CHAPTER 43

REMOTE METHOD INVOCATION

Objectives

- To explain how RMI works (§43.2).
- To describe the process of developing RMI applications (§43.3).
- To distinguish between RMI and socket-level programming (§43.4).
- To develop three-tier applications using RMI (§43.5).
- To use callbacks to develop interactive applications (§43.6).

43.1 Introduction

Remote Method Invocation (RMI) provides a framework for building distributed Java systems. Using RMI, a Java object on one system can invoke a method in an object on another system on the network. A *distributed Java system* can be defined as a collection of cooperative distributed objects on the network. In this chapter, you will learn how to use RMI to create useful distributed applications.

43.2 RMI Basics

RMI is the Java Distributed Object Model for facilitating communications among distributed objects. RMI is a higher-level API built on top of sockets. Socket-level programming allows you to pass data through sockets among computers. RMI enables you also to invoke methods in a remote object. Remote objects can be manipulated as if they were residing on the local host. The transmission of data among different machines is handled by the JVM transparently.

client
server

In many ways, RMI is an evolution of the client/server architecture. A *client* is a component that issues requests for services, and a *server* is a component that delivers the requested services. Like the client/server architecture, RMI maintains the notion of clients and servers, but the RMI approach is more flexible.

- An RMI component can act as both a client and a server, depending on the scenario in question.
- An RMI system can pass functionality from a server to a client, and vice versa. Typically a client/server system only passes data back and forth between server and client.

43.2.1 How Does RMI Work?

local object

All the objects you have used before this chapter are called *local objects*. *Local objects* are accessible only within the local host. Objects that are accessible from a remote host are called *remote objects*.

remote object

For an object to be invoked remotely, it must be defined in a Java interface accessible to both the server and the client. Furthermore, the interface must extend the `java.rmi.Remote` interface. Like the `java.io.Serializable` interface, `java.rmi.Remote` is a marker interface that contains no constants or methods. It is used only to identify remote objects.

The key components of the RMI architecture are listed below (see Figure 43.1):

- **Server object interface:** A subinterface of `java.rmi.Remote` that defines the methods for the server object.
- **Server class:** A class that implements the remote object interface.
- **Server object:** An instance of the server class.
- **RMI registry:** A utility that registers remote objects and provides naming services for locating objects.
- **Client program:** A program that invokes the methods in the remote server object.
- **Server stub:** An object that resides on the client host and serves as a surrogate for the remote server object.
- **Server skeleton:** An object that resides on the server host and communicates with the stub and the actual server object.

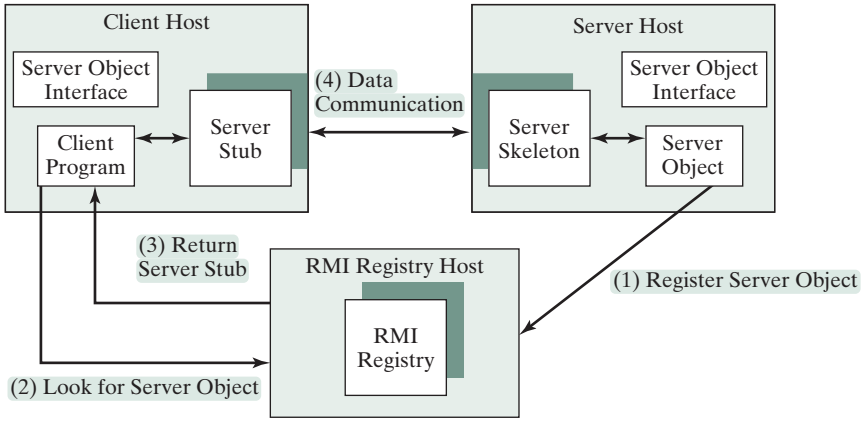


FIGURE 43.1 Java RMI uses a registry to provide naming services for remote objects, and uses the stub and the skeleton to facilitate communications between client and server.

RMI works as follows:

1. A server object is registered with the RMI registry.
2. A client looks through the RMI registry for the remote object.
3. Once the remote object is located, its stub is returned in the client.
4. The remote object can be used in the same way as a local object. Communication between the client and the server is handled through the stub and the skeleton.

The implementation of the RMI architecture is complex, but the good news is that RMI provides a mechanism that liberates you from writing the tedious code for handling parameter passing and invoking remote methods. The basic idea is to use two helper classes known as the *stub* and the *skeleton* for handling communications between client and server.

stub
skeleton

The stub and the skeleton are automatically generated. The *stub* resides on the client machine. It contains all the reference information the client needs to know about the server object. When a client invokes a method on a server object, it actually invokes a method that is encapsulated in the stub. The stub is responsible for sending parameters to the server and for receiving the result from the server and returning it to the client.

The *skeleton* communicates with the stub on the server side. The skeleton receives parameters from the client, passes them to the server for execution, and returns the result to the stub.

43.2.2 Passing Parameters

When a client invokes a remote method with parameters, passing the parameters is handled by the stub and the skeleton. Obviously, invoking methods in a remote object on a server is very different from invoking methods in a local object on a client, since the remote object is in a different address space on a separate machine. Let us consider three types of parameters:

- **Primitive data types**, such as `char`, `int`, `double`, or `boolean`, are passed by value like a local call.

primitive type

- **Local object types**, such as `java.lang.String`, are also passed by value, but this is completely different from passing an object parameter in a local call. In a local

local object

call, an object parameter's reference is passed, which corresponds to the memory address of the object. In a remote call, there is no way to pass the object reference, because the address on one machine is meaningless to a different JVM. Any object can be used as a parameter in a remote call as long as it is serializable. The stub serializes the object parameter and sends it in a stream across the network. The skeleton deserializes the stream into an object.

remote object

- **Remote object types** are passed differently from local objects. When a client invokes a remote method with a parameter of a remote object type, the stub of the remote object is passed. The server receives the stub and manipulates the parameter through it. Passing remote objects will be discussed in §43.6, "RMI Callbacks."

43.2.3 RMI Registry

How does a client locate the remote object? The RMI registry provides the registry services for the server to register the object and for the client to locate the object.

You can use several overloaded static `getRegistry()` methods in the `LocateRegistry` class to return a reference to a `Registry`, as shown in Figure 43.2. Once a `Registry` is obtained, you can bind an object with a unique name in the registry using the `bind` or `rebind` method or locate an object using the lookup method, as shown in Figure 43.3.

