Chapter 10

# Creating Distributed Applications Using RMI and JDBC

In this chapter:

- Understanding remote method invocation (RMI)
- Creating a multitier database application using RMI
- Implementing RMI in applets
- A working RMI applet example
- Connecting to the database from the RMI server host
- Parameterising connection information

This chapter introduces the essentials of creating distributed database applications using remote method invocation (RMI) and JDBC. Together, JDBC and RMI will allow you to implement highly scalable, distributed database applications. By the end of the chapter, you will understand how to connect Java client objects to remote objects using RMI, and how to invoke remote methods that connect to the database using JDBC.

## Understanding Remote Method Invocation (RMI)

The RMI technology lets you easily develop networked Java applications that hide the underlying mechanisms for transporting method arguments and return values.

RMI enables programmers to create distributed Java-to-Java programs. Under distributed Java computing, one part of the program—running in one Java Virtual Machine as a client—can invoke the methods of server objects in another virtual machine, often on a different host.

However, client programs do not communicate directly with server objects. When invoking remote methods, clients instead invoke the methods of a remote object's *stub,* which resides on the client host. The local stub does the networking needed to pass remote method invocations to the *skeleton* (another interface to the remote object) that resides on the server host.

Figure 10-1 shows the RMI architecture.

The RMI allows client and server objects to interact as if they were local objects residing in the same Java Virtual Machine. Additionally, server objects can be RMI clients of other remote objects.

### Passing Parameters and Return Values

RMI allows Java clients to pass parameter values to remote server methods and accept return values from those methods. It does this by allowing the stubs and skeletons to *marshal* and *unmarshal*

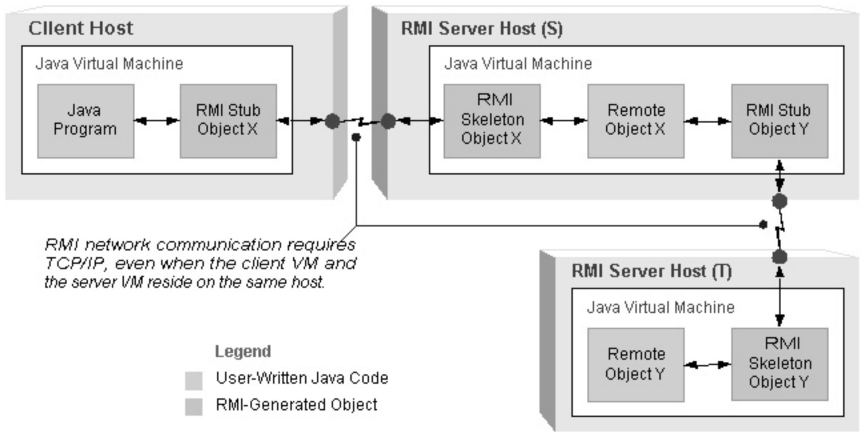


***Figure 10-1.*** *RMI architecture.*

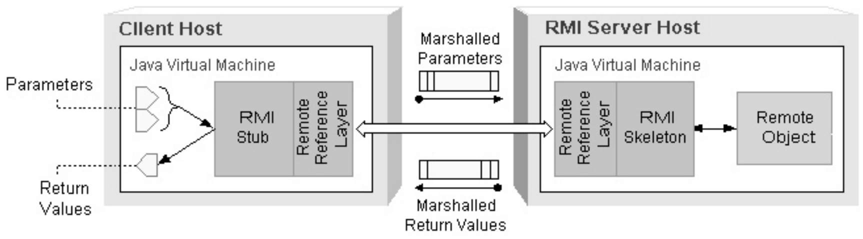


***Figure 10-2.*** *Marshalling parameters and return values.*

method parameters and return values. Figure 10-2 shows the characteristics of object marshalling.

### *What Is Marshalling?*

*Marshalling refers to the process of packaging method arguments and return values so that they can be moved across the network. This process involves* serialization, *which is a way to convert objects so they can be passed over the network. Serializable objects include primitive types, remote Java objects, AWT components, and any other object that implements the serializable interface. Note that, if you declare a Java property as transient, then that property will not be serialized.*

## Essential RMI Components

The essential RMI components that we will use to create a working RMI application are listed in Table 10-1. The next few sections will show in greater detail how to use these components.

## Useful Links

For more information about RMI, you can view the following documentation:

JavaSoft RMI Documentation at:

`http://Java.sun.com/products/jdk/1.1/docs/guide/rmi/index.html`

***Table 10-1. Essential RMI components.***

| ***Component*** | ***Function*** |
|---|---|
| RMI compiler (`rmic`) | Generates the stubs and skeletons for the remote server class. |
| Stubs | A client view of the remote methods. |
| Skeletons | On the server, connects the stub to the remote server class. |
| Remote interface | An interface that defines the remote methods that the remote server class will implement in code. |
| Remote server class | A class that implements the methods defined in the interface file. |
| Server factory | A program that generates an instance of the remote server class, and registers it with an RMI registry. |
| RMI registry service | The RMI bootstrap registry program that runs on the server host and registers remote RMI objects by name on the network. |

JavaSoft RMI FAQ at:

`http://Java.sun.com/products/jdk/1.1/docs/guide/rmi/faq.html`

JavaSoft RMI Specification at:

`http://Java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html`

## Creating a Multitier Database Application Using RMI

The steps involved in creating a distributed database application are as follows:

1. **Define the remote interfaces.** Create and compile an interface class that extends `java.rmi.Remote`. This interface defines the methods that the server implements. The client Java program will call these methods as if they were local, when in fact they are remote.
2. **Define the remote classes.** Create and compile a remote server class that extends a `RemoteObject` such as `java.rmi.server.UnicastRemoteObject`. This class will implement the interfaces defined in Step 1.

