# Chapter 11. Remote Method Invocation (RMI)



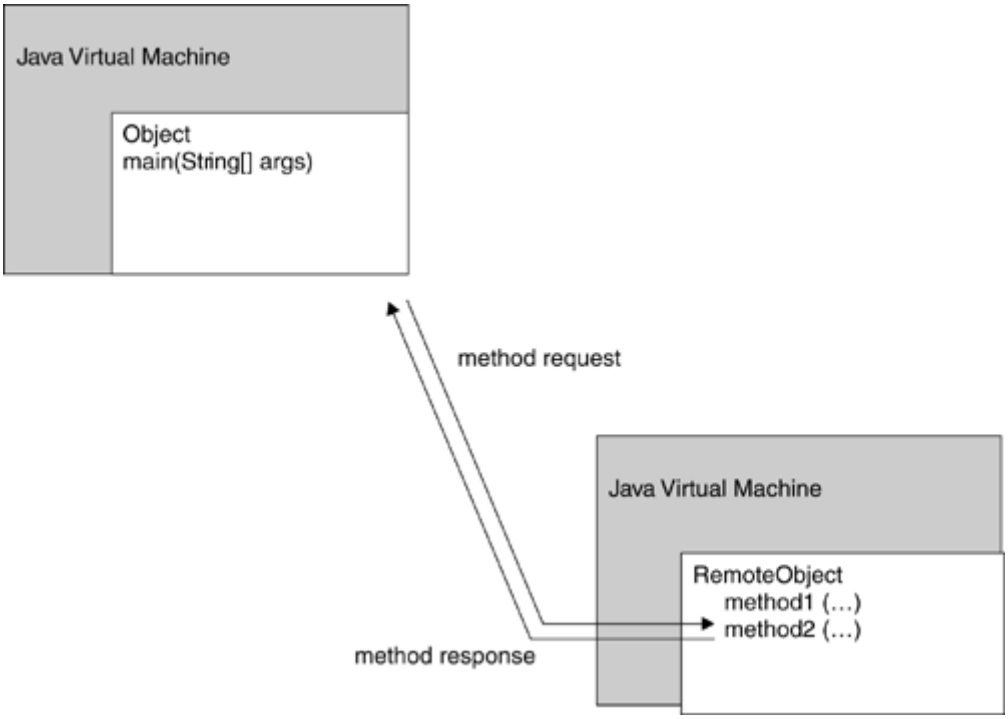
## 11.1 Overview

Remote Method Invocation (RMI) is a distributed systems technology that allows one Java Virtual Machine (JVM) to invoke object methods that will be run on another JVM located elsewhere on a network. This technology is extremely important for the development of large-scale systems, as it makes it possible to distribute resources and processing load across more than one machine.

### 11.1.1 What Is Remote Method Invocation?

RMI is a Java technology that allows one JVM to communicate with another JVM and have it execute an object method. Objects can invoke methods on other objects located remotely as easily as if they were on the local host machine (once a few initialization tasks have been performed). Figure 11-1 provides an example of this process, whereby an object running on one JVM invokes a method of an object hosted by another. Communication like this does not have to be a one-way process, either—a remote object method can return data as well as accept it as a parameter.

**Figure 11-1. Invocation of a method on a remote object, executing on a remote machine**




Each RMI service is defined by an interface, which describes object methods that can be executed remotely. This interface must be shared by all developers who will write software for that service—it acts as a blueprint for applications that will use and provide implementations of the service. More than one implementation of the interface can be created, and developers do not need to be aware of which implementation is being used or where it is located.

### 11.1.2 Comparison of Remote Method Invocation with Remote Procedure Calls

Object method invocation is not a new concept. Even before object-oriented programming, technologies existed that allowed software to call functions and procedures remotely. Systems such as remote procedure calls (RPCs) have been in use for years and continue to be used today. A popular implementation of RPC was developed by Sun Microsystems and published as RFC 1057 (making obsolete an earlier version, published as RFC 1050). Remote procedure calls were designed as a platform-neutral way of communicating between applications, regardless of any operating system or language differences.