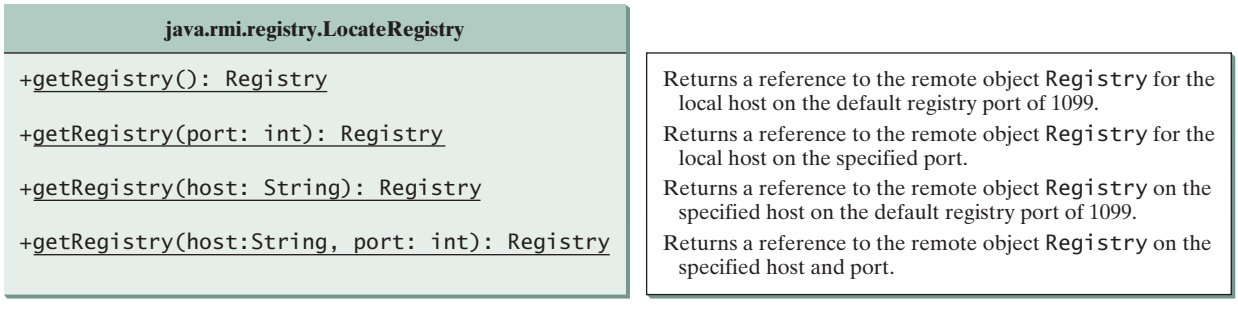


FIGURE 43.2 The `LocateRegistry` class provides the methods for obtaining a registry on a host.

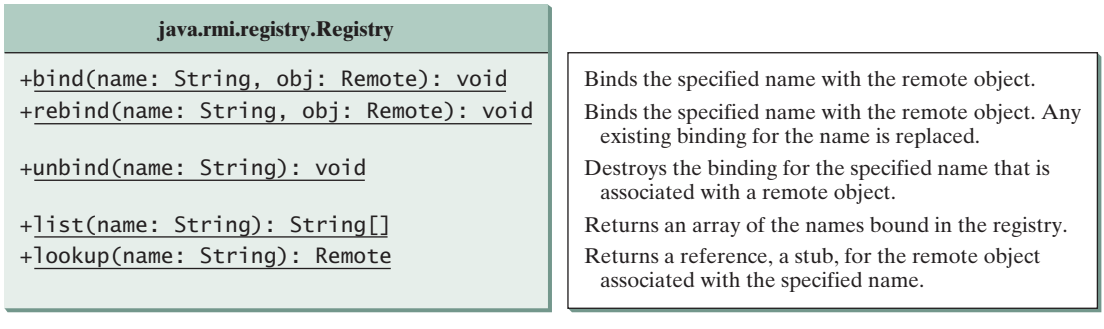


FIGURE 43.3 The `Registry` class provides the methods for binding and obtaining references to remote objects in a remote object registry.

43.3 Developing RMI Applications

Now that you have a basic understanding of RMI, you are ready to write simple RMI applications. The steps in developing an RMI application are shown in Figure 43.4 and listed below.

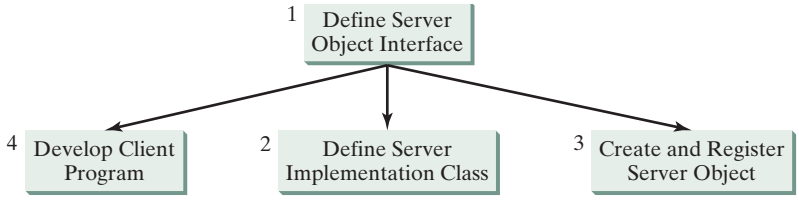


FIGURE 43.4 The steps in developing an RMI application.

1. *Define a server object interface* that serves as the contract between the server and its clients, as shown in the following outline:

```
public interface ServerInterface extends Remote {
  public void service1(...) throws RemoteException;
  // Other methods
}
```

   A server object interface must extend the `java.rmi.Remote` interface.

2. *Define a class that implements the server object interface*, as shown in the following outline:

```
public class ServerInterfaceImpl extends UnicastRemoteObject
    implements ServerInterface {
  public void service1(...) throws RemoteException {
    // Implement it
  }
  // Implement other methods
}
```

   The server implementation class must extend the `java.rmi.server.UnicastRemoteObject` class. The `UnicastRemoteObject` class provides support for point-to-point active object references using TCP streams.

3. *Create a server object* from the server implementation class and register it with an RMI registry:

```
ServerInterface server = new ServerInterfaceImpl(...);
Registry registry = LocateRegistry.getRegistry();
registry.rebind("RemoteObjectName", obj);
```

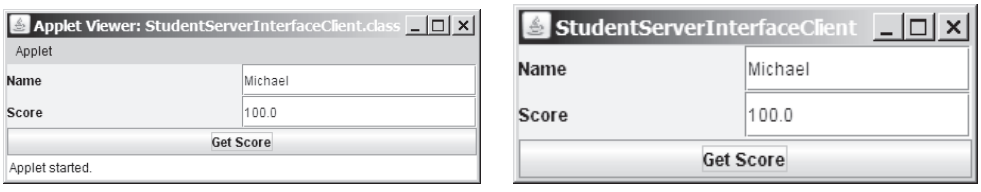
4. *Develop a client* that locates a remote object and invokes its methods, as shown in the following outline:

```
Registry registry = LocateRegistry.getRegistry(host);
ServerInterface server = (ServerInterfaceImpl)
  registry.lookup("RemoteObjectName");
server.service1(...);
```

The example that follows demonstrates the development of an RMI application through these steps.

43.3.1 Example: Retrieving Student Scores from an RMI Server

This example creates a client that retrieves student scores from an RMI server. The client, shown in Figure 43.5, displays the score for the specified name.



(a) Running as applet (b) Running as application

FIGURE 43.5 You can get the score by entering a student name and clicking the Get Score button.

1. Create a server interface named `StudentServerInterface` in Listing 43.1. The interface tells the client how to invoke the server's `findScore` method to retrieve a student score.

LISTING 43.1 StudentServerInterface.java

```
1 import java.rmi.*;
2
3 public interface StudentServerInterface extends Remote {
4   /**
5    * Return the score for the specified name
6    * @param   name   the student name
7    * @return  a double score or -1 if the student is not found
8    */
9   public double findScore(String name) throws RemoteException;
10 }
```

subinterface

server method

Any object that can be used remotely must be defined in an interface that extends the `java.rmi.Remote` interface (line 3). `StudentServerInterface`, extending `Remote`, defines the `findScore` method that can be remotely invoked by a client to find a student's score. Each method in this interface must declare that it may throw a `java.rmi.RemoteException` (line 9). Therefore your client code that invokes this method must be prepared to catch this exception in a try-catch block.

2. Create a server implementation named `StudentServerInterfaceImpl` (Listing 43.2) that implements `StudentServerInterface`. The `findScore` method returns the score for a specified student. It returns `-1` if the score is not found.

LISTING 43.2 StudentServerInterfaceImpl.java

```
1 import java.rmi.*;
2 import java.rmi.server.*;
3 import java.util.*;
4
5 public class StudentServerInterfaceImpl
6   extends UnicastRemoteObject
7     implements StudentServerInterface {
8   // Stores scores in a map indexed by name
```

```
 9   private HashMap<String, Double> scores =                          hash map
10     new HashMap<String, Double>();
11
12   public StudentServerInterfaceImpl() throws RemoteException {
13     initializeStudent();
14   }
15
16   /** Initialize student information */
17   protected void initializeStudent() {
18     scores.put("John", new Double(90.5));                           store score
19     scores.put("Michael", new Double(100));
20     scores.put("Michelle", new Double(98.5));
21   }
22
23   /** Implement the findScore method from the
24     Student interface */
25   public double findScore(String name) throws RemoteException {
26     Double d = (Double)scores.get(name);                            get score
27
28     if (d == null) {
29       System.out.println("Student " + name + " is not found ");
30       return -1;
31     }
32     else {
33       System.out.println("Student " + name + "\'s score is "
34         + d.doubleValue());
35       return d.doubleValue();
36     }
37   }
38 }
```

The `StudentServerInterfaceImpl` class implements `StudentServerInterface`. This class must also extend the `java.rmi.server.RemoteServer` class or its subclass. `RemoteServer` is an abstract class that defines the methods needed to create and export remote objects. Often its subclass `java.rmi.server.UnicastRemoteObject` is used (line 6). This subclass implements all the abstract methods defined in `RemoteServer`.

