CHAPTER

40

Remote Method Invocation

Objectives

- To explain how RMI works (§40.2).
- To describe the process of developing RMI applications (§40.3).
- To distinguish between RMI and socket-level programming (§40.4).
- To develop three-tier applications using RMI (§40.5).
- To use callbacks to develop interactive applications (§40.6).

40.1 Introduction

Remote Method Invocation is a high-level Java API for Java network programming.

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.

40.2 RMI Basics

RMI enables you to access a remote object and invoke its methods.

RMI is the Java Distributed Object Model for facilitating communications among distributed objects. RMI is a high-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.

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.

40.2.1 How Does RMI Work?

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*. 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 40.1):

- **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.

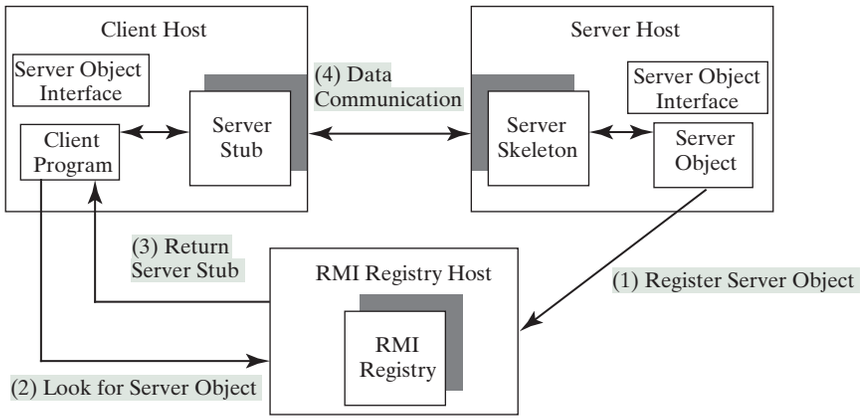

FIGURE 40.1 Java RMI uses a registry to provide naming services for remote objects, and uses the stub and the skeleton to facilitate communications between client and server.

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.

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.

40.2.2 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.
- **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 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 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 Section 40.6, "RMI Callbacks."

40.2.3 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 40.2. 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 40.3.

java.rmi.registry.LocateRegistry	
`+getRegistry(): Registry`	Returns a reference to the remote object Registry for the local host on the default registry port of 1099.
`+getRegistry(port: int): Registry`	Returns a reference to the remote object Registry for the local host on the specified port.
`+getRegistry(host: String): Registry`	Returns a reference to the remote object Registry on the specified host on the default registry port of 1099.
`+getRegistry(host:String, port: int): Registry`	Returns a reference to the remote object Registry on the specified host and port.

FIGURE 40.2 The `LocateRegistry` class provides the methods for obtaining a registry on a host.

java.rmi.registry.Registry	
`+bind(name: String, obj: Remote): void`	Binds the specified name with the remote object.
`+rebind(name: String, obj: Remote): void`	Binds the specified name with the remote object. Any existing binding for the name is replaced.
`+unbind(name: String): void`	Destroys the binding for the specified name that is associated with a remote object.
`+list(name: String): String[]`	Returns an array of the names bound in the registry.
`+lookup(name: String): Remote`	Returns a reference, a stub, for the remote object associated with the specified name.

FIGURE 40.3 The `Registry` class provides the methods for binding and obtaining references to remote objects in a remote object registry.

40.3 Developing RMI Applications

An RMI application consists of defining server object interface, defining a server object interface implementation class, creating and registering a server object, and developing a client program.

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 40.4 and listed below.

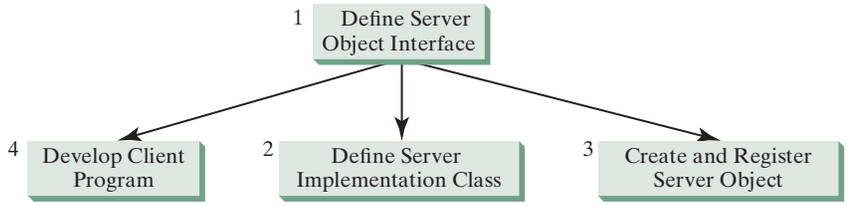

FIGURE 40.4 The steps in developing an RMI application.

