Client Server Application Programming

Week 11: Remote Method Invocation(Creating an RMI Server, Creating an RMI Client, Running the RMI System, Remote Method Invocation Packages and Classes)

By Elubu Joseph - MSc.IS
Lecturer
Department of Information Technology
Kumi University
Email: josebulinda@gmail.com
jose@kumiuniversity.ac.ug

Summary of Previous Lecture

1. Overview of Remote Method Invocation
2. How Remote Method Invocation Work,
3. Defining a RMI Service Interface,
4. Implementing an RMI Service Interface,
5. Creating Stub and Skeleton Classes

Agenda

1. Recap on Creating Stub and Skeleton Classes
2. Creating an RMI Server,
3. Creating an RMI Client,
4. Running the RMI System,
5. Remote Method Invocation Packages and Classes

A recap Creating Stub and Skeleton Classes

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 of an 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.

Creating Stub and Skeleton Classes+

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:

1. `RMILightBulbImpl_Stub.class`
2. `RMILightBulbImpl_Skeleton.class`

This procedure is good for those compiling their programs using cmd.

Creating Stub and Skeleton Classes+

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 earlier)

Creating Stub and Skeleton Classes in NetBeans IDE

Starting from JDK 5, the stub and skeleton classes are no longer required to be explicitly generated for RMI. The Java compiler automatically generates them when you compile your RMI-related classes. Therefore, in NetBeans 13 or any recent version, you don't need to manually generate the stub and skeleton classes.

Here's what you need to do to create an RMI application in NetBeans as long as you are running JDK5+:

1. Create a new Java project or open an existing one in NetBeans.
2. Create your remote interface and implementation classes. The remote interface should extend java.rmi.Remote.

Creating Stub and Skeleton Classes in NetBeans IDE+

3. Annotate your remote interface and any remote methods with java.rmi.RemoteException (optional but recommended for better exception handling).

4. Compile your RMI classes by building the project. Right-click on your project and select "Build" or use the keyboard shortcut "F11". This will compile your RMI classes and generate the necessary stub and skeleton classes automatically.

5. Once the build is successful, you can proceed with implementing your RMI server and client logic.

Creating Stub and Skeleton Classes in NetBeans IDE+

NetBeans takes care of generating the stub and skeleton classes behind the scenes, so you don't need to worry about manually invoking the rmic tool or explicitly generating these classes.

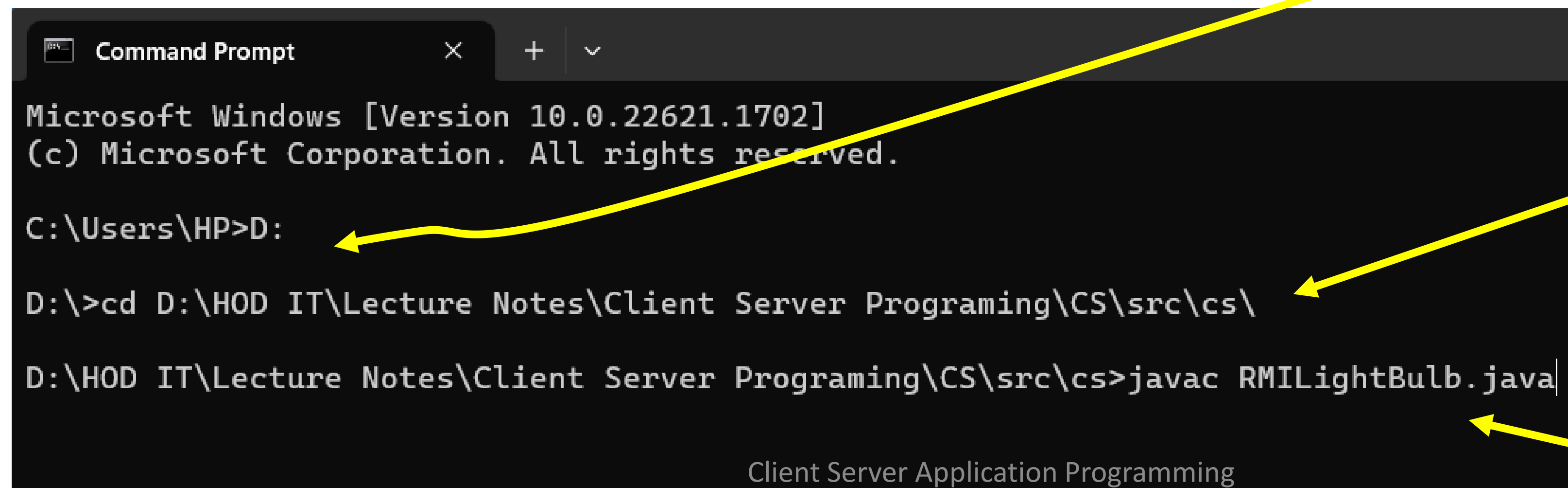
Remember to configure your project's build path and dependencies properly, including the required RMI libraries, to ensure smooth execution of your RMI application.