The difference between RPC and Java RMI is subtle. Java is, after all, a platform-neutral language, and conceivably would allow Java applications to communicate with Java applications running on any hardware and operating system environment that supported a JVM. The principal difference between the two goals is that RPC supports multiple languages, whereas RMI only supports applications written in Java.

**NOTE**

With the introduction of the Java 2 Platform, Enterprise Edition, a new technology known as RMI over IIOP helps bridge the gap between RMI and CORBA systems. This technology allows for translation between RMI services and CORBA services, via the Internet Inter-ORB Protocol (more detailed information on IIOP is provided in Chapter 12). In general practice, however, RMI is used primarily with Java-only systems.

Beyond the language that either system supports, there are some fundamental differences in the way that RPC and RMI work. Remote method invocation deals with objects, and allows methods to accept and return Java objects as well as primitive datatypes. Remote procedure calls, on the other hand, do not support the notion of objects. RPC services offer procedures, which are not associated with a particular object. Messages to an RPC service are represented by the External Data Representation (XDR) language, which abstracts the differences between byte ordering and structure of datatypes. Only datatypes that are definable by XDR can be passed, and while this amounts to a large variety of primitive datatypes and structures composed of primitive datatypes, it does not allow objects to be passed.

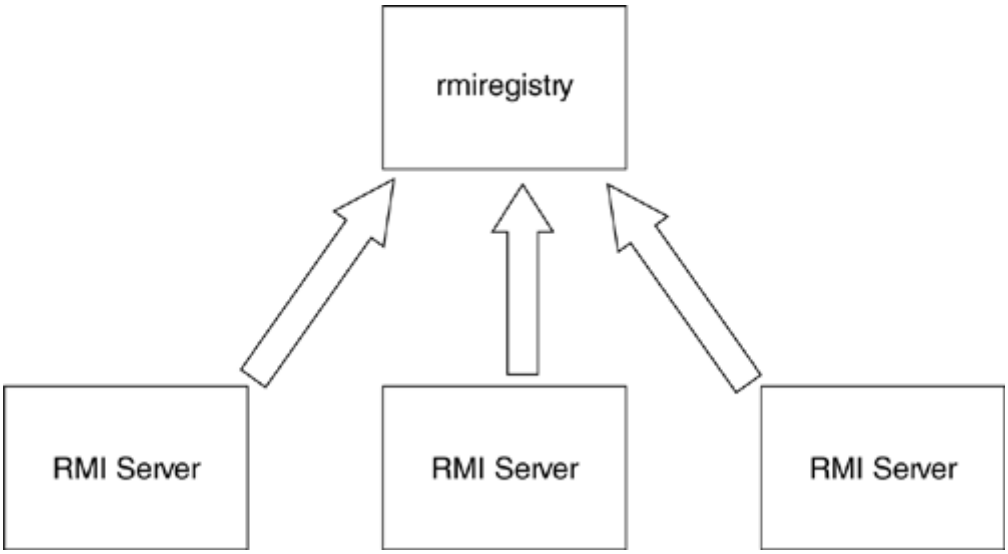
Neither system is perfect. Many RPC services already exist, and RMI is not compatible with these legacy applications. However, it is easier for Java developers to use RMI, rather than using a library that implements RPC, as services can exchange entire objects, rather than just individual data fields. By writing distributed systems in RMI, however, the ability to develop systems in other languages is lost. There are, however, distributed systems technologies that do support other languages. The most popular choice for this is the Common Object Request Broker Architecture (CORBA), which the Java 2 platform supports. CORBA has many advantages, as well as some limitations, and is discussed further in Chapter 12.

## 11.2 How Does Remote Method Invocation Work?

Systems that use RMI for communication typically are divided into two categories: clients and servers. A server provides an RMI service, and a client invokes object methods of this service.