3. **Create stubs and skeletons.** Using the RMI compiler, `rmic`, generate stubs and skeletons for the implementation class you created in Step 2.
4. **Create a server program to generate the remote object.** Create and compile a server program that creates an instance of the remote object you created in Step 2 and then registers the object by name with an RMI registry service.
5. **Start the RMI registry service on the server.** On the server host, start the bootstrap registry service, `rmiregistry`, and start the server program that you created in Step 4.
6. **Create the Java client program.** Create and compile the client Java program that will call the remote methods.

## Define the Remote Interfaces

What we are trying to do is run SQL queries using distributed Java objects. To manipulate a remote server object, client code needs to know what it can do with that object. Therefore, an interface is shared between the client and server. It is this interface that exposes the methods of the remote object to the client.

### Create the Remote Interface

We create an interface in a file called `JR.java`. To create a remote interface, import the RMI package in your Java code and, if you will use SQL, also import the SQL package. The interface in List. 10-1 declares methods that open and close database connections, execute queries, and retrieve table rows.

In order to expose the remote methods to a client, the interface must extend `java.rmi.Remote`.

Each exposed method must declare `java.rmi.RemoteException` in its `throws` clause, because many things can go wrong when making remote invocations over the network. Calls to remote hosts are inherently less reliable than calls to the local host. For example, the remote host might not be available when you make the remote call.

Note also that, because we are using the SQL package, we also need to declare `java.sql.SQLException` in the `throws` clause of the remote methods.

**tip**

*Try to maintain the design of the remote interfaces. If you are using parameters, try to encapsulate all the related parameters into a single object. If you need to add another parameter, you don't have to change the remote interface, only the parameter class.*

**Listing 10-1. Define the remote methods in an interface (`JR.java`).**

```
import java.rmi.*;
import java.sql.*;

public interface JR extends java.rmi.Remote {

          public int openConnection()
            throws RemoteException,
                 SQLException,
                 ClassNotFoundException;

          public void closeConnection( int id )
            throws RemoteException,
                 SQLException;

          public void performQuery( int id, String searchString )
            throws RemoteException,
                 SQLException;

          public String getNextRow( int id )
            throws RemoteException,
                 SQLException;
}
```

## Compile and Locate the Interface

You compile the interface file using the javac command.

```
javac JR.java
```

This produces a file called `JR.class`.

For stand-alone Java applications, place the compiled interface on *both* the client host and on the server host.

On both hosts, point the `CLASSPATH` environment variable to the directory in which you placed the interface. This must be done before starting the client program on the client host and before starting the registry service and server factory on the server host.

For client programs that are Java applets, place the compiled interface into the same directory on the RMI server host in which the applet resides.

Once you've declared and compiled an interface class, your next action is to define the remote server class that implements the remote methods you've declared.

tip

*You can place compiled interfaces anywhere on the RMI server. Make sure the* CLASSPATH *environment variable used when starting the server program points to the location of these interface classes.*

## Define the Remote Classes

This step creates the *remote server class*. The remote server class implements all of the methods defined by the interface we created in Step 1 by creating the actual Java methods to match the interface signatures.

### *What Does "Remote Server Class" Mean?*

*This implementation class is a server class because it extends* UnicastRemoteObject*, which makes objects remotely accessible. This class inherits the basic technology for communication between the server object and its stubs on the client host, so you don't have to code it.*

### Create the Remote Server Class

You can create methods in this class that are not remote. In more complex applications, you can create many interfaces and many remote server classes. This is a design issue; generally, the output of the design phase would be a set of interfaces that have one or more defined as remote.

Because the Java platform handles the marshalling of data and remote invocation, you don't have to code it. Apart from defining the remote nature of the methods, you concentrate on the application functionality you need to code.

In this example, we define the remote server by creating a Java class called `JRImpl` in a file called `JRImpl.java`, shown in List. 10-2.

Note that we import the `java.rmi.server` package into the class.

In order to define the class as containing methods that can be accessed remotely, we define `JRImpl` as a subclass of `UnicastRemoteObject`, implementing the interface `JR`.