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", server);
```

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.

40.3.1 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 40.5, displays the score for the specified name.

1. Create a server interface named `StudentServerInterface` in Listing 40.1. The interface tells the client how to invoke the server's `findScore` method to retrieve a student score.

FIGURE 40.5 You can get the score by entering a student name and clicking the Get Score button.

LISTING 40.1 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 }
```

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 40.2) that implements `StudentServerInterface`. The `findScore` method returns the score for a specified student. It returns `-1` if the score is not found.

LISTING 40.2 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 =
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));
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);
```

```
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 40.3).

LISTING 40.3 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 =
8          new StudentServerInterfaceImpl();
9        Registry registry = LocateRegistry.getRegistry();
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 named `StudentServerInterfaceClient` in Listing 40.4. The client locates the server object from the RMI registry and uses it to find the scores.

LISTING 40.4 StudentServerInterfaceClient.java

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
```

```
3  import javafx.scene.control.Button;
4  import javafx.scene.control.Label;
5  import javafx.scene.control.TextField;
6  import javafx.scene.layout.GridPane;
7  import javafx.stage.Stage;
8  import java.rmi.registry.LocateRegistry;
9  import java.rmi.registry.Registry;
10
11 public class StudentServerInterfaceClient extends Application {
12   // Declare a Student instance
13   private StudentServerInterface student;
14
15   private Button btGetScore = new Button("Get Score");
16   private TextField tfName = new TextField();
17   private TextField tfScore = new TextField();
18
19   public void start(Stage primaryStage) {
20     GridPane gridPane = new GridPane();
21     gridPane.setHgap(5);
22     gridPane.add(new Label("Name"), 0, 0);
23     gridPane.add(new Label("Score"), 0, 1);
24     gridPane.add(tfName, 1, 0);
25     gridPane.add(tfScore, 1, 1);
26     gridPane.add(btGetScore, 1, 2);
27
28     // Create a scene and place the pane in the stage
29     Scene scene = new Scene(gridPane, 250, 250);
30     primaryStage.setTitle("StudentServerInterfaceClient");
31     primaryStage.setScene(scene); // Place the scene in the stage
32     primaryStage.show(); // Display the stage
33
34     initializeRMI();
35     btGetScore.setOnAction(e - > getScore());
36   }
37
38   private void getScore() {
39     try {
40       // Get student score
41       double score = student.findScore(tfName.getText().trim());
42
43       // Display the result
44       if (score < 0)
45         tfScore.setText("Not found");
46       else
47         tfScore.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
58     try {
59       Registry registry = LocateRegistry.getRegistry(host);
60       student = (StudentServerInterface)
61         registry.lookup("StudentServerInterfaceImpl");
62       System.out.println("Server object " + student + " found");
63     }
64     catch(Exception ex) {
```