Note: If you're using an older version of Java or have specific requirements for generating stub and skeleton classes, you may need to use the **rmic tool** or consider an alternative approach.

Creating Stub and Skeleton Classes in NetBeans IDE+ special note.

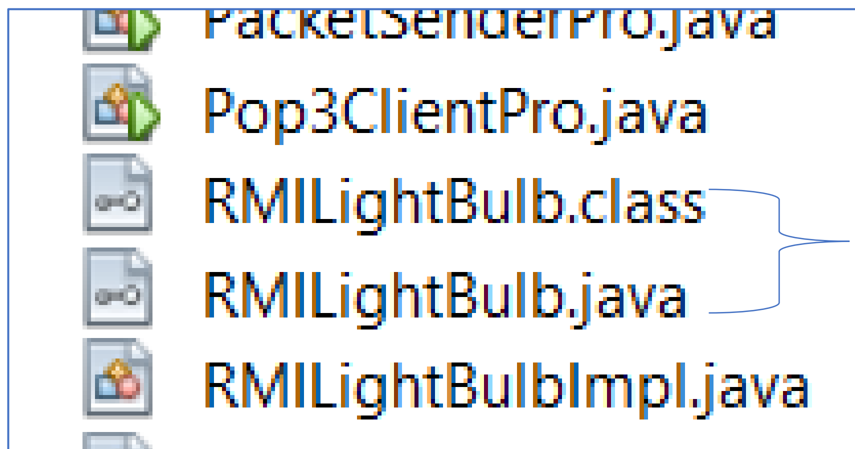
Although NetBeans automatically Compiles class for us, you may need to compile your interface manually through the following steps.

1. Open cmd
2. Change Directory to where your application files are using **cd** command
3. Type **javac** then name of your file. Eg.



Creating Stub and Skeleton Classes in NetBeans IDE+ special note.

After pressing Enter key, you should be able to see a File with the same name as the one you were compiling but with **.class** extension.



RMILightBulb.java has been compiled and produced RMILightBulb.class

Creating an RMI Server

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.

Creating an RMI Server+
LightBulbServer class

Our RMI server below is represented by a **class** called **LightBulbServer**

NOTE: We will be referring to the **service interface** file **RMILightBulb.java** and the Servant Class called **RMILightBulbImpl.java** we created before. Lets remind our selves.