***Listing 10-2. Remote server class `JRImpl.java`.***

```
import java.rmi.*;
import java.rmi.server.*;
import java.sql.*;

public class JRImpl
          extends UnicastRemoteObject
          implements JR {

//Default constructor
public JRImpl () throws RemoteException { }

          public synchronized int openConnection()
             throws RemoteException,
                SQLException,
             ClassNotFoundException
          {
    . . .
          }

          public void closeConnection( int id )
             throws RemoteException,
                SQLException
          {
    . . .
          }

          public void performQuery( int id, String searchString )
             throws RemoteException,
                SQLException
          {
    . . .
          }

          public String getNextRow( int id )
             throws RemoteException,
                SQLException
          {
      . . .
          }
}
```

### *Why Use* `UnicastRemoteObject`*?*

*The* `UnicastRemoteObject`, *along with other RMI components, takes care of all remote method invocation tasks; you need not take any special action whatsoever, other than to ensure that all remote methods declared in the interface class declare* `java.rmi.RemoteException` *in their respective* `throws` *clauses.*

### Compile and Locate the Remote Server Class

Compile the remote server class using `javac`:

```
javac JRImpl.java
```

This produces a file called `JRImpl.class`. You can place compiled remote server classes anywhere on the RMI server host. Make sure the `CLASSPATH` environment variable points to the location of these remote server classes.

Alternatively, place all the supporting classes that you develop for a particular program in the same directory on the RMI server; and point the `java.rmi.server.codebase` property to that directory on the command line when starting the *server factory* (see subsection "Start the Server Factory" later in this chapter).

**important** *RMI client and server programs should install a security manager to control the actions of stub and skeleton objects loaded from a network location. For applets, there is a default security manager to ensure that the applet does no harm. For stand-alone applications, there is no default security manager, so you must set one when using RMI.*

**note** *Remote server objects must already be running on the server host when a service is requested from the client.*

## Create Stubs and Skeletons

With RMI, client programs and remote objects use proxies, called stubs and skeletons, to handle the necessary networking between them.

You don't need to be concerned in any way with stubs and skeletons other than how to create and locate them.

Stubs and skeletons are Java classes. You place the compiled stub on the client (unless it's an applet, in which case you bundle it with the applet code), and the compiled skeleton on the server. Note that the RMI server host may cooperate as a client for another RMI server host in a highly distributed application. Even so, the steps to follow are the same.

### Generate the Stub and Skeleton Classes

You create stubs and skeletons using `rmic` (RMI compiler), which ships with the JDK. Make sure that you have already compiled the remote interface, `JR.java`, and the remote server, `JRImpl.java`, before you generate the stubs and skeletons.

You generate the stub and skeleton for the remote server, `JRImpl.class`, as follows:

```
rmic JRImpl
```

This creates two class files:

```
JRImpl_Stub.class
JRImpl_Skel.class
```

note

*If the compiled class is part of a Java package, give the full package name to* `rmic`.

### Locate the Stubs and Skeletons

You can place the generated skeletons, `*.Skel`, anywhere on the RMI server. Make sure that the `CLASSPATH` environment variable used when starting the server program points to the location of the skeleton classes. Typically, you'd place all the supporting classes for the server application in the same directory on the RMI server, and point the `CLASSPATH` environment variable to that directory when starting the server factory.

For stand-alone Java applications, you need to copy the stub classes to a directory on the client host and point the `CLASSPATH` environment variable to that directory before starting the client program.

For Java applets, you need to place the stub classes into the same directory on the RMI server host in which the applet resides.

## Create a Server Program to Generate the Remote Object

You need to create and compile a server program that creates an instance of the remote object and then registers the object by name with an RMI registry service. When the remote object is registered with the RMI registry service, the clients can access the remote methods in the object.

The server program that we create now is called a *server factory* because it instantiates the remote object.

## Creating the Server Factory

For a client program to access a remote object, the client must first have a reference, or *object handle,* to that remote object. To get that reference, the client can invoke a remote method in the server factory running on the RMI server host, which returns a reference to the object that contains the remote methods.

Server programs register remote objects with the *bootstrap registry service*. Once the remote object is registered, server programs can return references to the object.

You register a remote object on the RMI server host by giving a reference to the object (servlet) and some name (`rmiExample`) to the bootstrap registry. This registration process *binds* the remote object by name to a host (`dellcpi`) and port (1099). These objects are summarized in the Table 10-2. The `JRServer.java` is shown in List. 10-3.

The class that contains the remote methods is instantiated in the following line:

```
JRImpl servlet = new JRImpl ();
```

The object is then registered with the RMI registry service:

```
Naming.rebind(
  "//dellcpi.eu.informix.com/rmiExample", servlet );
```

***Table 10-2. Important objects in*** **`JRServer.java`.**

| ***Object*** | ***Description*** |
|---|---|
| `servlet` | An instance of the class that implements the remote methods the client will invoke. |
| `Naming.rebind` | Naming provides the default registry service. The rebind method adds an object to the RMI registry. |
| `//dellcpi:1099` | Host and port on which the RMI registry runs. |
| `rmiExample` | A label for the registered object; all clients can use the label. |

This is all you need to do to set up the remote factory to instantiate remote objects; RMI does everything else.

## Locating the Server Factory

You can place the compiled server factory class anywhere on the RMI server host, but make sure that the CLASSPATH environment variable used when starting the server factory points to the location of this server factory class.

If your Java client is an applet, then the server factory must be running before the applet starts.