RMI servers must register with a lookup service, to allow clients to find them, or they can make available a reference to the service in some other fashion. Included as part of the Java platform is an application called *rmiregistry,* which runs as a separate process and allows applications to register RMI services or obtain a reference to a named service. Once a server has registered, it will then wait for incoming RMI requests from clients. Figure 11-2 illustrates services registering with a single RMI registry. Associated with each service registration is a name (represented as a string), to allow clients to select the appropriate service. If a service moves from one server to another, the client need only look up the registry again to find the new location. This makes for a more fault-tolerant system—if the service is unavailable because a machine is down, a system administrator could launch a new instance of the service on another system and have it register with the RMI registry. Providing the registry remains active, you can have your servers go online and offline or move from host to host. The registry doesn't care which host a service is offered from, and clients get the service location directly from the registry.

**Figure 11-2. Multiple services can register with the same registry.**




RMI clients will send RMI messages to invoke an object method remotely. Before any remote method invocation can occur, however, the client must have a remote object reference. This is normally obtained by looking up a service in the RMI registry. The client application requests a particular service name, and receives a URL to the remote resource. Remember, URLs are not just for HTTP—most protocols can be represented using URL syntax. The following format is used by RMI for representing a remote object reference:

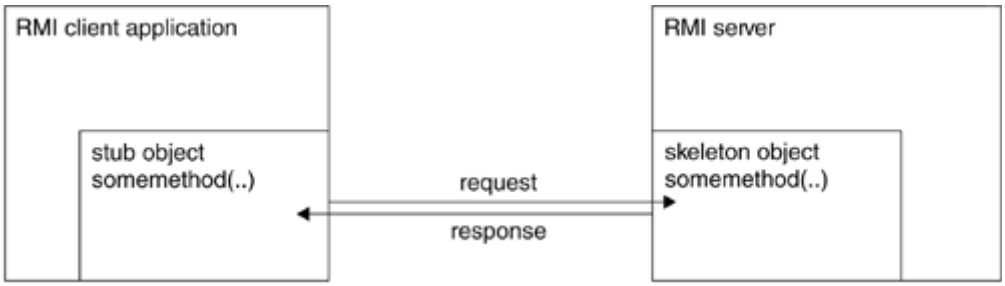
```
rmi://hostname:port/servicename
```

where `hostname` represents the name of a server (or IP address), `port` the location of the service on that machine, and `servicename` a string description of the service.

Once an object reference is obtained (either through the rmiregistry, a custom lookup service, or by reading an object reference URL from a file), the client can then interact with the remote service. The networking details of requests are completely transparent to the application developer—working with remote objects becomes as simple as working with local ones. This is achieved through a clever division of the RMI system into two components, the stub and the skeleton.

The stub object acts as a proxy object, conveying object requests to the remote RMI server. Remember that every RMI service is defined as an interface, not as an implementation. The stub object implements a particular RMI interface, which the client application can use just like any other object implementation. Rather than performing the work itself, however, the stub passes a message to a remote RMI service, waits for a response, and returns this response to the calling method. The application developer doesn't need to be concerned about where the RMI resource is located, on which platform it is running, or how it will fulfill the request. The RMI client simply invokes a method of the proxy object, which handles all the implementation details. Figure 11-3 illustrates how this is achieved; shown is an RMI client invoking an object method on the stub proxy, which conveys this request to the remote server.

**Figure 11-3. The RMI client stub calls the RMI server skeleton.**




At the RMI server end, the skeleton object is responsible for listening for incoming RMI requests and passing these on to the RMI service. The skeleton object does not provide an implementation of an RMI service, however. It only acts as a receiver for requests, and passes these requests on further. After a developer creates an RMI interface, he or she must still provide a concrete implementation of the interface. This implementation object will be called by the skeleton object, which invokes the appropriate method and passes the results back to the stub object in the RMI client. This model makes for much simpler programming, as the skeleton is separated from the actual implementation of the service. All the developer of the server needs to be concerned about is some brief initialization code (to register a service and accept requests), and providing an implementation of the RMI service interface.