RMILightBulb.java

```
2    package cs;
3    import java.rmi.RemoteException;
4    import java.rmi.Remote;
     public interface RMILightBulb extends Remote{
             public void on () throws RemoteException;
             public void off() throws RemoteException;
             public boolean isOn() throws RemoteException;
9     }
```

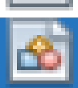
Note the declaration of three methods

Reminder on Servant class

RMILightBulbImpl.java

```
1  package cs; import java.rmi.RemoteException;
2  import java.rmi.server.UnicastRemoteObject;
3  public class RMILightBulbImpl extends UnicastRemoteObject implements
4          RMILightBulb {
5   // A constructor must be provided for the remote object
6      public RMILightBulbImpl() throws RemoteException {
7          // Default value of off
           setBulb(false);
9      }
0      // Boolean flag to maintain light bulb state information
1      private boolean lightOn;
2      // Remotely accessible "on" method - turns on the light
       public void on() throws RemoteException {
4          // Turn bulb on
5          setBulb (true);
6      }
```

Reminder on Servant class



RMILightBulbImpl.java

```
17        // Remotely accessible "off" method - turns off the light
18        public void off() throws RemoteException{
19            // Turn bulb off
20            setBulb (false);
21        }
22        // Remotely accessible "isOn" method, returns state of bulb
23        public boolean isOn() throws RemoteException{
24            return getBulb();
25        }
26
27        // Locally accessible "setBulb" method, changes state of bulb
28        public void setBulb (boolean value){
29            lightOn = value;
30        }
31        // Locally accessible "getBulb" method, returns state of bulb
32        public boolean getBulb (){
33            return lightOn;
34        }
35    }
```

This class has implementation of the three remote methods; **on(), off() and isOn(),** accessed from the service interface

and two local methods; setBulb() and getBulb() making them Five methods.

Create the Server++

Now that we have reminded our selvels on the interface and class we created earlier, let go ahead to create the LightBulbServer.java.

Creating an RMI Server+

LightBulbServer class

Having noted that lets now create the server class called **LightBulbServer**

```
1    package cs;
2    import java.rmi.*;  import java.rmi.registry.LocateRegistry;
3    public class LightBulbServer{
4        public static void main(String args[])throws RemoteException,
5                AlreadyBoundException {
6            System.out.println ("Loading RMI service");
7            try{
8                // Load the service
9                RMILightBulbImpl bulbService = new RMILightBulbImpl();
10
11               // Start the RMI registry on port 1099
12                 LocateRegistry.createRegistry(1099);
```

Creating an RMI Server+

LightBulbServer class

```
14              // Register with service so that clients can find us
15              Naming.rebind("rmi://localhost:1099/RMILightBulb", bulbService);
16              System.out.println("RMILightBulb service registered and running.");
17          }
18          catch (RemoteException re){
19              System.err.println ("Remote Error - " + re);
20          }
            catch (Exception e){
22              System.err.println ("Error - " + e);
23          }
24      }
25  }
```

End of the code

Creating an RMI Server+

LightBulbServer class code Explained

The code provided above is for a server-side implementation of an RMI (Remote Method Invocation) service for a light bulb. Here's a summary of how the code works:

1. The **LightBulbServer** class is defined with a main method.
2. The main method handles exceptions of type **RemoteException** and **AlreadyBoundException**.
3. The code starts by printing a message indicating that the RMI service is being loaded.
4. Inside a **try-catch block**, the RMI service implementation class **RMILightBulbImpl** is instantiated.