***Listing 10-3. The server factory, `JRServer.java`, which creates and registers objects with the RMI registry.***

```
import java.rmi.*;
import java.rmi.registry.*;
import java.net.*;

public class JRServer {
  public static void main( String args[] ) {

    String hostName   = null;
    String hostIP   = null;
    try {
           java.net.InetAddress hostInet =
                            java.net.InetAddress.getLocalHost();
    hostName = hostInet.getHostName();
    hostIP   = hostInet.getHostAddress();
    System.out.println(
                 "Starting RemoteSQL on " + hostName +
                 " ["+hostIP+"]");
        }
        catch (java.net.UnknownHostException e) {
         System.err.println("JRServer.main exception: "
                            + e.getMessage());
         e.printStackTrace();
        }
        if ( System.getSecurityManager() == null ) {
         System.setSecurityManager( new RMISecurityManager() );
        }
```

***Listing 10-3. (continued)***

```
        try {
            JRImpl servlet = new JRImpl ();

                    System.out.println( "Binding to the registry" );
            Naming.rebind(
               "//dellcpi.eu.informix.com/rmiExample", servlet );
            System.out.println( "...RemoteSQL is bound and ready." );
            }
        catch ( Exception e ) {
                System.err.println("JRServer.main exception: "
                                   + e.getMessage());
                e.printStackTrace();
        }
   }
}
```

## Start the RMI Registry Service on the Server

The RMI bootstrap registry program runs on the server host and registers remote RMI objects by name on the network. Once registered, remote objects are in effect exported and can be accessed by RMI clients.

To access a remote object, a client program first gets a reference to the object by using the `Naming` class to look up the object using its registered name. The client program specifies the name as an rmi: URL. The registry service then returns an instance of the remote interface for the remote object—the object's *stub.*

### Starting the Registry Naming Service on the RMI Server

Set your environment to point to the location of your Java installation. On UNIX, for example:

```
setenv JAVA_HOME /usr/local/java/jdk1.1.7
setenv PATH $JAVA_HOME/bin:$PATH
```

The Windows equivalent would be

```
set JAVA_HOME=D:\java\jdk1.1.7
set PATH=%JAVA_HOME%\bin;%PATH%
```

You can then start the registry naming service, `rmiregistry`. You start this program on UNIX as follows:

```
unsetenv CLASSPATH
rmiregistry &
```

On Windows:

```
unset CLASSPATH
start rmiregistry
```

In the Windows example, if `unset` doesn't work on your system, use `set CLASSPATH=`.

**note** *Before starting the registry,* `CLASSPATH` *should be* `unset`*; at least, it should not be pointing to any classes. When the* `rmiregistry` *program starts, if it can find your stub classes, it won't remember that these stubs can be loaded from your server's codebase.*

**tip** *The registry runs on port 1099 by default. To start the registry on a different port, specify the port number on the command line. For example,* `start rmiregistry 1090`*. The RMI bootstrap registry must remain running so that the registry's naming service is always available.*

### Start the Server Factory

Set `CLASSPATH` as appropriate and start the server program that you created in Step 4, which creates an object that implements the methods accessed remotely by the client. You can start the server program as follows:

```
java JRServer
```

## Create the Java Client Program

Client programs use remote objects that have been exported by remote server programs. In order to do this, the client program must look up the remote object in the remote RMI registry. When the remote object is located, the stub of the remote object is sent to the client. The client invokes the methods in this stub as if the stub were the actual remote object in the local Java Virtual Machine. The stub communicates with the remote skeleton associated with that stub, and the skeleton invokes the method that the remote client has requested.

## Obtaining a Reference to the Remote Object

The client uses the `Naming.lookup` method to obtain a reference to the remote object. The returned value is actually a reference to a stub object.

```
JR servlet = (JR)Naming.lookup( registryName );
```

The `registryName` variable should define the RMI server host and port that the RMI registry runs on, and it should also state the name of the object.

For example, the applet in List. 10-4 is downloaded from `//dellcpi.eu.informix.com`. The RMI registry on that host is running on the default port, 1099. The required `JRImpl` object is labeled `rmiExample`. Therefore, the URL for the `lookup` method is:

```
//dellcpi.eu.informix.com/rmiExample
```

If the RMI server was started on a different port (`rmiregistry 1090`), then the URL would be:

```
//dellcpi.eu.informix.com:1090/rmiExample
```

## Implementing an RMI Security Manager on the Client

Because the Java client will be downloading an untrusted stub object, implement the `RMISecurityManager` object in the client code:

```
System.setSecurityManager( new RMISecurityManager() );
```

Listing 10-4 shows the `init()` method of a Java applet that uses RMI to connect to the remote instance of the `JR` class that is stored in the RMI registry.