```
65        System.out.println(ex);
66      }
67    }
68
69    /**
70    * The main method is only needed for the IDE with limited
71    * JavaFX support. Not needed for running from the command line.
72    */
73    public static void main(String[] args) {
74      launch(args);
75    }
76  }
```

`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 55–67), which is responsible for locating the server stub.

The `lookup(String name)` method (line 61) 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 40.5(b).

Note:
You must start rmiregistry from the directory where you will run the RMI server, as shown in Figure 40.6. Otherwise, you will receive the error `ClassNotFoundException` on `StudentServerInterfaceImpl_Stub`.

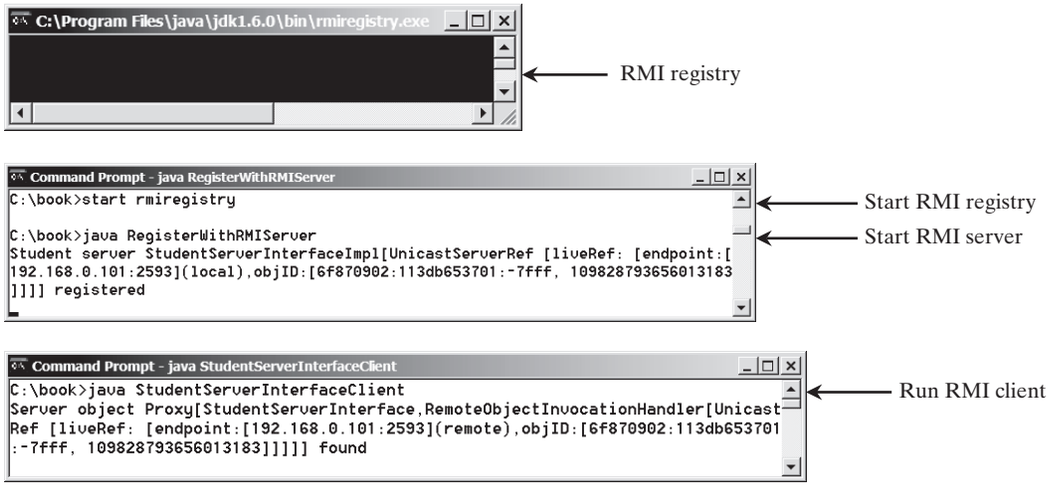

FIGURE 40.6 To run an RMI program, first start the RMIRegistry, then register the server object with the registry. The client locates it from the registry.

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.

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.

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

40.3.2 Describe the roles of the stub and the skeleton.

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

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

40.3.5 What is the command to start an RMI registry?

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

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

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

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

40.4 RMI vs. Socket-Level Programming

RMI is a high-level network programming and socket-level network programming is low-low-level.

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, whereas RMI programming is like programming in a high-level language.

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

40.5 Developing Three-Tier Applications Using RMI

RMI can be used in the middle between a client and a database to develop scalable and flexible business applications.

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, but 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 clients that need to access multiple databases on different servers, since the server may change.

To demonstrate, let us rewrite the example in Section 40.3.1, "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 Section 40.3.1 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 40.5. The server retrieves a record from the `Scores` table, processes the retrieved information, and sends the result back to the client.

LISTING 40.5 Student3TierImpl.java

```
1  import java.rmi.*;
2  import java.rmi.server.*;
3  import java.sql.*;
4
5  public class Student3TierImpl extends UnicastRemoteObject
6      implements StudentServerInterface {
7    // Use prepared statement for querying DB
8    private PreparedStatement pstmt;
9
10   /** Constructs Student3TierImpl object and exports it on
11   * default port.
12   */
13   public Student3TierImpl() throws RemoteException {
14     initializeDB();
15   }
16
17   /** Constructs Student3TierImpl object and exports it on
18    * specified port.
19    * @param port The port for exporting
20    */
21   public Student3TierImpl(int port) throws RemoteException {
22     super(port);
```

