModel to provide a
3  // ListModel for storing a List of WeatherBeans.
4  package com.deitel.advjhtp1.rmi.weather;
5
6  // Java core packages
7  import java.util.*;
8
9  // Java extension packages
10 import javax.swing.AbstractListModel;
11
12 public class WeatherListModel extends AbstractListModel {
13
14    // List of elements in ListModel
15    private List list;
16
17    // no-argument WeatherListModel constructor
18    public WeatherListModel()
19    {
20       // create new List for WeatherBeans
21       list = new ArrayList();
22    }
23
```

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.

```
24     // WeatherListModel constructor
25     public WeatherListModel( List elementList )
26     {
27        list = elementList;
28     }
29
30     // get size of List
31     public int getSize()
32     {
33        return list.size();
34     }
35
36     // get Object reference to element at given index
37     public Object getElementAt( int index )
38     {
39        return list.get( index );
40     }
41
42     // add element to WeatherListModel
43     public void add( Object element )
44     {
45        list.add( element );
46        fireIntervalAdded( this, list.size(), list.size() );
47     }
48
49     // remove element from WeatherListModel
50     public void remove( Object element )
51     {
52        int index = list.indexOf( element );
53
54        if ( index != -1 ) {
55           list.remove( element );
56           fireIntervalRemoved( this, index, index );
57        }
58
59     } // end method remove
60
61     // remove all elements from WeatherListModel
62     public void clear()
63     {
64        // get original size of List
65        int size = list.size();
66
67        // clear all elements from List
68        list.clear();
69
70        // notify listeners that content changed
71        fireContentsChanged( this, 0, size );
72     }
73  }
```

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).

```
1  // WeatherCellRenderer.java
2  // WeatherCellRenderer is a custom ListCellRenderer for
3  // WeatherBeans in a JList.
4  package com.deitel.advjhtp1.rmi.weather;
5
6  // Java core packages
7  import java.awt.*;
8
9  // Java extension packages
10 import javax.swing.*;
11
12 public class WeatherCellRenderer extends DefaultListCellRenderer {
13
14    // returns a WeatherItem object that displays city's weather
15    public Component getListCellRendererComponent( JList list,
16       Object value, int index, boolean isSelected, boolean focus )
17    {
18       return new WeatherItem( ( WeatherBean ) value );
19    }
20 }
```

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

```
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).

```
13  public class WeatherItem extends JPanel {
14
15     private WeatherBean weatherBean;  // weather information
16
17     // background ImageIcon
18     private static ImageIcon backgroundImage;
19
20     // static initializer block loads background image when class
21     // WeatherItem is loaded into memory
22     static {
23
24        // get URL for background image
25        URL url = WeatherItem.class.getResource( "images/back.jpg" );
26
27        // background image for each city's weather info
28        backgroundImage = new ImageIcon( url );
29     }
30
31     // initialize a WeatherItem
32     public WeatherItem( WeatherBean bean )
33     {
34        weatherBean = bean;
35     }
36
37     // display information for city's weather
38     public void paintComponent( Graphics g )
39     {
40        super.paintComponent( g );
41
42        // draw background
43        backgroundImage.paintIcon( this, g, 0, 0 );
44
45        // set font and drawing color,
46        // then display city name and temperature
47        Font font = new Font( "SansSerif", Font.BOLD, 12 );
48        g.setFont( font );
49        g.setColor( Color.white );
50        g.drawString( weatherBean.getCityName(), 10, 19 );
51        g.drawString( weatherBean.getTemperature(), 130, 19 );
52
53        // display weather image
54        weatherBean.getImage().paintIcon( this, g, 253, 1 );
55
56     } // end method paintComponent
57
58     // make WeatherItem's preferred size the width and height of
59     // the background image
60     public Dimension getPreferredSize()
61     {
62        return new Dimension( backgroundImage.getIconWidth(),
63           backgroundImage.getIconHeight() );
64     }
65  }
```

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.

Fig. 13.8 Running the **rmiregistry**.

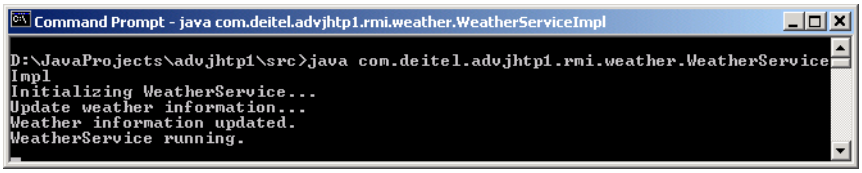

Fig. 13.9 Executing the **`WeatherServiceImpl`** remote object.

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.

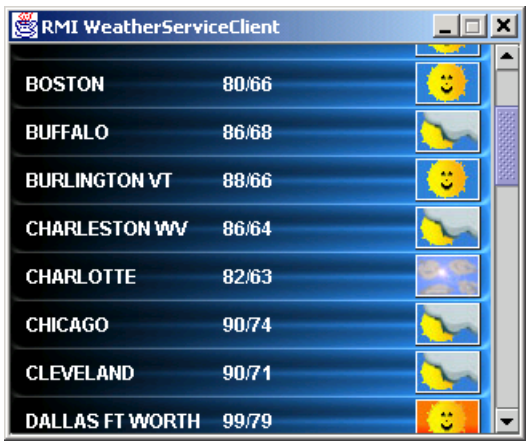

Fig. 13.10 **`WeatherServiceClient`** application window.