***Listing 10-4. The `TestApplet.java` client applet.***

```
import java.rmi.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class TestApplet extends Applet
         implements ActionListener {

         // install security manager
         public void init() {
            if (System.getSecurityManager() == null) {
               System.setSecurityManager(
                        new RMISecurityManager() );
            }

            //get a reference to the remote object
            String registryName;
            try {
               // Get the name of the host machine from where this
               // applet was loaded. The remote object and its
            // interface resides on that host.
                  registryName = "//" +
               getCodeBase().getHost() +
                  "/";

               // Append the predetermined name of the remote
               // object that has been bound to the registry
               // on the server host
                  registryName += "rmiExample";

               // Get a reference to the remote object.
         JR servlet = (JR)Naming.lookup( registryName );
               }
            catch ( Exception e ) {
               showStatus( "Cannot connect to RMI registry." );
            return;
            }

               // invoke the remote methods
               try {
```

***Listing 10-4. (continued)***

```
            connectionId = servlet.openConnection();
            }
            catch (RemoteException e) {
               showStatus("Failed to Open Connection");
            }
         }
}
```

## Implementing RMI in Applets

Because of applet security restrictions, untrusted applets can make network connections only to the host from which they were loaded. This means that all server objects must reside on the same machine that hosts the web page. These objects include

- the HTML web page that contains the `APPLET` tag,
- the applet class,
- the compiled stubs,
- the compiled skeletons,
- the compiled server classes and objects,
- the RMI bootstrap registry.

For applets, compiled stubs are bundled with the applet code. You need to place the stub classes into the same directory on the RMI server host in which the applet resides. In addition, place the compiled interfaces (e.g., `JR.class`) into the same directory.

***tip***

*The remote object that an applet can reference is not prevented from connecting to other machines and objects on the network. For example, the remote object on* machine 1 *can implement a method to invoke another remote method in an object on another machine,* machine 2. *If an applet required service from a remote object on* machine 2, *it would call the remote method on* machine 1 *to manage this interface.*

note

*The RMI client and server programs should install a security manager to control the actions of stub and skeleton objects loaded from a network location. For applets, there is a default security manager to ensure that the applet does no harm. For stand-alone applications, there is no default security manager, so you must set one when using RMI.*

note

*Registry and server objects must be running before the applet starts.*

## A Working RMI Applet Example

Once you have the RMI registry running and the server factory has instantiated and registered the remote object, you can invoke the client application.

Listing 10-4 contained the `init()` method of a Java applet that attempted to open a JDBC connection to a database through a remote object. Listing 10-4 is a good summary of the essentials of establishing a connection to a remote object.

This section lists a fully working applet that calls methods in the remote object (`JRImpl.java`) to open a database connection and execute SQL commands entered by the user at the browser.

To run this example, make sure that you follow all the steps in the previous section so that you have a registered instance of the `JRImpl` class in the RMI registry.

warning

*To run RMI-based applets, the viewer or browser used to run the applet must support the RMI features of Java 1.1. Browsers not supporting these features may crash.*

This section lists the following code:

- `RemoteSQLApplet.java`, which is the applet that requests a remote connection from the browser
- `JR.html`, which contains the `APPLET` tag to download and instantiate the applet.

### The `RemoteSQLApplet` Class

This Java class creates and displays a form in the browser so that the user can submit `SELECT` queries to the database and display the results. Listing 10-5 shows the Java source code.

***Listing 10-5. Java source code for*** **`RemoteSQLApplet.java`.**

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
import java.rmi.*;
import java.rmi.registry.*;


public class RemoteSQLApplet
         extends Applet
   implements ActionListener {

  Panel      north, center;
  TextField  searchCriteria;
  Button     searchButton, clearButton;
  List       searchResults;
  extArea    samples;

  int        connectionId;
  JR         servlet;

  //Initializes the applet, displays a GUI
  // for entering SQL queries, and opens a
  // connection to the remote object.
  public void init() {
        String   registryName;

        // Build the applet display.
        setLayout( new BorderLayout() );

        north = new Panel();
        north.setLayout( new FlowLayout() );
        north.add( new Label( "Enter SELECT statement:" ) );
        searchCriteria = new TextField( 40 );
        north.add( searchCriteria );
        searchButton = new Button( "Query" );
        searchButton.addActionListener( this );
        north.add( searchButton );

        clearButton = new Button( "Clear" );
        clearButton.addActionListener( this );
        north.add( clearButton );
        add( "North", north );
```

**Listing 10-5. (continued)**

```
        searchResults = new List( 10, true );
        add( "Center", searchResults );

        Panel notes = new Panel();
        notes.setLayout( new BorderLayout() );
        samples = new TextArea( "", 4,60,
                  TextArea.SCROLLBARS_BOTH );
        samples.setFont( new Font( "TimesRoman",
                  Font.BOLD, 10 ) );
        samples.setBackground( Color.darkGray );
        samples.setForeground( Color.white );
        samples.setText("");
        notes.add( "North", samples );

        Panel messages = new Panel();
        messages.setFont( new Font( "SansSerif",
                 Font.BOLD + Font.ITALIC, 12 ) );
        messages.setBackground( Color.white );
        messages.setForeground( Color.black );
        messages.add(
           new Label( "Check your browser status bar" +
                        " for any SQL messages.", Label.CENTER ) );
        notes.add( "South", messages );

        add( "South", notes );

        validate();
        setVisible( true );

        try {
            // install security manager
                 if (System.getSecurityManager() == null) {
                     System.setSecurityManager(
                               new RMISecurityManager() );
             }

            searchResults.addItem("RemoteSQLApplet:init: Preparing
for registry lookup..." );
        registryName = "//" + getCodeBase().getHost() + "/";
        registryName += "rmiExample";
```

**Listing 10-5. (continued)**