`StudentServerInterfaceImpl` implements the `findScore` method (lines 25–37) defined in `StudentServerInterface`. For simplicity, three students, John, Michael, and Michelle, and their corresponding scores are stored in an instance of `java.util.HashMap` named `scores`. `HashMap` is a concrete class of the `Map` interface in the Java Collections Framework, which makes it possible to search and retrieve a value using a key. Both values and keys are of `Object` type. The `findScore` method returns the score if the name is in the hash map, and returns `-1` if the name is not found.

3. Create a server object from the server implementation and register it with the RMI server (Listing 43.3).

LISTING 43.3 RegisterWithRMIServer.java

```
1 import java.rmi.registry.*;
2
3 public class RegisterWithRMIServer {
4   /** Main method */
5   public static void main(String[] args) {
6     try {
7       StudentServerInterface obj =                                  server object
8         new StudentServerInterfaceImpl();
9       Registry registry = LocateRegistry.getRegistry();             registry reference
```

register

```
10        registry.rebind("StudentServerInterfaceImpl", obj);
11        System.out.println("Student server " + obj + " registered");
12      }
13      catch (Exception ex) {
14        ex.printStackTrace();
15      }
16    }
17  }
```

`RegisterWithRMIServer` contains a main method, which is responsible for starting the server. It performs the following tasks: (1) create a server object (line 8); (2) obtain a reference to the RMI registry (line 9), and (3) register the object in the registry (line 10).

4. Create a client as an applet named `StudentServerInterfaceClient` in Listing 43.4. The client locates the server object from the RMI registry and uses it to find the scores.

LISTING 43.4 StudentServerInterfaceClient.java

```
1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.*;
4  import java.rmi.registry.LocateRegistry;
5  import java.rmi.registry.Registry;
6
7  public class StudentServerInterfaceClient extends JApplet {
8    // Declare a Student instance
```

remote object

```
9    private StudentServerInterface student;
10
```

standalone?

```
11   private boolean isStandalone; // Is applet or application
12
13   private JButton jbtGetScore = new JButton("Get Score");
14   private JTextField jtfName = new JTextField();
15   private JTextField jtfScore = new JTextField();
16
17   public void init() {
18     // Initialize RMI
```

initialize RMI

```
19     initializeRMI();
20
21     JPanel jPanel1 = new JPanel();
22     jPanel1.setLayout(new GridLayout(2, 2));
23     jPanel1.add(new JLabel("Name"));
24     jPanel1.add(jtfName);
25     jPanel1.add(new JLabel("Score"));
26     jPanel1.add(jtfScore);
27
28     add(jbtGetScore, BorderLayout.SOUTH);
29     add(jPanel1, BorderLayout.CENTER);
30
```

register listener

```
31     jbtGetScore.addActionListener(new ActionListener() {
32       public void actionPerformed(ActionEvent evt) {
```

get score

```
33         getScore();
34       }
35     });
36   }
37
38   private void getScore() {
39     try {
40       // Get student score
41       double score = student.findScore(jtfName.getText().trim());
42
```

```
43        // Display the result
44        if (score < 0)
45          jtfScore.setText("Not found");
46        else
47          jtfScore.setText(new Double(score).toString());
48      }
49      catch(Exception ex) {
50        ex.printStackTrace();
51      }
52    }
53
54    /** Initialize RMI */
55    protected void initializeRMI() {
56      String host = "";
57      if (!isStandalone) host = getCodeBase().getHost();
58
59      try {
60        Registry registry = LocateRegistry.getRegistry(host);
61        student = (StudentServerInterface)                          locate student
62          registry.lookup("StudentServerInterfaceImpl");
63        System.out.println("Server object " + student + " found");
64      }
65      catch(Exception ex) {
66        System.out.println(ex);
67      }
68    }
69
70    /** Main method */
71    public static void main(String[] args) {                        main method
72      StudentServerInterfaceClient applet =
73        new StudentServerInterfaceClient();
74      applet.isStandalone = true;                                   standalone
75      JFrame frame = new JFrame();
76      frame.setTitle("StudentServerInterfaceClient");
77      frame.add(applet, BorderLayout.CENTER);
78      frame.setSize(250, 150);
79      applet.init();
80      frame.setLocationRelativeTo(null);
81      frame.setVisible(true);
82      frame.setDefaultCloseOperation(3);
83    }
84  }
```

`StudentServerInterfaceClient` invokes the `findScore` method on the server to find the score for a specified student. The key method in `StudentServerInterfaceClient` is the `initializeRMI` method (lines 55–68), which is responsible for locating the server stub.

The `initializeRMI()` method treats standalone applications differently from applets. The host name should be the name where the applet is downloaded. It can be obtained using the `Applet`'s `getCodeBase().getHost()` (line 57). For standalone applications, the host name should be specified explicitly.

The `lookup(String name)` method (line 62) returns the remote object with the specified name. Once a remote object is found, it can be used just like a local object. The stub and the skeleton are used behind the scenes to make the remote method invocation work.

5. Follow the steps below to run this example.

   5.1. Start the RMI Registry by typing "**start rmiregistry**" at a DOS prompt from the book directory. By default, the port number `1099` is used by rmiregistry. To use a

different port number, simply type the command "**start rmiregistry** ***portnumber***" at a DOS prompt.

5.2. Start the server `RegisterWithRMIServer` using the following command at C:\book directory:

```
C:\book>java RegisterWithRMIServer
```

5.3. Run the client `StudentServerInterfaceClient` as an application. A sample run of the application is shown in Figure 43.5(b).

5.4. Run the client `StudentServerInterfaceClient.html` from the appletviewer. A sample run is shown in Figure 43.5(a).

Note

You must start rmiregistry from the directory where you will run the RMI server, as shown in Figure 43.6. Otherwise, you will receive the error `ClassNotFoundException` on `StudentServerInterfaceImpl_Stub`.