Creating an RMI Server+

LightBulbServer class code Explained+

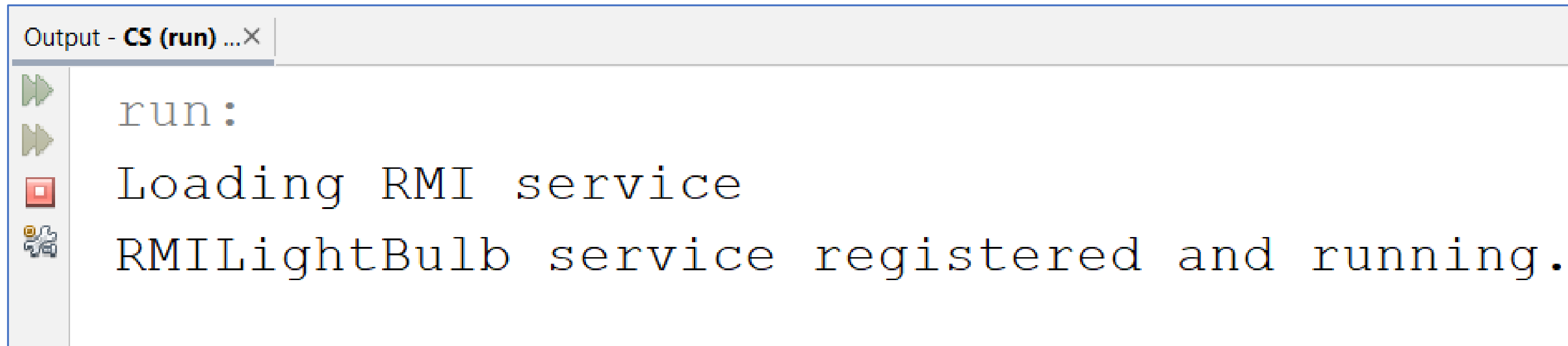
5. The code then starts the RMI registry on port 1099 using **LocateRegistry.createRegistry(1099).** This is the default port for RMI.
6. The **Naming.rebind()** method is used to bind the RMI service implementation object to a specific name in the RMI registry. In this case, it is bound to the name **"rmi://localhost:1099/RMILightBulb".**
7. If the above steps are successful, a message is printed indicating that the **RMILightBulb** service has been registered and is running.
8. Exceptions of type **RemoteException** and Exception are caught separately, and their error messages are printed.

In summary, this code sets up and registers the RMI service implementation (the **RMILightBulbImpl** object) with a specific name in the RMI registry. Clients can then use the RMI registry to locate and invoke the methods provided by the **RMILightBulbImpl** object remotely.

Creating an RMI Server+

LightBulbServer class code output

Below is the output of the server code above.



Enter it into your NetBeans to test.

Creating an RMI Client

Creating an RMI Client+ **LightBulbClient** class

Having successfully created the RMI Implementation Server class above, we can now go a head and create the client side for the implementation of the RMI Applications. Our Class name will be **LightBulbClient.java**

Creating an RMI Client+ **LightBulbClient** class

```
1   package cs;
2   import java.net.MalformedURLException; import java.rmi.Naming;
3   import java.rmi.NotBoundException; import java.rmi.RemoteException;
4
5   public class LightBulbClient{
6   public static void main(String args[])throws NotBoundException,
7               MalformedURLException,RemoteException{
8       System.out.println ("Looking for light bulb service");
9       try{
10     // Check to see if a registry was specified
11      String hostname = "localhost";
12      // Lookup the service in the registry, and obtain a remote service
13     RMILightBulb remoteService = (RMILightBulb)Naming.lookup("rmi://"
14             +hostname+":1099/RMILightBulb");
15      // Cast to a RMILightBulb interface
16      RMILightBulb bulbService = (RMILightBulb)remoteService;
```

Creating an RMI Client
LightBulbClient class

```
17          // Turn it on
18          System.out.println ("Invoking bulbservice.on()");
19          bulbService.on();
20          // See if bulb has changed
21          System.out.println ("Bulb state : " + bulbService.isOn() );
22          // Conserve power
23          System.out.println ("Invoking bulbservice.off()");
24          bulbService.off();
25          // See if bulb has changed
26          System.out.println ("Bulb state : " + remoteService.isOn() );
27          }
28          catch (NotBoundException nbe){
29          System.out.println ("No light bulb service available in registry!");
30          }
```

Creating an RMI Client+
LightBulbClient class++

```
31          catch (RemoteException re){
32              System.out.println ("RMI Error - " + re);
33          }
            catch (Exception e){
35              System.out.println ("Error - " + e);
36          }
37          }
38      }
```

End of the program

Creating an RMI Client+
LightBulbClient class code explained

The code provided above is for a client-side implementation of an RMI (Remote Method Invocation) client that interacts with the light bulb service.