```
            searchResults.addItem("RemoteSQLApplet:init: Looking up
                                   '"+registryName+"'..." );

        servlet = (JR)Naming.lookup( registryName );
        if ( servlet == null ) {
           searchResults.addItem(
"RemoteSQLApplet:init:Naming.lookup: Lookup failed. Servlet is null."
);
             return;
        }
            searchResults.addItem(
"RemoteSQLApplet:init: Lookup successful..." );
        }
      catch ( Exception e ) {
        showStatus( "Cannot CONNECT to RMI registry
                     for 'RemoteSQL'" );
        e.printStackTrace();
        return;
      }

      try {

         // Open a connection to the remote object.
         // This call causes the remote server
         // to load the JDBC driver on the server host and to
         // use that driver to establish
         //a JDBC connection to the "stores7" database.

            searchResults.addItem(
"RemoteSQLApplet:init: Starting OPEN db connection task..." );
        connectionId = servlet.openConnection();

            searchResults.addItem(
"RemoteSQLApplet:init: Finished OPEN db connection task..." );
        if ( connectionId == -1 ) {
                       searchResults.addItem(
"RemoteSQLApplet:init: Error during OPEN db connection task..." );
                       searchResults.addItem(
"-1: Cannot OPEN DATABASE connection." );
        }
      }
      catch (Exception ex) {
                  searchResults.addItem(
"RemoteSQLApplet:init: Exception during OPEN db connection task..."
);
                  searchResults.addItem(
"Exception: Cannot OPEN DATABASE connection." );
                  ex.printStackTrace();
      }
  }

  // Closes the connection to the remote object.
  // This call causes the remote server to close all
  // server host and database resources that are associated
  // with the JDBC connection. For example, all cursors,
  // result sets, statement objects, and connection objects
  // are freed.

  public void finalize() {
    try {
          servlet.closeConnection( connectionId );
    }
    catch (Exception ex) {}

    super.destroy();
  }

  // Handles the two action events belonging to this applet:
  // Query button and Clear button clicks.

  public void actionPerformed( ActionEvent e ) {
   if ( e.getActionCommand().equals( "Query" ) ) {
    try {
        showStatus( "" );
        performQuery();
      }
      catch (Exception ex) {
            showStatus( ex.toString() + ": Cannot issue Query");
      }
    }
    else {
```

*Listing 10-5. (continued)*

```
      showStatus( "" );
      clearListBox();
    }
    return;
  }


  public void clearListBox() {
   searchResults.removeAll();
  }

  // Passes the SQL string that was entered to the
  // remote object for execution and retrieves the result set
  // row by row from the remote object, displaying
  // each row as it's retrieved.

  public void performQuery() throws Exception {
     String searchString;
     String result;

     clearListBox();
     searchString = searchCriteria.getText();

     // Execute the query.
     servlet.performQuery( connectionId, searchString );

     // Get and display the result set.
     result = servlet.getNextRow( connectionId );
     if ( result == null ) {
                 showStatus( "No rows found using specified
                              search criteria." );
                 return;
     }

     while ( result != null ) {
               searchResults.addItem( result );
               result = servlet.getNextRow( connectionId );
     }
   }
}
```

## Using `RemoteSQLApplet` in the Browser

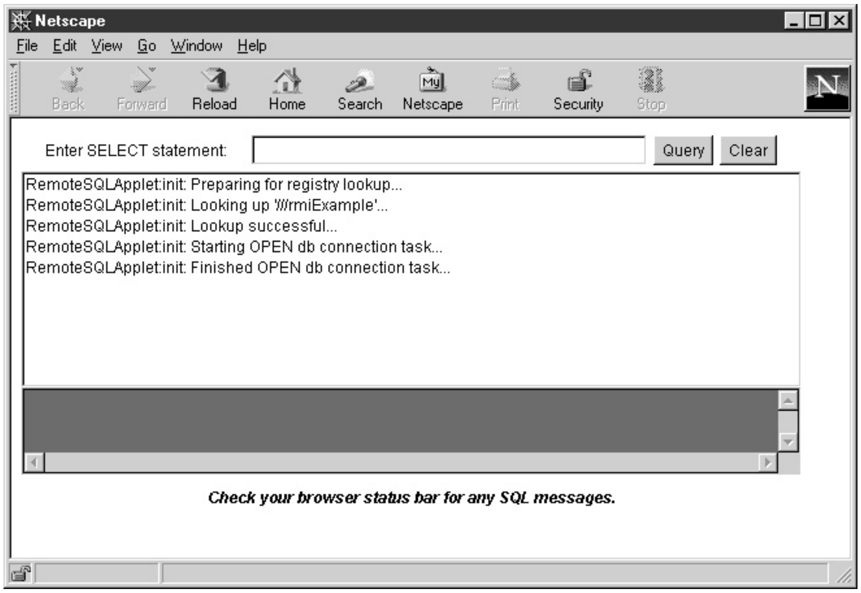
Listing 10-6 shows the HTML page, `JR.html`, that references the applet.

Figure 10-3 shows the `RemoteSQLApplet` class instantiated and displayed by the browser. When the applet attempts to open a remote connection to the database, the remote object writes a set of confirmatory messages to the console, which are shown in List. 10-7. The applet also confirms it's own processing state by writing to the List object, which is displayed in the applet window.

Listing 10-7 shows the output from the server factory, `JRServer`, from starting up the factory using `java JRServer`, connecting to

***Listing 10-6. The*** **`JR.html page.`**