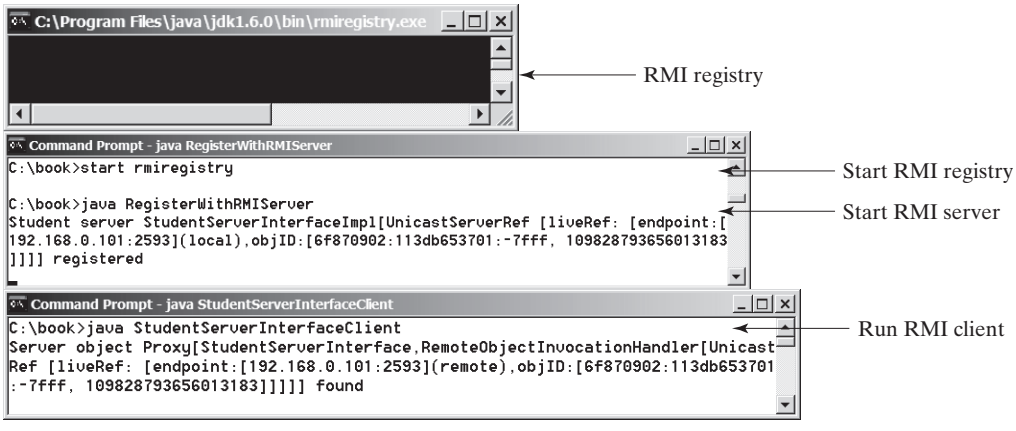


FIGURE 43.6 To run an RMI program, first start the RMIRegistry, then register the server object with the registry. The client locates it from the registry.

Note

Server, registry, and client can be on three different machines. If you run the client and the server on separate machines, you need to place `StudentServerInterface` on both machines. If you deploy the client as an applet, place all client files on the registry host.

Caution

If you modify the remote object implementation class, you need to restart the server class to reload the object to the RMI registry. In some old versions of rmiregistry, you may have to restart rmiregistry.

43.4 RMI vs. Socket-Level Programming

RMI enables you to program at a higher level of abstraction. It hides the details of socket server, socket, connection, and sending or receiving data. It even implements a multithreading server under the hood, whereas with socket-level programming you have to explicitly implement threads for handling multiple clients.

RMI applications are scalable and easy to maintain. You can change the RMI server or move it to another machine without modifying the client program except for resetting the URL to locate the server. (To avoid resetting the URL, you can modify the client to pass the

URL as a command-line parameter.) In socket-level programming, a client operation to send data requires a server operation to read it. The implementation of client and server at the socket level is tightly synchronized.

RMI clients can directly invoke the server method, whereas socket-level programming is limited to passing values. Socket-level programming is very primitive. Avoid using it to develop client/server applications. As an analogy, socket-level programming is like programming in assembly language, while RMI programming is like programming in a high-level language.

43.5 Developing Three-Tier Applications Using RMI

Three-tier applications have gained considerable attention in recent years, largely because of the demand for more scalable and load-balanced systems to replace traditional two-tier client/server database systems. A centralized database system does not just handle data access, it also processes the business rules on data. Thus, a centralized database is usually heavily loaded, because it requires extensive data manipulation and processing. In some situations, data processing is handled by the client and business rules are stored on the client side. It is preferable to use a middle tier as a buffer between client and database. The middle tier can be used to apply business logic and rules, and to process data to reduce the load on the database.

A three-tier architecture does more than just reduce the processing load on the server. It also provides access to multiple network sites. This is especially useful to Java applets that need to access multiple databases on different servers, since an applet can connect only with the server from which it is downloaded.

To demonstrate, let us rewrite the example in §43.3.1, "Example: Retrieving Student Scores from an RMI Server," to find scores stored in a database rather than a hash map. In addition, the system is capable of blocking a client from accessing a student who has not given the university permission to publish his/her score. An RMI component is developed to serve as a middle tier between client and database; it sends a search request to the database, processes the result, and returns an appropriate value to the client.

For simplicity, this example reuses the `StudentServerInterface` interface and `StudentServerInterfaceClient` class from §43.3.1 with no modifications. All you have to do is to provide a new implementation for the server interface and create a program to register the server with the RMI. Here are the steps to complete the program:

1. Store the scores in a database table named `Score` that contains three columns: `name`, `score`, and `permission`. The permission value is `1` or `0`, which indicates whether the student has given the university permission to release his/her grade. The following is the statement to create the table and insert three records:

```
create table Scores (name varchar(20),
  score number, permission number);

insert into Scores values ('John', 90.5, 1);
insert into Scores values ('Michael', 100, 1);
insert into Scores values ('Michelle', 100, 0);
```

2. Create a new server implementation named `Student3TierImpl` in Listing 43.5. The server retrieves a record from the `Scores` table, processes the retrieved information, and sends the result back to the client.

LISTING 43.5 Student3TierImpl.java

```
1 import java.rmi.*;
2 import java.rmi.server.*;
3 import java.sql.*;
4
5 public class Student3TierImpl extends UnicastRemoteObject
```

```
    implements StudentServerInterface {
  // Use prepared statement for querying DB
  private PreparedStatement pstmt;

  /** Constructs Student3TierImpl object and exports it on
   * default port.
   */
  public Student3TierImpl() throws RemoteException {
    initializeDB();
  }

  /** Constructs Student3TierImpl object and exports it on
   * specified port.
   * @param port The port for exporting
   */
  public Student3TierImpl(int port) throws RemoteException {
    super(port);
    initializeDB();
  }

  /** Load JDBC driver, establish connection and
    create statement */
  protected void initializeDB() {
    try {
      // Load the JDBC driver
      // Class.forName("oracle.jdbc.driver.OracleDriver");
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      System.out.println("Driver registered");

      // Establish connection
      /*Connection conn = DriverManager.getConnection
        ("jdbc:oracle:thin:@drake.armstrong.edu:1521:orcl",
          "scott", "tiger");  */
      Connection conn = DriverManager.getConnection
        ("jdbc:odbc:exampleMDBDataSource", "", "" );
      System.out.println("Database connected");

      // Create a prepared statement for querying DB
      pstmt = conn.prepareStatement(
        "select * from Scores where name = ?");
    }
    catch (Exception ex) {
      System.out.println(ex);
    }
  }

  /** Return the score for specified the name
   * Return -1 if score is not found.
   */
  public double findScore(String name) throws RemoteException {
    double score = -1;
    try {
      // Set the specified name in the prepared statement
      pstmt.setString(1, name);

      // Execute the prepared statement
      ResultSet rs = pstmt.executeQuery();
```

initialize db

load driver

connect db

prepare statement

set name

execute SQL

```
64        // Retrieve the score
65        if (rs.next()) {
66          if (rs.getBoolean(3))
67            score = rs.getDouble(2);                    get score
68        }
69      }
70      catch (SQLException ex) {
71        System.out.println(ex);
72      }
73
74      System.out.println(name + "\'s score is " + score);
75      return score;
76    }
77  }
```

`Student3TierImpl` is similar to StudentServerInterfaceImpl in §43.3.1 except that the `Student3TierImpl` class finds the score from a JDBC data source instead from a hash map.

The table named `Scores` consists of three columns, `name`, `score`, and `permission`, where the latter indicates whether the student has given permission to show his/her score. Since SQL does not support a `boolean` type, permission is defined as a number whose value of `1` indicates `true` and of `0` indicates `false`.