With respect to the question of how messages are sent, the answer is fairly straightforward. Communication occurs between stub and skeleton using TCP sockets. The skeleton, once it is created, listens for incoming socket requests issued by stubs. Parameters in an RMI system are not limited to primitive datatypes—any object that is serializable can be passed as a parameter or returned from a remote method. When a stub passes a request to a remote skeleton, it must package the parameters (either primitive datatypes, objects, or both) for transmission, which is known as data marshalling. At the skeleton end the parameters are reconstituted to form primitive datatypes and objects, which is known as unmarshalling. For this task, specialized subclasses of the `ObjectOutputStream` and `ObjectInputStream` classes are used, to read and write the contents of objects. Parameters are normally passed by value in this case, unless the parameter is itself a reference to a remote object.

## 11.3 Defining an RMI Service Interface

Any system that uses RMI will use a service interface. The service interface defines the object methods that can be invoked remotely, and specifies parameters, return types, and exceptions that may be thrown. Stub and skeleton objects, as well as the RMI service, will implement this interface. For this reason, developers are urged to define all methods in advance, and to freeze changes to the interface once development begins.

**NOTE**

It is possible to make changes to an interface, but all clients and servers must have a new copy of the service interface, and code for the stubs and skeletons must be rebuilt.

All RMI service interfaces extend the `java.rmi.Remote` interface, which assists in identifying methods that may be executed remotely. To define a new interface for an RMI system, you must

declare a new interface extending the `Remote` interface. Only methods defined in a `java.rmi.Remote` interface (or its subclasses) may be executed remotely—other methods of an object are hidden from RMI clients.

For example, to define an interface for a remote lightbulb system (a high-tech version of the traditional on-off switch for a networked world), we could define an interface such as the following:

```
//Chapter 11, Listing 1
public interface RMILightBulb extends java.rmi.Remote
{
      public void on () throws java.rmi.RemoteException;
      public void off() throws java.rmi.RemoteException;
      public boolean isOn() throws java.rmi.RemoteException;
}
```

The interface is identified as remotely accessible, by extending from the `Remote` interface. Each method is marked as public, and may throw a `java. rmi.RemoteException`. This is important, as network errors might occur that will prevent the request from being issued or responded to. In an RMI client, a stub object that implements this interface will act as a proxy to the remote system—if the system is down, the stub will throw a `RemoteException` error that must be caught. If a method is defined as part of an RMI interface, it must be marked as able to throw a `RemoteException`—if it is not, stub and skeleton classes cannot be generated by the "rmic" tool (a tool that ships with the Java SDK which automates the generation of these classes).

Methods are not limited to throwing only a `RemoteException`, however. They may throw additional exceptions that are already defined as part of the Java API (such as an `IllegalArgumentException` to indicate bad method parameters), or custom exceptions created for a system. For example, the `on()` method could be modified to throw a `BrokenBulb` exception, if it could not be successfully activated.

## 11.4 Implementing an RMI Service Interface

Once a service interface is defined, the next step is to implement it. This implementation will provide the functionality for each of the methods, and may also define additional methods. However, only those methods defined by the RMI service interface will be accessible remotely, even if they are marked public in scope or as being able to throw a `RemoteException`.

For our lightbulb system, the following implementation could be created.

### Code for RMILightBulbImpl

```
// Chapter 11, Listing 2
public class RMILightBulbImpl
     // Extends for remote object functionality
     extends java.rmi.server.UnicastRemoteObject
     // Implements a light bulb RMI interface
     implements RMILightBulb
{
     // A constructor must be provided for the remote object
     public RMILightBulbImpl() throws java.rmi.RemoteException
     {
          // Default value of off
```

```
            setBulb(false);
        }

        // Boolean flag to maintain light bulb state information
        private boolean lightOn;
        // Remotely accessible "on" method - turns on the light
        public void on() throws java.rmi.RemoteException
        {
            // Turn bulb on
            setBulb (true);
        }

        // Remotely accessible "off" method - turns off the light
        public void off() throws java.rmi.RemoteException
        {
            // Turn bulb off
            setBulb (false);
        }

        // Remotely accessible "isOn" method, returns state of bulb
        public boolean isOn() throws java.rmi.RemoteException
        {
            return getBulb();
        }

        // Locally accessible "setBulb" method, changes state of
        // bulb
        public void setBulb (boolean value)
        {
            lightOn = value;
        }


        // Locally accessible "getBulb" method, returns state of
        // bulb
        public boolean getBulb ()
        {
            return lightOn;
        }
}
```