Here's an explanation of how the code works:

1. The **LightBulbClient** class is defined with a main method.
2. The main method handles exceptions of type **NotBoundException**, **MalformedURLException**, and **RemoteException**.
3. The code starts by printing a message indicating that it is **looking for the light bulb service.**

Creating an RMI Client+
LightBulbClient class code explained+

The code provided above is for a client-side implementation of an RMI (Remote Method Invocation) client that interacts with the light bulb service.

Here's an explanation of how the code works:

1. The **LightBulbClient** class is defined with a main method.
2. The main method handles exceptions of type **NotBoundException**, **MalformedURLException**, and **RemoteException**.
3. The code starts by printing a message indicating that it is **looking for the light bulb service.**

Creating an RMI Client+
LightBulbClient class code explained++

4. Inside **a try-catch block**, the following steps are performed:
   i. A string variable **hostname** is assigned the value "localhost". This specifies the host where the RMI registry is running.
   ii. The **Naming.lookup()** method is used to look up the RMI service in the registry. The URL **"rmi://localhost:1099/RMILightBulb"** is used to specify the location of the service. This URL includes the hostname, **port number (1099),** and the name under which the service was registered.
   iii. The returned object is cast to the **RMILightBulb** interface type and assigned to the **bulbService** variable.
   iv. The **bulbService.on()** method is invoked to turn on the light bulb.

Creating an RMI Client+
LightBulbClient class code explained+++

v. The current state of the light bulb is obtained by calling the **bulbService.isOn()** method, and the result is printed.

vi. The **bulbService.off()** method is invoked to turn off the light bulb.

vii. The state of the light bulb is obtained again by calling **remoteService.isOn(),** where **remoteService** is the original returned object from the RMI registry. The result is printed.

5. Exceptions of type **NotBoundException**, **RemoteException**, and **Exception** are caught separately, and their error messages are printed.

In summary, this code represents a client that connects to the RMI registry, looks up the light bulb service by its registered name, and then invokes methods on the obtained service object. It demonstrates turning the light bulb on and off, as well as retrieving and printing its current state.

Running the RMI System

Running the RMI System Steps

When Running any RMI system, one needs to take a little care, as there is a precise order to the running of applications.

Note!

1. Before the client can invoke methods, the registry must be running.
2. the service can not be started before the registry, as an exception will be thrown when registration fails.
3. Furthermore, the client, server, and registry need access to the service interface(must have been compiled), stub, and skeleton classes, which means they must be available in the class path, unless dynamic loading is used (discussed in earlier). 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

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

Running the RMI System

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.

Creating an RMI Client+
LightBulbClient class and code Output

LightBulbServer output

Output × CS (run) × CS (run) #2 ×

```
run:
Loading RMI service
RMILightBulb service registered and running.
```

LightBulbClient output

Output × CS (run) × CS (run) #2 ×

```
run:
Looking for light bulb service
Invoking bulbservice.on()
Bulb state : true
Invoking bulbservice.off()
Bulb state : false
```

Remote Method Invocation Packages and Classes

Remote Method Invocation Packages and Classes

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

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.