The `initializeDB()` method (lines 28–50) loads the appropriate JDBC driver, establishes connections with the database, and creates a prepared statement for processing the query.

The `findScore` method (lines 55–76) sets the name in the prepared statement, executes the statement, processes the result, and returns the score for a student whose permission is `true`.

3. Write a `main` method in the class `RegisterStudent3TierServer` (Listing 43.6) that registers the server object using StudentServerInterfaceImpl, the same name as in Listing 43.2, so that you can use `StudentServerInterfaceClient`, created in §43.3.1, to test the server.

LISTING 43.6 RegisterStudent3TierServer.java

```
1  import java.rmi.registry.*;
2
3  public class RegisterStudent3TierServer {
4    public static void main(String[] args) {
5      try {
6        StudentServerInterface obj = new Student3TierImpl();
7        Registry registry = LocateRegistry.getRegistry();      registry on local host
8        registry.rebind("StudentServerInterfaceImpl", obj);    register server object
9        System.out.println
10         ("Student server " + obj + " registered");
11     } catch (Exception ex) {
12       ex.printStackTrace();
13     }
14   }
15 }
```

4. Follow the steps below to run this example.

   4.1. Start RMI Registry by typing "**start rmiregistry**" at a DOS prompt from the book directory.

4.2. Start the server `RegisterStudent3TierServer` using the following command at the C:\book directory:

```
C:\book>java RegisterStudent3TierServer
```

4.3. Run the client `StudentServerInterfaceClient` as an application or applet.

43.6 RMI Callbacks

In a traditional client/server system, a client sends a request to a server, and the server processes the request and returns the result to the client. The server cannot invoke the methods on a client. One important benefit of RMI is that it supports *callbacks*, which enable the server to invoke methods on the client. With the RMI callback feature, you can develop interactive distributed applications.

In §30.9, "Case Studies: Distributed TicTacToe Games," you developed a distributed TicTacToe game using stream socket programming. The example that follows demonstrates the use of the RMI callback feature to develop an interactive TicTacToe game.

All the examples you have seen so far in this chapter have simple behaviors that are easy to model with classes. The behavior of the TicTacToe game is somewhat complex. To create the classes to model the game, you need to study and understand it and distribute the process appropriately between client and server.

Clearly the client should be responsible for handling user interactions, and the server should coordinate with the client. Specifically, the client should register with the server, and the server can take two and only two players. Once a client makes a move, it should notify the server; the server then notifies the move to the other player. The server should determine the status of the game—that is, whether it has been won or drawn—and notify the players. The server should also coordinate the turns—that is, which client has the turn at a given time. The ideal approach for notifying a player is to invoke a method in the client that sets appropriate properties in the client or sends messages to a player. Figure 43.7 illustrates the relationship between clients and server.

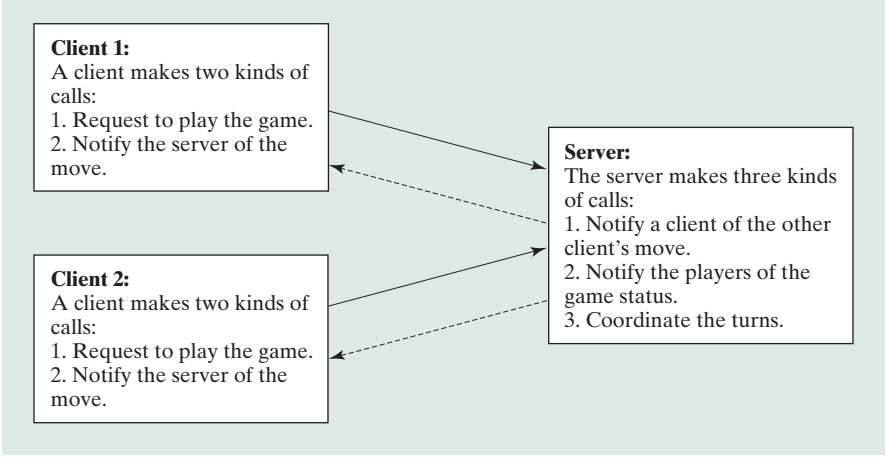


FIGURE 43.7 The server coordinates the activities with the clients.

All the calls a client makes can be encapsulated in one remote interface named `TicTacToe` (Listing 43.7), and all the calls the server invokes can be defined in another interface named `CallBack` (Listing 43.8). These two interfaces are defined as follows:

LISTING 43.7 TicTacToeInterface.java

```
1  import java.rmi.*;
2
3  public interface TicTacToeInterface extends Remote {                    subinterface
4    /**
5     * Connect to the TicTacToe server and return the token.
6     * If the returned token is ' ', the client is not connected to
7     * the server
8     */
9    public char connect(CallBack client) throws RemoteException;           server method
10
11   /** A client invokes this method to notify the server of its move*/
12   public void myMove(int row, int column, char token)                    server method
13     throws RemoteException;
14 }
```

LISTING 43.8 CallBack.java

```
1  import java.rmi.*;
2
3  public interface CallBack extends Remote {                               subinterface
4    /** The server notifies the client for taking a turn */
5    public void takeTurn(boolean turn) throws RemoteException;             server method
6
7    /** The server sends a message to be displayed by the client */
8    public void notify(java.lang.String message)                           server method
9      throws RemoteException;
10
11   /** The server notifies a client of the other player's move */
12   public void mark(int row, int column, char token)                      server method
13     throws RemoteException;
14 }
```

What does a client need to do? The client interacts with the player. Assume that all the cells are initially empty, and that the first player takes the X token and the second player the O token. To mark a cell, the player points the mouse to the cell and clicks it. If the cell is empty, the token (X or O) is displayed. If the cell is already filled, the player's action is ignored.

From the preceding description, it is obvious that a cell is a GUI object that handles mouse-click events and displays tokens. The candidate for such an object could be a button or a panel. Panels are more flexible than buttons. The token (X or O) can be drawn on a panel in any size, but it can be displayed only as a label on a button.

Let `Cell` be a subclass of `JPanel`. You can declare a 3 × 3 grid to be an array `Cell[][] cell = new Cell[3][3]` for modeling the game. How do you know the state of a cell (marked or not)? You can use a property named `marked` of the `boolean` type in the `Cell` class. How do you know whether the player has a turn? You can use a property named `myTurn` of `boolean`. This property (initially `false`) can be set by the server through a callback.

The `Cell` class is responsible for drawing the token when an empty cell is clicked, so you need to write the code for listening to the `MouseEvent` and for painting the shape for tokens X and O. To determine which shape to draw, introduce a variable named `marker` of the `char` type. Since this variable is shared by all the cells in a client, it is preferable to declare it in the client and to declare the `Cell` class as an inner class of the client so that this variable will be accessible to all the cells.

Now let us turn our attention to the server side. What does the server need to do? The server needs to implement `TicTacToeInterface` and notify the clients of the game status. The server has to record the moves in the cells and check the status every time a player makes a move. The status information can be kept in a 3 × 3 array of `char`. You can implement a

method named `isFull()` to check whether the board is full and a method named `isWon(token)` to check whether a specific player has won.