## How RMILightBulbImpl Works

An object that can be accessed remotely must—at a minimum—extend the `java.server.RemoteObject` class. However, a support class exists that provides all the necessary functionality for exporting an object remotely. When implementing an interface, the class should extend the `java.server.UnicastRemoteOb ject` class—this saves significant time and effort and provides for simpler code.

Extending from `UnicastRemoteObject` is the only RMI-specific code that needs to be written for a service implementation. As can be seen from the code, there is very little work that needs to be done to create RMI service implementations. Methods defined in the service interface must be implemented (or the class would fail to compile until implemented or marked abstract), but beyond that there is no actual networking code required. This means that even developers not experienced with networking could contribute to RMI systems, although writing code for a server and client requires somewhat more effort.

## 11.5 Creating Stub and Skeleton Classes

The `stub` and `skeleton` classes are responsible for dispatching and processing RMI requests. Developers should not write these classes, however. Once a service implementation exists, the rmic tool, which ships with the JDK, should be used to create them.

The implementation and interface should be compiled, and then the following typed at the command line:

```
rmic implementation
```

where `implementation` is the name of the service implementation class.

For example, if the class files for the `RMILightBulb` system are in the current directory, the following would be typed to produce stub and skeleton classes:

```
rmic RMILightBulbImpl
```

Two files would then be produced in this case:

- `RMILightBulbImpl_Stub.class`
- `RMILightBulbImpl_Skeleton.class`

RMI clients, and the RMI registry, will require these classes as well as the service interface class. They can be copied to a local file system, or distributed remotely via a Web server using dynamic class loading (discussed in Section 11.10.1).

## 11.6 Creating an RMI Server

The RMI server is responsible for creating an instance of a service implementation and then registering it with the remote method invocation registry (rmiregistry). This actually amounts to only a few lines of code, and is extremely easy to do. In small systems, the server could even be combined with the service implementation by adding a `main()` method for this purpose, though a separation of classes is a cleaner design.

### Code for LightBulbServer

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

// Chapter 11, Listing 3
public class LightBulbServer
{
          public static void main(String args[])
          {
          System.out.println ("Loading RMI service");

          try
          {
               // Load the service
               RMILightBulbImpl bulbService = new RMILightBulbImpl();

               // Examine the service, to see where it is stored
```

```
                RemoteRef location = bulbService.getRef();
                System.out.println (location.remoteToString());


                // Check to see if a registry was specified
                String registry = "localhost";
                if (args.length >=1)
                {
                     registry = args[0];
                }

                // Registration format //registry_hostname :port
/service
                // Note the :port field is optional
                String registration = "rmi://" + registry +
"/RMILightBulb";

                // Register with service so that clients can find us
                Naming.rebind( registration, bulbService );

           }
           catch (RemoteException re)
           {
                System.err.println ("Remote Error - " + re);
           }
           catch (Exception e)
           {
                System.err.println ("Error - " + e);
           }
     }
}
```

## How LightBulbServer Works

The lightbulb server defines a single method, `main`, which allows it to be run as an application. The application encloses almost all of the code in a try { .. } catch block, since networking errors that occur at runtime must be caught. The most likely error to occur is for a `RemoteException` to be thrown, if the service could not be started or if the server is unable to register with the rmiregistry.