```
<HTML>
<APPLET CODE="RemoteSQLApplet" WIDTH=600 HEIGHT=300></APPLET>
</HTML>
```


Netscape
File Edit View Go Window Help
Back Forward Reload Home Search Netscape Print Security Stop
Enter SELECT statement:
Query
Clear

```
RemoteSQLApplet:init: Preparing for registry lookup...
RemoteSQLApplet:init: Looking up '//rmiExample'...
RemoteSQLApplet:init: Lookup successful...
RemoteSQLApplet:init: Starting OPEN db connection task...
RemoteSQLApplet:init: Finished OPEN db connection task...
```

*Check your browser status bar for any SQL messages.*

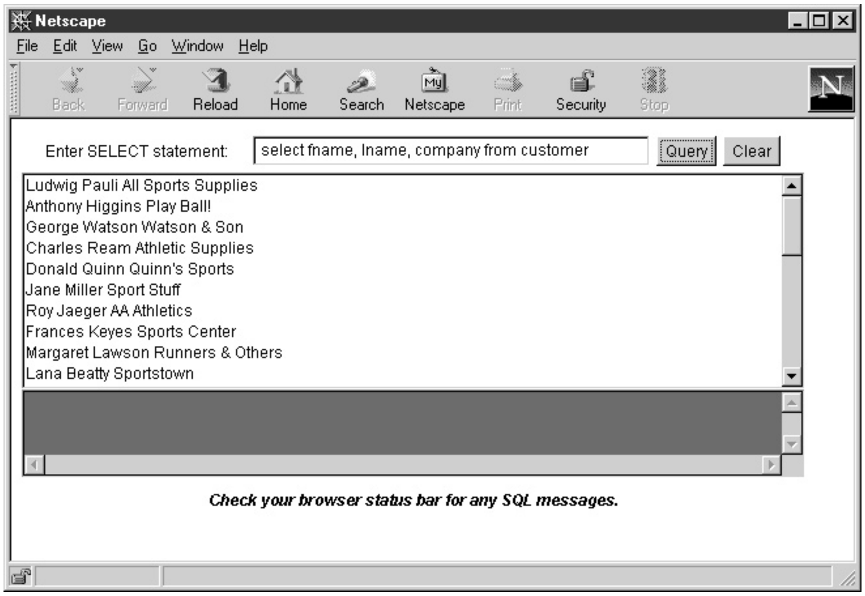
***Figure 10-3.*** *Starting* `RemoteSQLApplet` *in the browser.*

**Listing 10-7. Output from invoking `JRServer` from command line.**

```
Binding to the registry
...RemoteSQL is bound and ready.
JRImpl:openConnection: Starting...
JRImpl:openConnection: Creating connection object...
JRImpl:openConnection: Opening connection to DATABASE...
JRConn:openConnection:  Loading driver...
JRConn:openConnection:urlConnection: jdbc:informix-sqli://
dellcpi:1526/stores7:INFORMIXSERVER=ol_dellcpi;user=
informix;password=informix;
JRConn:openConnection:getConnection: Making connection...
JRImpl:openConnection: Returning connection Id...
JRConn:performQuery:stmt.executeQuery().
```

the database on behalf of the applet running in the client browser, and running an SQL query when requested from the client.

Figure 10-4 shows the browser output when the user enters a query string and clicks the `Query` button.


Netscape
File Edit View Go Window Help
Back Forward Reload Home Search Netscape Print Security Stop
Enter SELECT statement:
Query
Clear

```
select fname, lname, company from customer

Ludwig Pauli All Sports Supplies
Anthony Higgins Play Ball!
George Watson Watson & Son
Charles Ream Athletic Supplies
Donald Quinn Quinn's Sports
Jane Miller Sport Stuff
Roy Jaeger AA Athletics
Frances Keyes Sports Center
Margaret Lawson Runners & Others
Lana Beatty Sportstown
```

*Check your browser status bar for any SQL messages.*

**Figure 10-4.** *Browser output of query results.*

## Connecting to the Database from the RMI Server Host

We can use JDBC to connect to the Informix database from the remote object. This database can reside on the RMI server host, or it may be located on another visible host in the network.

Because we are not connecting to the database from the applet, we are not subject to the restrictions imposed on applets. The applet connects to the remote object on the RMI server that implements the remote methods it requires.

Listing 10-8 shows the `JRImpl` class. The remote object, `JRImpl`, creates an instance of the `JRConn` class for every connection established on behalf of a client. It is the `JRConn` class that implements the JDBC API calls. The remote methods in `JRImpl` are wrappers for calls to the `JRConn` methods. Multiple clients will all connect to the same `JRImpl` object. Collisions are avoided when opening connections by using the `synchronized` keyword as follows:

```
public synchronized int openConnection()
```

### *Why Use* `synchronized`*?*

*When a* `synchronised` *method is called, an* object lock *is taken at the start of the method. This prevents other objects from invoking the method while the lock is taken. When the method is exited, the lock is released. Because multiple remote client connections wait in turn to obtain object locks, each will occupy a different slot in the connection pool.*

***Listing 10-8. The*** `JRImpl.java class.`

```
public class JRImpl
        extends UnicastRemoteObject implements JR {

        // Create a table that can hold
        // connection (JRConn) objects.
        Private JRConn jrc [] = new JRConn [100];

        public JRImpl () throws RemoteException {
        super();
        }
```

***Listing 10-8. (continued)***

```
         // Implement each of the remote methods
         // specified in the remote interface, JR
         public synchronized int openConnection()
       throws RemoteException,
            SQLException,
            ClassNotFoundException {

            int connectionId;
            System.out.println(
               "JRImpl:openConnection: Starting...");

      // Loop through connection table
      // until an empty slot is found.
      for ( connectionId = 0; connectionId < jrc.length;
                              connectionId++ ) {
            if ( jrc [connectionId] == null )
            break;
         }

      // If no empty slots found, generate an error.
      if ( connectionId >= jrc.length ) {
            System.out.println(
         "WARNING: No more connection objects available"
               + " for RemoteSQL example." );
           return -1;
      }

      // Create a connection for the new process
      // and run it
      System.out.println(
"JRImpl:openConnection: Creating connection object...");
      jrc [connectionId] = new JRConn();

      System.out.println(
"JRImpl:openConnection: Opening connection to DATABASE...");
      jrc [connectionId].openConnection();

      // Return the connection identifier.
      System.out.println(
"JRImpl:openConnection: Returning connection Id...");
```

*Listing 10-8. (continued)*

```
         return connectionId;
      }// end openConnection

   // Executes the SQL query by passing the string
   // on to the connection object for
   // actual execution.
   Public void performQuery( int id,
                             String searchString )
      throws RemoteException,
         SQLException {
            jrc[id].performQuery( searchString );
   }

   // Fetches the next row from the current query result.
      Public String getNextRow( int id )
      throws RemoteException,
            SQLException  {
      return jrc[id].getNextRow();
      }

   // Closes the current database connection
      public void closeConnection( int id )
      throws RemoteException,
            SQLException {
      jrc[id].closeConnection();
      jrc[id] = null;
   }
}
```

The `JRImpl` object instantiates a `JRConn` object for every remote client, then slots the object into the array of `JRConn` objects, which has a limit of 100 remote client connections. In effect, the `JRImpl` class implements a basic *connection pool,* which is a way of managing multiple connections to one or more data sources. A connection ID is returned to the remote client to identify the database connection to the client. The remote methods implemented in `JRImpl` map down onto `JRConn` method calls, which implement the JDBC interface.

## Parameterising Connection Information

We may want to specify a different remote host to access a different set of services. In addition, we may want to connect to one or more remote objects—multiple databases, for example. An Informix server can connect to other Informix servers, but we may want to use the JDBC/ODBC Bridge or another JDBC driver to connect to a non-Informix database. This is especially true if the database is a non-Informix legacy database we are in the process of web enabling to participate in a larger application framework.

### Parameterising Client Applets Dynamically

If you are using an applet to connect to a remote service, you can use the `PARAM` attribute of the `APPLET` tag to nominate the remote host and object name that you use in the `Naming.lookup()` method call.

In a larger treatment of a distributed database application, you may want your secure users to connect to a different database or a different set of remote objects. In this case, you would implement a secure interface, such as NSAPI authentication, with an NSAPI-bound URL prefix and parameterize the `APPLET` tag accordingly to nominate the appropriate remote objects.

You can use an AppPage to parameterize the parameter values passed into an applet. For simplicity, assume that you have a secure URL prefix called `/catalog`. Your users are registered in the `webusers` table in the database. Each user has a nominated remote interface to one or more remote Informix databases.

```
<APPLET CODE=catalog.class WIDTH=400 HEIGHT=500>
<PARAM NAME=user
    VALUE="<?MIVAR>$REMOTE_USER<?/MIVAR>">
<?MISQL SQL="select host, db1, db2 from rmiUsers">
  <PARAM NAME=remoteHost VALUE="$1">
  <PARAM NAME=remoteObject1 VALUE="$2">
  <PARAM NAME=remoteObject2 VALUE="$3">
<?/MISQL>
</APPLET>
```

## Summary

In this chapter, we have learned how to create and deploy a distributed, multitier Java application that connects to a remote database using JDBC, and we have seen how easy it is to do this using RMI. The concept of a connection pool was introduced, and we also learned how to hide away the JDBC detail from the client and server objects by implementing a database handler.

The example presented in this chapter is a clear and simple multitiered, distributed-component application that you can use to scale up to industrial-strength, distributed Java applications.