Once a client is connected to the server, the server notifies the client which token to use—that is, X for the first client and O for the second. Once a client notifies the server of its move, the server checks the game status and notifies the clients.

Now the most critical question is how the server notifies a client. You know that a client invokes a server method by creating a server stub on the client side. A server cannot directly invoke a client, because the client is not declared as a remote object. The `CallBack` interface was created to facilitate the server's callback to the client. In the implementation of `CallBack`, an instance of the client is passed as a parameter in the constructor of `CallBack`. The client creates an instance of `CallBack` and passes its stub to the server, using a remote method named `connect()` defined in the server. The server then invokes the client's method through a `CallBack` instance. The triangular relationship of client, `CallBack` implementation, and server is shown in Figure 43.8.

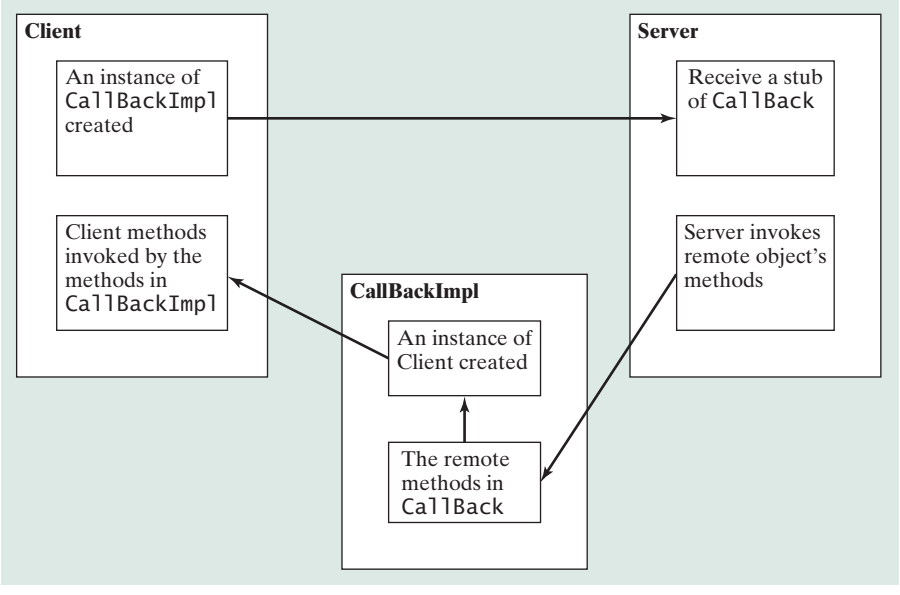


FIGURE 43.8 The server receives a `CallBack` stub from the client and invokes the remote methods defined in the `CallBack` interface, which can invoke the methods defined in the client.

Here are the steps to complete the example.

1. Create TicTacToeImpl.java (Listing 43.9) to implement `TicTacToeInterface`. Add a main method in the program to register the server with the RMI.

LISTING 43.9 TicTacToeImpl.java

```
1 import java.rmi.*;
2 import java.rmi.server.*;
3 import java.rmi.registry.*;
4 import java.rmi.registry.*;
5
6 public class TicTacToeImpl extends UnicastRemoteObject
7     implements TicTacToeInterface {
8   // Declare two players, used to call players back
9   private CallBack player1 = null;
```

call back objects

```
10  private CallBack player2 = null;
11
12  // board records players' moves
13  private char[][] board = new char[3][3];
14
15  /** Constructs TicTacToeImpl object and
16    exports it on default port.
17   */
18  public TicTacToeImpl() throws RemoteException {
19    super();
20  }
21
22  /** Constructs TicTacToeImpl object and exports it on specified
23   * port.
24   * @param port The port for exporting
25   */
26  public TicTacToeImpl(int port) throws RemoteException {
27    super(port);
28  }
29
30  /**
31   * Connect to the TicTacToe server and return the token.
32   * If the returned token is ' ', the client is not connected to
33   * the server
34   */
35  public char connect(CallBack client) throws RemoteException {          implement connect
36    if (player1 == null) {
37      // player1 (first player) registered
38      player1 = client;
39      player1.notify("Wait for a second player to join");
40      return 'X';
41    }
42    else if (player2 == null) {
43      // player2 (second player) registered
44      player2 = client;
45      player2.notify("Wait for the first player to move");
46      player2.takeTurn(false);
47      player1.notify("It is my turn (X token)");
48      player1.takeTurn(true);
49      return 'O';
50    }
51    else {
52      // Already two players
53      client.notify("Two players are already in the game");
54      return ' ';
55    }
56  }
57
58  /** A client invokes this method to notify the
59    server of its move*/
60  public void myMove(int row, int column, char token)                    implement myMove
61      throws RemoteException {
62    // Set token to the specified cell
63    board[row][column] = token;
64
65    // Notify the other player of the move
66    if (token == 'X')
67      player2.mark(row, column, 'X');
68    else
69      player1.mark(row, column, 'O');
```

```
70
71      // Check if the player with this token wins
72      if (isWon(token)) {
73        if (token == 'X') {
74          player1.notify("I won!");
75          player2.notify("I lost!");
76          player1.takeTurn(false);
77        }
78        else {
79          player2.notify("I won!");
80          player1.notify("I lost!");
81          player2.takeTurn(false);
82        }
83      }
84      else if (isFull()) {
85        player1.notify("Draw!");
86        player2.notify("Draw!");
87      }
88      else if (token == 'X') {
89        player1.notify("Wait for the second player to move");
90        player1.takeTurn(false);
91        player2.notify("It is my turn, (O token)");
92        player2.takeTurn(true);
93      }
94      else if (token == 'O') {
95        player2.notify("Wait for the first player to move");
96        player2.takeTurn(false);
97        player1.notify("It is my turn, (X token)");
98        player1.takeTurn(true);
99      }
100   }
101
102   /** Check if a player with the specified token wins */
isWon 103   public boolean isWon(char token) {
104     for (int i = 0; i < 3; i++)
105       if ((board[i][0] == token) && (board[i][1] == token)
106         && (board[i][2] == token))
107         return true;
108
109     for (int j = 0; j < 3; j++)
110       if ((board[0][j] == token) && (board[1][j] == token)
111         && (board[2][j] == token))
112         return true;
113
114     if ((board[0][0] == token) && (board[1][1] == token)
115       && (board[2][2] == token))
116       return true;
117
118     if ((board[0][2] == token) && (board[1][1] == token)
119       && (board[2][0] == token))
120       return true;
121
122     return false;
123   }
124
125   /** Check if the board is full */
isFull 126   public boolean isFull() {
127     for (int i = 0; i < 3; i++)
128       for (int j = 0; j < 3; j++)
129         if (board[i][j] == '\u0000')
```

```
130           return false;
131
132       return true;
133     }
134
135     public static void main(String[] args) {
136       try {
137         TicTacToeInterface obj = new TicTacToeImpl();
138         Registry registry = LocateRegistry.getRegistry();
139         registry.rebind("TicTacToeImpl", obj);                    register object
140         System.out.println("Server " + obj + " registered");
141       }
142       catch (Exception ex) {
143         ex.printStackTrace();
144       }
145     }
146 }
```