```
23      initializeDB();
24    }
25
26    /** Load JDBC driver, establish connection and
27     * create statement */
28    protected void initializeDB() {
29      try {
30        // Load the JDBC driver
31        // Class.forName("oracle.jdbc.driver.OracleDriver");
32        Class.forName("com.mysql.jdbc.Driver ");
33
34        System.out.println("Driver registered");
35
36        // Establish connection
37        /*Connection conn = DriverManager.getConnection
38          ("jdbc:oracle:thin:@drake.armstrong.edu:1521:orcl",
39            "scott", "tiger"); */
40        Connection conn = DriverManager.getConnection
41          ("jdbc:mysql://localhost/javabook", "scott", "tiger");
42        System.out.println("Database connected");
43
44        // Create a prepared statement for querying DB
45        pstmt = conn.prepareStatement(
46          "select * from Scores where name = ?");
47      }
48      catch (Exception ex) {
49        System.out.println(ex);
50      }
51    }
52
53    /** Return the score for specified the name
54     * Return -1 if score is not found.
55     */
56    public double findScore(String name) throws RemoteException {
57      double score = -1;
58      try {
59        // Set the specified name in the prepared statement
60        pstmt.setString(1, name);
61
62        // Execute the prepared statement
63        ResultSet rs = pstmt.executeQuery();
64
65        // Retrieve the score
66        if (rs.next()) {
67          if (rs.getBoolean(3))
68            score = rs.getDouble(2);
69        }
70      }
71      catch (SQLException ex) {
72        System.out.println(ex);
73      }
74
75      return score;
76    }
77  }
```

`Student3TierImpl` is similar to StudentServerInterfaceImpl in Section 40.3.1 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 28–51) establishes connections with the database and creates a prepared statement for processing the query.

The `findScore` method (lines 56–76) 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 40.6) that registers the server object using StudentServerInterfaceImpl, the same name as in Listing 40.2, so that you can use `StudentServerInterfaceClient`, created in Section 40.3.1, to test the server.

LISTING 40.6 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();
8        registry.rebind("StudentServerInterfaceImpl", obj);
9        System.out.println("Student server " + obj + " registered");
10     } catch (Exception ex) {
11       ex.printStackTrace();
12     }
13   }
14 }
```

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`. A sample run is shown in Figure 40.6.

40.5.1 Describe how parameters are passed in RMI.

40.6 RMI Callbacks

RMI callbacks enable the server to invoke the methods on a client.

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 Section 33.6, "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 40.7 illustrates the relationship between clients and server.

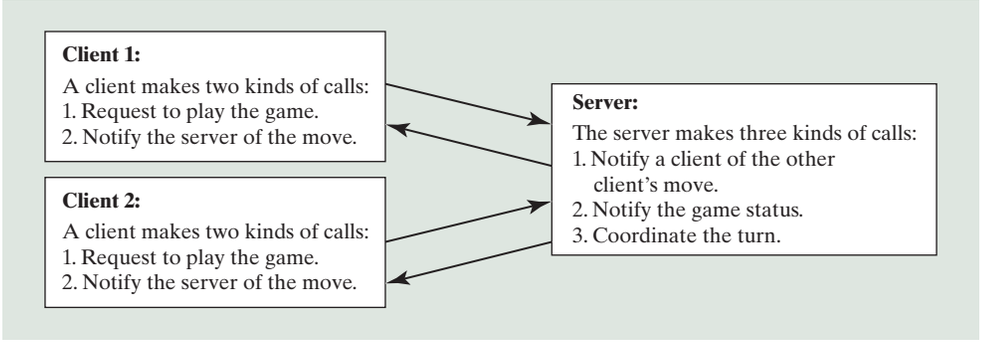

FIGURE 40.7 The server coordinates the activities with the clients.

All the calls a client makes can be encapsulated in one remote interface named `TicTacToe` (Listing 40.7), and all the calls the server invokes can be defined in another interface named `CallBack` (Listing 40.8). These two interfaces are defined as follows:

LISTING 40.7 TicTacToeInterface.java

```
1  import java.rmi.*;
2
3  public interface TicTacToeInterface extends Remote {
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;
10
11   /** A client invokes this method to notify the server of its move*/
12   public void myMove(int row, int column, char token)
13     throws RemoteException;
14 }
```