Remote Method Invocation Packages and Classes

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

Packages Relating to Remote Method Invocation

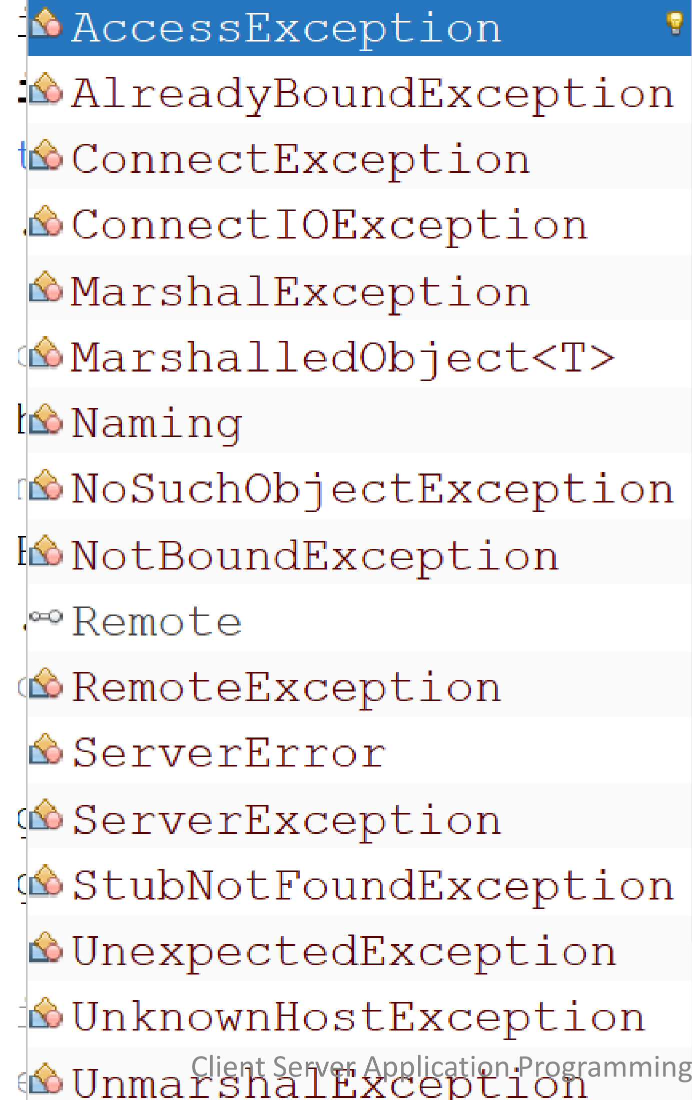
Five packages deal with RMI. The two most commonly used are java.rmi and java.rmi.server, but it's important to be aware of the functionality provided by the remaining three.

Packages Relating to Remote Method Invocation

1. `java.rmi`— defines the Remote interface, classes used in RMI, and a number of exceptions that can occur at runtime.
2. `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.
3. `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.
4. `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.
5. `java.rmi.registry`— provides an interface to represent an RMI registry and a class to locate an existing registry or launch a new one.

A glance on java.rmi Package

This package has one Interface(Remote) and a number of classes as shown below.



java.rmi.*;

All these are classes and interface of rmi package, accessable by typing **import java.rmi.** In your class file.

A glance on java.rmi Package+
Remote Interface

The java.rmi.Remote interface is unusual kind of interface, 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 condition will cause an error when stub and skeleton classes are generated.

A glance on java.rmi Package+

MarshalledObject and Naming Classes

MarshalledObject Class

This class represents a serialized object, using the same mechanism for serialization as used in marshalling and unmarshalling parameters and return values. Objects serialized in this way have the codebase of their class definition files annotated, so that they can be dynamically loaded by RMI clients or servers.

Naming Class

The Naming class offers static methods to assign or retrieve object references of the RMI object registry (rmiregistry). Each method takes a parameter as Java string, representing a URL to a registry entry. The format of these entries is as follows:

```
rmi://registry_hostname:registry_port/servicename
```

where **registry_hostname** is the hostname or IP address of a rmiregistry, **registry_port** is an optional port field for nonstandard registry locations, and **servicename** is the name of an RMI service. The default port for an rmiregistry **1099**.

A glance on java.rmi Package+

AccessException and AlreadyBoundException Classes

AccessException Class

An AccessException may be thrown by the Naming class to indicate that a bind, rebind, lookup, or unbind operation cannot be performed. This class extends the java.rmi.RemoteException class.