2. Create CallBackImpl.java (Listing 43.10) to implement the `CallBack` interface.

LISTING 43.10 CallBackImpl.java

```
1  import java.rmi.*;
2  import java.rmi.server.*;
3
4  public class CallBackImpl extends UnicastRemoteObject
5      implements CallBack {
6    // The client will be called by the server through callback
7    private TicTacToeClientRMI thisClient;
8
9    /** Constructor */
10   public CallBackImpl(Object client) throws RemoteException {
11     thisClient = (TicTacToeClientRMI)client;
12   }
13
14   /** The server notifies the client for taking a turn */
15   public void takeTurn(boolean turn) throws RemoteException {      implement
16     thisClient.setMyTurn(turn);
17   }
18
19   /** The server sends a message to be displayed by the client */
20   public void notify(String message) throws RemoteException {      implement
21     thisClient.setMessage(message);
22   }
23
24   /** The server notifies a client of the other player's move */
25   public void mark(int row, int column, char token)                 implement
26       throws RemoteException {
27     thisClient.mark(row, column, token);
28   }
29 }
```

3. Create an applet `TicTacToeClientRMI` (Listing 43.11) for interacting with a player and communicating with the server. Enable it to run standalone.

LISTING 43.11 TicTacToeClientRMI.java

```
1 import java.rmi.*;
2 import java.awt.*;
```

```
 3 import java.awt.event.*;
 4 import javax.swing.*;
 5 import javax.swing.border.*;
 6 import java.rmi.registry.Registry;
 7 import java.rmi.registry.LocateRegistry;
 8
 9 public class TicTacToeClientRMI extends JApplet {
10   // marker is used to indicate the token type
11   private char marker;
12
13   // myTurn indicates whether the player can move now
14   private boolean myTurn = false;
15
16   // Each cell can be empty or marked as 'O' or 'X'
17   private Cell[][] cell;
18
19   // ticTacToe is the game server for
20   // coordinating with the players
21   private TicTacToeInterface ticTacToe;
22
23   // Border for cells and panel
24   private Border lineBorder =
25     BorderFactory.createLineBorder(Color.yellow, 1);
26
27   private JLabel jlblStatus = new JLabel("jLabel1");
28   private JLabel jlblIdentification = new JLabel();
29
30   boolean isStandalone = false;
31
32   /** Initialize the applet */
33   public void init() {
34     JPanel jPanel1 = new JPanel();
35     jPanel1.setBorder(lineBorder);
36     jPanel1.setLayout(new GridLayout(3, 3, 1, 1));
37
38     add(jlblStatus, BorderLayout.SOUTH);
39     add(jPanel1, BorderLayout.CENTER);
40     add(jlblIdentification, BorderLayout.NORTH);
41
42     // Create cells and place cells in the panel
43     cell = new Cell[3][3];
44     for (int i = 0; i < 3; i++)
45       for (int j = 0; j < 3; j++)
46         jPanel1.add(cell[i][j] = new Cell(i, j));
47
48     try {
49       initializeRMI();
50     }
51     catch (Exception ex) {
52       ex.printStackTrace();
53     }
54   }
55
56   /** Initialize RMI */
57   protected boolean initializeRMI() throws Exception {
58     String host = "";
59     if (!isStandalone) host = getCodeBase().getHost();
60
61     try {
62       Registry registry = LocateRegistry.getRegistry(host);
```

server object

create UI

registry host

```
63        ticTacToe = (TicTacToeInterface)                                  server object
64          registry.lookup("TicTacToeImpl");
65        System.out.println
66          ("Server object " + ticTacToe + " found");
67      }
68      catch (Exception ex) {
69        System.out.println(ex);
70      }
71
72      // Create callback for use by the
73      // server to control the client
74      CallBackImpl callBackControl = new CallBackImpl(this);            call back
75
76      if (
77        (marker =
78          ticTacToe.connect((CallBack)callBackControl)) != ' ')
79      {
80        System.out.println("connected as " + marker + " player.");
81        jlblIdentification.setText("You are player " + marker);
82        return true;
83      }
84      else {
85        System.out.println("already two players connected as ");
86        return false;
87      }
88    }
89
90    /** Set variable myTurn to true or false */
91    public void setMyTurn(boolean myTurn) {
92      this.myTurn = myTurn;
93    }
94
95    /** Set message on the status label */
96    public void setMessage(String message) {
97      jlblStatus.setText(message);
98    }
99
100   /** Mark the specified cell using the token */
101   public void mark(int row, int column, char token) {
102     cell[row][column].setToken(token);
103   }
104
105   /** Inner class Cell for
106     modeling a cell on the TicTacToe board */
107   private class Cell extends JPanel {
108     // marked indicates whether the cell has been used
109     private boolean marked = false;
110
111     // row and column indicate where the cell
112     // appears on the board
113     int row, column;
114
115     // The token for the cell
116     private char token;
117
118     /** Construct a cell */
119     public Cell(final int row, final int column) {
120       this.row = row;
121       this.column = column;
122       addMouseListener(new MouseAdapter() {                             register listener
```

```
123         public void mouseClicked(MouseEvent e) {
124           if (myTurn && !marked) {
125             // Mark the cell
126             setToken(marker);
127
128             // Notify the server of the move
129             try {
130               ticTacToe.myMove(row, column, marker);
131             }
132             catch (RemoteException ex) {
133               System.out.println(ex);
134             }
135           }
136         }
137       });
138
139       setBorder(lineBorder);
140     }
141
142     /** Set token on a cell (mark a cell) */
143     public void setToken(char c) {
144       token = c;
145       marked = true;
146       repaint();
147     }
148
149     /** Paint the cell to draw a shape for the token */
150     protected void paintComponent(Graphics g) {
151       super.paintComponent(g);
152
153       // Draw the border
154       g.drawRect(0, 0, getSize().width, getSize().height);
155
156       if (token == 'X') {
157         g.drawLine(10, 10, getSize().width - 10,
158           getSize().height - 10);
159         g.drawLine(getSize().width - 10, 10, 10,
160           getSize().height - 10);
161       }
162       else if (token == 'O') {
163         g.drawOval(10, 10, getSize().width - 20,
164           getSize().height - 20);
165       }
166     }
167   }
168
169   /** Main method */
170   public static void main(String[] args) {
171     TicTacToeClientRMI applet = new TicTacToeClientRMI();
172     applet.isStandalone = true;
173     applet.init();
174     applet.start();
175     JFrame frame = new JFrame();
176     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
177     frame.setTitle("TicTacToeClientRMI");
178     frame.add(applet, BorderLayout.CENTER);
179     frame.setSize(400, 320);
180     frame.setVisible(true);
181   }
182 }
```

standalone

4. Follow the steps below to run this example.

   4.1. Start RMI Registry by typing "**start rmiregistry**" at a DOS prompt from the book directory.

   4.2. Start the server `TicTacToeImpl` using the following command at the C:\book directory:

   ```
   C:\book>java TicTacToeImpl
   ```

   4.3. Run the client `TicTacToeClientRMI` as an application or an applet. A sample run is shown in Figure 43.9.

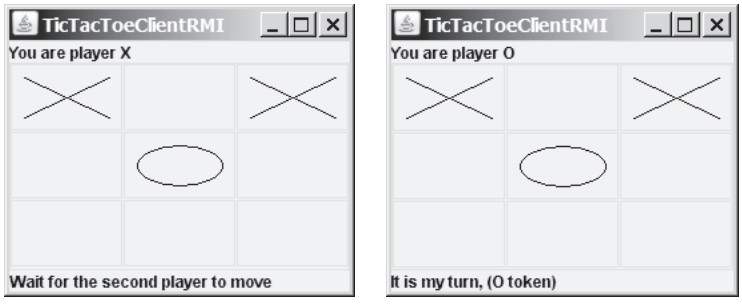


FIGURE 43.9 Two players play each other through the RMI server.