LISTING 40.8 CallBack.java

```
1  import java.rmi.*;
2
3  public interface CallBack extends Remote {
4    /** The server notifies the client for taking a turn */
5    public void takeTurn(boolean turn) throws RemoteException;
6
7    /** The server sends a message to be displayed by the client */
8    public void notify(java.lang.String message)
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)
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 40.8.

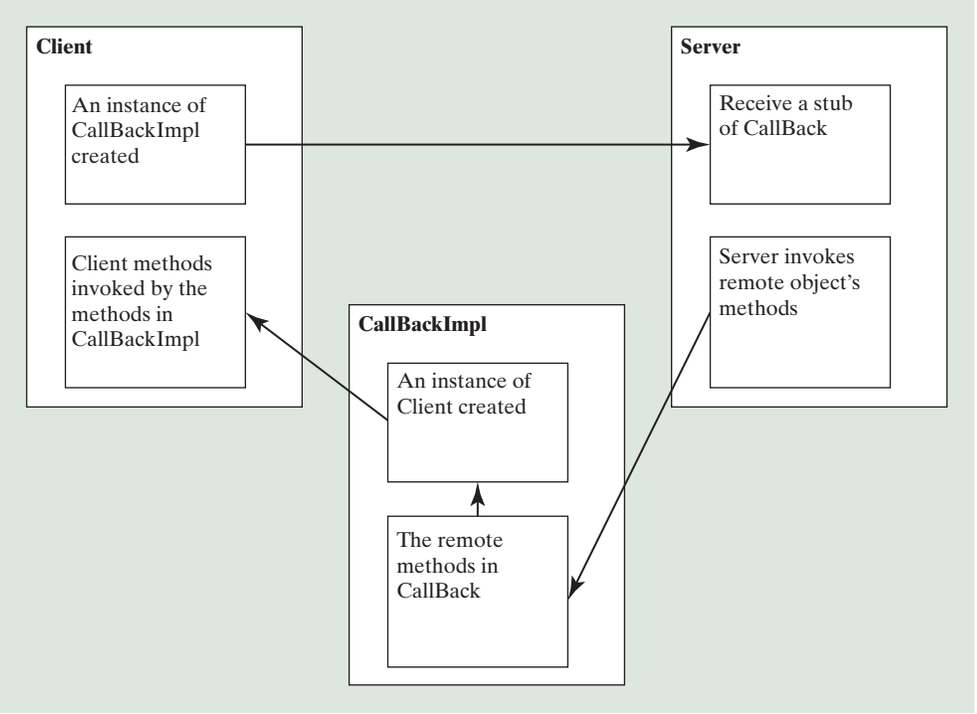

FIGURE 40.8 The server receives a `CallBack` stub from the client and invokes the remote methods defined in the `CallBack` interface, which can invoke the methods defined in the client.

Here are the steps to complete the example.

1. Create TicTacToeImpl.java (Listing 40.9) to implement `TicTacToeInterface`. Add a main method in the program to register the server with the RMI.

LISTING 40.9 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;
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 {
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)
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 */
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 */
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);
140       System.out.println("Server " + obj + " registered");
141     }
142     catch (Exception ex) {
143       ex.printStackTrace();
144     }
145   }
146 }
```

2. Create CallBackImpl.java (Listing 40.10) to implement the `CallBack` interface.

LISTING 40.10 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 {
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 {
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)
26       throws RemoteException {
27     thisClient.mark(row, column, token);
28   }
29 }
```

3. Create a client named `TicTacToeClientRMI` (Listing 40.11) for interacting with a player and communicating with the server. Enable it to run standalone.

LISTING 40.11 TicTacToeClientRMI.java

```
1  import java.rmi.*;
2
3  import javafx.application.Application;
4  import javafx.application.Platform;
5  import javafx.stage.Stage;
6  import javafx.scene.Scene;
7  import javafx.scene.control.Label;
8  import javafx.scene.layout.BorderPane;
9  import javafx.scene.layout.GridPane;
10 import javafx.scene.layout.Pane;
11 import javafx.scene.paint.Color;
12 import javafx.scene.shape.Line;
13 import javafx.scene.shape.Ellipse;
14
15 import java.rmi.registry.Registry;
16 import java.rmi.registry.LocateRegistry;
17
18 public class TicTacToeClientRMI extends Application {
19   // marker is used to indicate the token type
20   private char marker;
21
22   // myTurn indicates whether the player can move now
23   private boolean myTurn = false;
```