AlreadyBoundException Class

If a remote object is already bound to a registry entry, or if another object has already bound that service name, then AlreadyBoundException class is used to indicate that the bind operation could not proceed. The RMI service could unbind and attempt the bind operation again, or it could use the rebind method of the Naming class. This class extends the generic java.lang.Exception class

A glance on java.rmi Package+

ConnectException, ConnectIOException and MarshalException

Classes

ConnectException Class

This exception class represents an inability to connect to a remote service, such as a registry. This class extends the java.rmi.RemoteException class.

ConnectIOException Class

Similar to the ConnectException class, this represents an inability to connect to a service to execute a remote method call. This class extends the java.rmi. RemoteException class.

MarshalException Class

This class represents an error that has occurred during the marshalling of parameters for a remote method call, or when sending a return value.

For example, if the server connection is disconnected while writing arguments, this may be thrown.

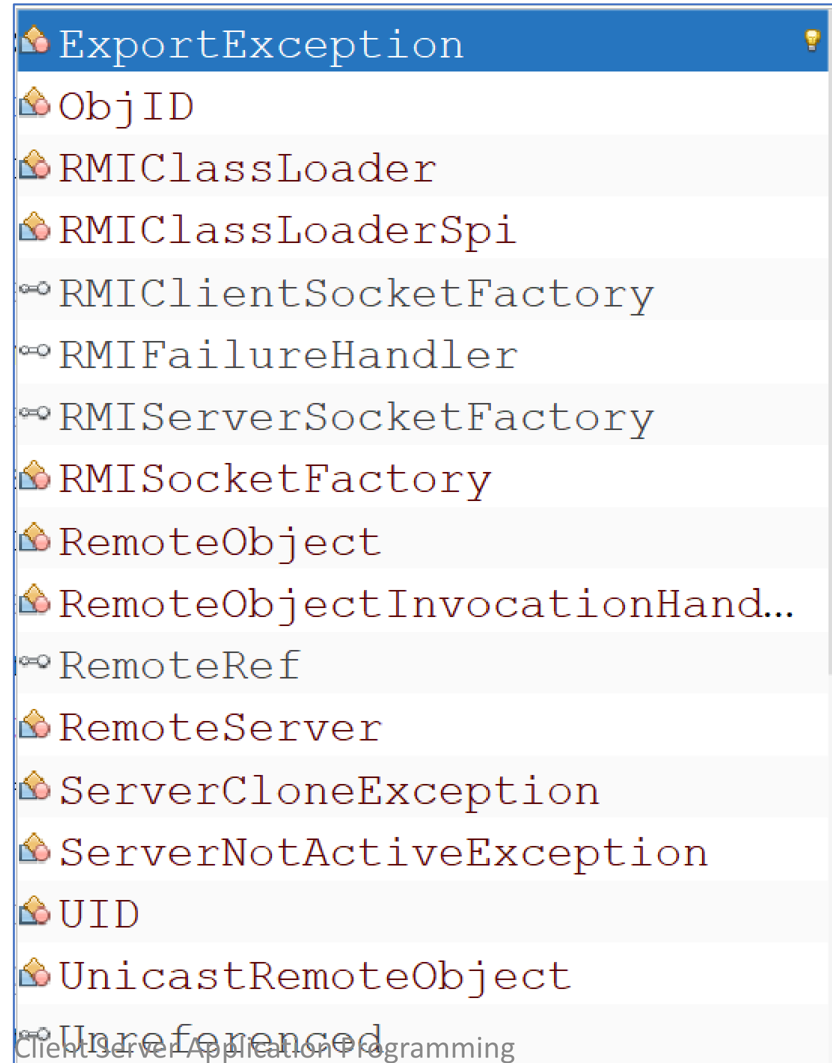
A glance on java.rmi Package+ Assignment

In your own ways, take a look at the functionally of the following classes of the rmi package.