```
public static void main(String args[])
{
     System.out.println ("Loading RMI service");
     try
     {
          ....
     }
     catch (RemoteException re)
     {
          System.err.println ("Remote Error - " + re);
     }
     catch (Exception e)
     {
          System.err.println ("Error - " + e);
     }
}
```

The next step is to create an instance of the RMI lightbulb service defined by the `RMILightBulbImpl` class. The constructor for this class takes no parameters and is simple to

invoke. Once the service has been created, the application obtains a remote reference to the newly created lightbulb service and displays its contents so that it is possible to see exactly where the service is located. Each service binds to a local TCP port through which it is located, and the reference is composed of the hostname and port of the service.

```
// Load the service
RMILightBulbImpl bulbService = new RMILightBulbImpl();

// Examine the service, to see where it is stored
RemoteRef location = bulbService.getRef();
System.out.println (location.remoteToString());
```

The final step is to register the service with the rmiregistry, so that other clients can access it. This registry could be located on any computer on the local network, or on the Internet, or it could be found on the current host. By default, the server will attempt a registration on the local machine, but if the hostname of a registry is specified on the command line, this setting will be overridden.

```
// Check to see if a registry was specified
String registry = "localhost";
if (args.length >=1)
{
     registry = args[0];
}

// Registration format //registry_hostname:port /service
// Note the :port field is optional
String registration = "rmi://" + registry + "/RMILightBulb";

// Register with service so that clients can find us
Naming.rebind( registration, bulbService );
```

The registration details include the location of the registry, followed by an optional port and then the name of the service. Using static methods of the class `java.rmi.Naming`, which is responsible for retrieving and placing remote object references in the registry, we register the service so that clients can access it. The details, represented by our registration string, are passed to the `Naming. rebind(..)` method, which accepts as a parameter a registration name and an instance of the `java.rmi.Remote` interface. We pass the service created earlier, and the service becomes bound to the registry. The binding process creates an entry in the registry for clients to locate a particular service, so that if the service is shut down, it should of course unbind itself. If no errors occur, the service is then available for remote method invocation.

## 11.7 Creating an RMI Client

Writing a client that uses an RMI service is easy compared with writing the service itself. The client needs only to obtain an object reference to the remote interface, and doesn't need to be concerned with how messages are sent or received or the location of the service. To find the service initially, a lookup in the RMI registry is made, and after that, the client can invoke methods of the service interface just as if it were a local object. The next example demonstrates how to turn a remote lightbulb on and off.

### Code for LightBulbClient

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

```
// Chapter 11, Listing 4
public class LightBulbClient
{
     public static void main(String args[])
     {
          System.out.println ("Looking for light bulb service");

          try
          {
               // Check to see if a registry was specified
               String registry = "localhost";
               if (args.length >=1)
               {
                    registry = args[0];
               }

               // Registration format //registry_hostname
(optional):port /service
               String registration = "rmi://" + registry +
"/RMILightBulb";
               // Lookup the service in the registry, and obtain
               // a remote service
               Remote remoteService = Naming.lookup ( registration );

               // Cast to a RMILightBulb interface
               RMILightBulb bulbService = (RMILightBulb)
remoteService;

               // Turn it on
               System.out.println ("Invoking bulbservice.on()");
               bulbService.on();

               // See if bulb has changed
               System.out.println ("Bulb state : " +
bulbService.isOn()  );

               // Conserve power
               System.out.println ("Invoking bulbservice.off()");
               bulbService.off();

               // See if bulb has changed
               System.out.println ("Bulb state : " +
bulbService.isOn() );
          }
          catch (NotBoundException nbe)
          {
               System.out.println (
                   "No light bulb service available in registry!");
          }
          catch (RemoteException re)
          {
               System.out.println ("RMI Error - " + re);
          }
          catch (Exception e)
          {
               System.out.println ("Error - " + e);
          }
     }
}
```

## How LightBulbClient Works

As in the lightbulb server, the client application must be careful to catch any exceptions thrown at runtime, if the server or rmiregistry is unavailable. It also needs to create a URL to the registry and service name, which by default will point to the local machine, but can be overridden by a command-line parameter. The code for this is similar to that in the server, and for this reason is not repeated here. Once a URL to the registry entry has been created, the application attempts to look up the location of the service and obtain an object reference to it, by using the `Naming.lookup(String)` method. An explicit cast is made to the `RMILightBulb` interface; if no such interface is found, a `NotBoundException` will be thrown and caught.