```
24
25    // Indicate which player has a turn, initially it is the X player
26    private char whoseTurn = 'X';
27
28    // Create and initialize cell
29    private Cell[][] cell = new Cell[3][3];
30
31    // Create and initialize a status label
32    private Label lblStatus = new Label("X's turn to play");
33
34    // ticTacToe is the game server for coordinating
35    // with the players
36    private TicTacToeInterface ticTacToe;
37
38    private Label lblIdentification = new Label();
39
40    @Override // Override the start method in the Application class
41    public void start(Stage primaryStage) {
42      // Pane to hold cell
43      GridPane pane = new GridPane();
44      for (int i = 0; i < 3; i++)
45        for (int j = 0; j < 3; j++)
46          pane.add(cell[i][j] = new Cell(i, j), j, i);
47
48      BorderPane borderPane = new BorderPane();
49      borderPane.setCenter(pane);
50      borderPane.setTop(lblStatus);
51      borderPane.setBottom(lblIdentification);
52
53      // Create a scene and place it in the stage
54      Scene scene = new Scene(borderPane, 450, 170);
55      primaryStage.setTitle("TicTacToe"); // Set the stage title
56      primaryStage.setScene(scene); // Place the scene in the stage
57      primaryStage.show(); // Display the stage
58
59      new Thread( () -> {
60      try {
61        initializeRMI();
62      }
63      catch (Exception ex) {
64        ex.printStackTrace();
65      }}).start();
66    }
67
68    /** Initialize RMI */
69    protected boolean initializeRMI() throws Exception {
70      String host = "";
71
72      try {
73        Registry registry = LocateRegistry.getRegistry(host);
74        ticTacToe = (TicTacToeInterface)
75          registry.lookup("TicTacToeImpl");
76        System.out.println
77          ("Server object " + ticTacToe + " found");
78      }
79      catch (Exception ex) {
80        System.out.println(ex);
81      }
82
83      // Create callback for use by the
84      // server to control the client
```

```
85      CallBackImpl callBackControl = new CallBackImpl(this);
86
87      if (
88        (marker =
89          ticTacToe.connect((CallBack)callBackControl)) != ' ')
90      {
91        System.out.println("connected as " + marker + " player.");
92        Platform.runLater(() ->
93          lblIdentification.setText("You are player " + marker));
94        return true;
95      }
96      else {
97        System.out.println("already two players connected as ");
98        return false;
99      }
100   }
101
102   /** Set variable myTurn to true or false */
103   public void setMyTurn(boolean myTurn) {
104     this.myTurn = myTurn;
105   }
106
107   /** Set message on the status label */
108   public void setMessage(String message) {
109     Platform.runLater(() -> lblStatus.setText(message));
110   }
111
112   /** Mark the specified cell using the token */
113   public void mark(int row, int column, char token) {
114     cell[row][column].setToken(token);
115   }
116
117   // An inner class for a cell
118   public class Cell extends Pane {
119     // marked indicates whether the cell has been used
120     private boolean marked = false;
121
122     // row and column indicate where the cell appears on the board
123     int row, column;
124
125     // Token used for this cell
126     private char token = ' ';
127
128     public Cell(final int row, final int column) {
129       this.row = row;
130       this.column = column;
131       setStyle("-fx-border-color: black");
132       this.setPrefSize(2000, 2000);
133       this.setOnMouseClicked(e -> handleMouseClick());
134     }
135
136     /** Return token */
137     public char getToken() {
138       return token;
139     }
140
141     /** Set a new token */
142     public void setToken(char c) {
143       token = c;
144       marked = true;
145
```