`TicTacToeInterface` defines two remote methods, `connect(CallBack client)` and `myMove(int row, int column, char token)`. The `connect` method plays two roles: one is to pass a `CallBack` stub to the server, and the other is to let the server assign a token for the player. The `myMove` method notifies the server that the player has made a specific move.

The `CallBack` interface defines three remote methods, `takeTurn(boolean turn)`, `notify(String message)`, and `mark(int row, int column, char token)`. The `takeTurn` method sets the client's `myTurn` property to `true` or `false`. The `notify` method displays a message on the client's status label. The `mark` method marks the client's cell with the token at the specified location.

`TicTacToeImpl` is a server implementation for coordinating with the clients and managing the game. The variables `player1` and `player2` are instances of `CallBack`, each of which corresponds to a client, passed from a client when the client invokes the `connect` method. The variable `board` records the moves by the two players. This information is needed to determine the game status. When a client invokes the `connect` method, the server assigns a token X for the first player and O for the second player, and accepts only two players. You can modify the program to accept additional clients as observers. See Exercise 43.7 for more details.

Once two players are in the game, the server coordinates the turns between them. When a client invokes the `myMove` method, the server records the move and notifies the other player by marking the other player's cell. It then checks to see whether the player wins or whether the board is full. If neither condition applies and therefore the game continues, the server gives a turn to the other player.

The `CallBackImpl` implements the `CallBack` interface. It creates an instance of `TicTacToeClientRMI` through its constructor. The `CallBackImpl` relays the server request to the client by invoking the client's methods. When the server invokes the `takeTurn` method, `CallBackImpl` invokes the client's `setMyTurn()` method to set the property `myTurn` in the client. When the server invokes the `notify()` method, `CallBackImpl` invokes the client's `setMessage()` method to set the message on the client's status label. When the server invokes the `mark` method, `CallBackImpl` invokes the client's `mark` method to mark the specified cell.

`TicTacToeClientRMI` can run as a standalone application or as an applet. The `initializeRMI` method is responsible for creating the URL for running as a standalone application or as an applet, for locating the `TicTacToeImpl` server stub, for creating the `CallBack` server object, and for connecting the client with the server.

Interestingly, obtaining the `TicTacToeImpl` stub for the client is different from obtaining the `CallBack` stub for the server. The `TicTacToeImpl` stub is obtained by invoking the `lookup()` method through the RMI registry, and the `CallBack` stub is passed to the server through the `connect` method in the `TicTacToeImpl` stub. It is a common practice to obtain the first stub with the `lookup` method, but to pass the subsequent stubs as parameters through remote method invocations.

Since the variables `myTurn` and `marker` are defined in `TicTacToeClientRMI`, the `Cell` class is defined as an inner class within `TicTacToeClientRMI` in order to enable all the cells in the client to access them. Exercise 43.8 suggests alternative approaches that implement the `Cell` as a noninner class.

Key Terms

callback 43–14
RMI registry 43–2
skeleton 43–3
stub 43–3

Chapter Summary

1. RMI is a high-level Java API for building distributed applications using distributed objects.
2. The key idea of RMI is its use of stubs and skeletons to facilitate communications between objects. The stub and skeleton are automatically generated, which relieves programmers of tedious socket-level network programming.
3. For an object to be used remotely, it must be defined in an interface that extends the `java.rmi.Remote` interface.
4. In an RMI application, the initial remote object must be registered with the RMI registry on the server side and be obtained using the `lookup` method through the registry on the client side. Subsequent uses of stubs of other remote objects may be passed as parameters through remote method invocations.
5. RMI is especially useful for developing scalable and load-balanced multitier distributed applications.

Review Questions

Sections 43.2–43.3

43.1 How do you define an interface for a remote object?

43.2 Describe the roles of the stub and the skeleton.

43.3 What is `java.rmi.Remote`? How do you define a server class?

43.4 What is an RMI registry for? How do you create an RMI registry?

43.5 What is the command to start an RMI Registry?

43.6 How do you register a remote object with the RMI registry?

43.7 What is the command to start a custom RMI server?

43.8 How does a client locate a remote object stub through an RMI registry?

43.9 How do you obtain a registry? How do you register a remote object? How do you locate remote object?

Sections 43.4–43.6

43.10 What are the advantages of RMI over socket-level programming?

43.11 Describe how parameters are passed in RMI.

43.12 What is the problem if the `connect` method in the `TicTacToeInterface` is defined as

```
public boolean connect(CallBack client, char token)
  throws RemoteException;
```

or as

```
public boolean connect(CallBack client, Character token)
  throws RemoteException;
```

43.13 What is callback? How does callback work in RMI?

Programming Exercises

Section 43.3

43.1* (*Limiting the number of clients*) Modify the example in §43.3.1, "Example: Retrieving Student Scores from an RMI Server," to limit the number of concurrent clients to ten.

43.2* (*Computing loan*) Rewrite Exercise 30.1 using RMI. You need to define a remote interface for computing monthly payment and total payment.

43.3** (*Web visit count*) Rewrite Exercise 30.4 using RMI. You need to define a remote interface for obtaining and increasing the count.

43.4** (*Displaying and adding addresses*) Rewrite Exercise 30.6 using RMI. You need to define a remote interface for adding addresses and retrieving address information.

Section 43.5

43.5** (*Address in a database table*) Rewrite Exercise 43.4. Assume that the address is stored in a table.

43.6** (*Three-tier application*) Use the three-tier approach to modify Exercise 43.4, as follows:

- Create an applet client to manipulate student information, as shown in Figure 30.23(a).
- Create a remote object interface with methods for retrieving, inserting, and updating student information, and an object implementation for the interface.

Section 43.6

43.7** (*Chat*) Rewrite Exercise 30.13 using RMI. You need to define a remote interface for sending and receiving a message.

43.8** (*Improving TicTacToe*) Modify the TicTacToe example in §43.6, "RMI Callbacks," as follows:

- Allow a client to connect to the server as an observer to watch the game.
- Rewrite the `Cell` class as a noninner class.