```
// Lookup the service in the registry, and obtain a remote
// service
Remote remoteService = Naming.lookup ( registration );

// Cast to a RMILightBulb interface
RMILightBulb bulbService = (RMILightBulb) remoteService;
```

Here the networking code ends and the service code begins. Three methods were defined in the service interface:

- `public void RMILightBulb.on()`
- `public void RMILightBulb.off()`
- `public boolean RMILightBulb.isOn()`

Each of the three methods is tested, and the changes in bulb state displayed to the user. Once this is done, the application will terminate, its task complete.

## 11.8 Running the RMI System

Running any RMI system takes a little care, as there is a precise order to the running of applications. Before the client can invoke methods, the registry must be running. Nor can the service be started before the registry, as an exception will be thrown when registration fails. Furthermore, the client, server, and registry need access to the interface, stub, and skeleton classes, which means they must be available in the class path, unless dynamic loading is used (discussed in Section 11.10). This means compiling these classes, and copying them to a directory on the local file system of both client and server (as well as the registry, if it is located elsewhere), before running them.

**NOTE**

Remember to run the rmic tool over the `RMILightBulbImpl` class, to generate stub and skeleton classes before copying the class files.

The following steps show how to run the lightbulb system, but apply to other RMI systems as well.

1. Copy all necessary files to a directory on the local file system of all clients and the server.
2. Check that the current directory is included in the classpath, or an alternate directory where the classes are located.

3. Change to the directory where the files are located, and run the rmiregistry application (no parameters are required), by invoking the following command: rmiregistry
4. In a separate console window, run the server (specifying if necessary the hostname of the machine where the rmiregistry application was run): java `LightBulbServer` hostname.
5. In a separate console window, and preferably a different machine, run the client (specifying the hostname of the machine where the rmiregistry application was run): java `LightBulbClient` hostname.

You will then see a message indicating that the bulb was activated and then switched off using RMI.

## 11.9 Remote Method Invocation Packages and Classes

Now that you're familiar with the basics of RMI and how to write an RMI service, server, and client, let's look at the packages and classes that comprise the RMI subset of the Java API.

### 11.9.1 Packages Relating to Remote Method Invocation

Five packages deal with RMI. The two most commonly used are `java.rmi` and `java.rmi.server`, but it's important to be aware of the functionality provided by the remaining three.

1. `java.rmi`— defines the Remote interface, classes used in RMI, and a number of exceptions that can occur at runtime.
2. `java.rmi.activation`— introduced in the Java 2 platform, this package supports "activation" of remote services. Activation allows a service to be started on demand, rather than running continually and consuming system resources.
3. `java.rmi.dgc`— provides an interface and two classes to support distributed garbage collection. Just as objects can be garbage-collected by the JVM, distributed objects may be collected when clients no longer maintain a reference to them.
4. `java.rmi.registry`— provides an interface to represent an RMI registry and a class to locate an existing registry or launch a new one.
5. `java.rmi.server`— provides interfaces and classes related to RMI servers, as well as a number of server-specific exceptions that may be thrown at runtime.

#### 11.9.1.1 Package java.rmi

There are one interface, three classes, and a large number of exception classes defined by this package. Each is discussed separately.

##### Remote Interface

The `java.rmi.Remote` interface is unusual, in that it does not define any methods for implementing classes. It is, instead, used as a means of identifying a remote service. Every RMI service interface will extend the Remote interface. Being an interface, it cannot be instantiated (only a class that implements an interface may be instantiated). However, an implementing class may be cast to a Remote instance at runtime.

When creating a service that extends the `java.rmi.Remote` interface, you should be aware that methods must declare a throws clause, listing at least `java.rmi.RemoteException`, and (optionally) any application-specific exceptions. This is not declared in the API documentation for the Remote interface, but is a condition imposed by the rmic tool. Failure to adhere to this