```
146         if (token == 'X') {
147           Line line1 = new Line(10, 10,
148             this.getWidth() - 10, this.getHeight() - 10);
149           line1.endXProperty().bind(this.widthProperty().subtract(10));
150           line1.endYProperty().bind(this.heightProperty().subtract(10));
151           Line line2 = new Line(10, this.getHeight() - 10,
152             this.getWidth() - 10, 10);
153           line2.startYProperty().bind(
154             this.heightProperty().subtract(10));
155           line2.endXProperty().bind(this.widthProperty().subtract(10));
156
157           // Add the lines to the pane
158           Platform.runLater(() ->
159             this.getChildren().addAll(line1, line2));
160         }
161         else if (token == 'O') {
162           Ellipse ellipse = new Ellipse(this.getWidth() / 2,
163             this.getHeight() / 2, this.getWidth() / 2 - 10,
164             this.getHeight() / 2 - 10);
165           ellipse.centerXProperty().bind(
166             this.widthProperty().divide(2));
167           ellipse.centerYProperty().bind(
168             this.heightProperty().divide(2));
169           ellipse.radiusXProperty().bind(
170             this.widthProperty().divide(2).subtract(10));
171           ellipse.radiusYProperty().bind(
172             this.heightProperty().divide(2).subtract(10));
173           ellipse.setStroke(Color.BLACK);
174           ellipse.setFill(Color.WHITE);
175
176           Platform.runLater(() ->
177             getChildren().add(ellipse)); // Add the ellipse to the pane
178         }
179       }
180
181       /* Handle a mouse click event */
182       private void handleMouseClick() {
183         if (myTurn && !marked) {
184           // Mark the cell
185           setToken(marker);
186
187           // Notify the server of the move
188           try {
189             ticTacToe.myMove(row, column, marker);
190           }
191           catch (RemoteException ex) {
192             System.out.println(ex);
193           }
194         }
195       }
196     }
197
198     /**
199      * The main method is only needed for the IDE with limited
200      * JavaFX support. Not needed for running from the command line.
201      */
202     public static void main(String[] args) {
203       launch(args);
204     }
205   }
```

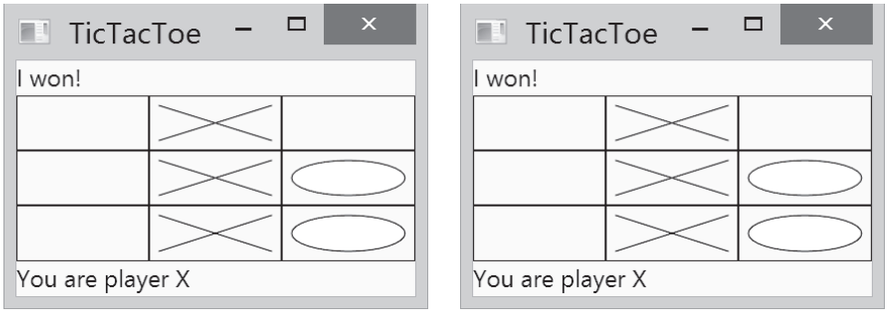

FIGURE 40.9 Two players play each other through the RMI server.

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`. A sample run is shown in Figure 40.9.

`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 40.7 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.

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 40.8 suggests alternative approaches that implement the `Cell` as a noninner class.

40.6.1 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;
```

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

Key Terms

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.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

Programming Exercises

MyProgrammingLab™

Section 40.3

*40.1 (*Limit the number of clients*) Modify the example in Section 40.3.1, "Example: Retrieving Student Scores from an RMI Server," to limit the number of concurrent clients to 10.

*40.2 (*Compute loan*) Rewrite Programming Exercise 33.1 using RMI. You need to define a remote interface for computing monthly payment and total payment.

**40.3 (*Web visit count*) Rewrite Programming Exercise 33.4 using RMI. You need to define a remote interface for obtaining and increasing the count.

**40.4 (*Display and add addresses*) Rewrite Programming Exercise 33.6 using RMI. You need to define a remote interface for adding addresses and retrieving address information.

Section 40.5

**40.5 (*Address in a database table*) Rewrite Programming Exercise 40.4. Assume that the address is stored in a table.

**40.6 (*Three-tier application*) Use the three-tier approach to modify Programming Exercise 40.4, as follows:

- Create a JavaFX client to manipulate student information, as shown in Figure 33.23a.
- Create a remote object interface with methods for retrieving, inserting, and updating student information, and an object implementation for the interface.

Section 40.6

**40.7 (*Chat*) Rewrite Programming Exercise 33.13 using RMI. You need to define a remote interface for sending and receiving a message.

**40.8 (*Improve TicTacToe*) Modify the TicTacToe example in Section 40.6, "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.