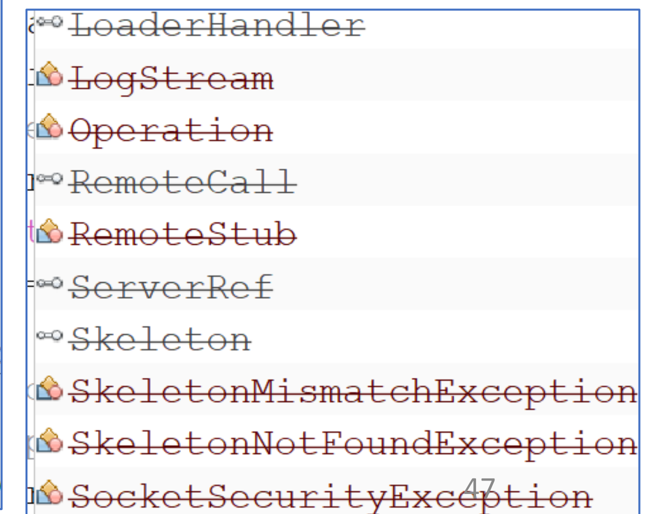
1. NotBoundException Class
2. RemoteException Class
3. ServerError Class
4. ServerException Class
5. StubNotFoundException Class
6. UnexpectedException Class
7. UnknownHostException Class
8. UnmarshalException Class

java.rmi.server Package

defines a number of interfaces, objects, and exceptions relating to RMI servers and remote objects. While a number of useful classes are defined, developers do not normally use most of the classes in this package in the production of RMI services. For this reason, only the most important classes are discussed herein.



Classes and Interfaces deprecated Include:



java.rmi.server Package classes and Interfaces

RemoteRef Interface

The remote reference interface is a handle to the remote object. Stubs can use a remote reference to issue method requests and to compare two remote objects for equality.

RMIClientSocketFactory Interface

acts as a factory for producing client sockets. Classes that provide alternate types of socket communication for sending RMI requests should implement this interface.

For example, in a special network environment where TCP communication is restricted by a firewall, or where for security reasons encrypted sockets must be used, a custom socket factory could be produced, so that normal TCP sockets are replaced. Under normal circumstances, developers will not create custom sockets and socket factories.

java.rmi.server Package+ RMIClientSocketFactory Interface Method

The `RMIClientSocketFactory` defines a single method:

`public Socket createSocket ( String host, int port )` throws `java. io.IOException`— returns a `Socket` instance, connected to the specified host and port. If a connection can't be established, an `IOException` will be thrown.

java.rmi.server Package classes and Interfaces+

RMIServerSocketFactory Interface

RMIServerSocketFactory Interface

acts as a factory for producing server sockets. Classes that provide alternate types of socket communication for listening for RMI requests should implement this interface.

For example, in a high-security network system, or a system that uses an insecure network connection such as a public communications network, encryption of sockets may be required. A custom server socket might automatically decrypt communications transparently to the developer, to allow encrypted RMI requests to be sent.

Methods

The `RMIClientSocketFactory` also defines a single method:

`public Socket createServerSocket ( int port )` throws `java.io.IOException`—returns a custom `ServerSocket` instance, bound to the specified port. If the service can't bind to that port, an `IOException` will be thrown.

java.rmi.server Package classes and Interfaces+

RemoteObject Class

The `RemoteObject` class implements the `java.rmi.Remote` interface, and provides a base template for all remote object implementations. It overrides several important methods defined in the `java.lang.Object` class, making them "remote" aware.

Methods

The `RemoteObject` class provides implementations for the equals, `hashCode()`, and `toString()` methods, as well as the following new methods (both of which are public):

1. `RemoteRef` **`getRef()`**— returns a remote reference for this object.
2. `static Remote` **`toStub( Remote obj )`** throws `java.rmi.NoSuchObjectException`— returns a stub for the specified remote object. If invoked before the object has been exported, a `NoSuchObjectException` will be thrown.

java.rmi.server Package classes and Interfaces+

Other three common classes

The other three commonly used classes of the java.rmi.server package include;

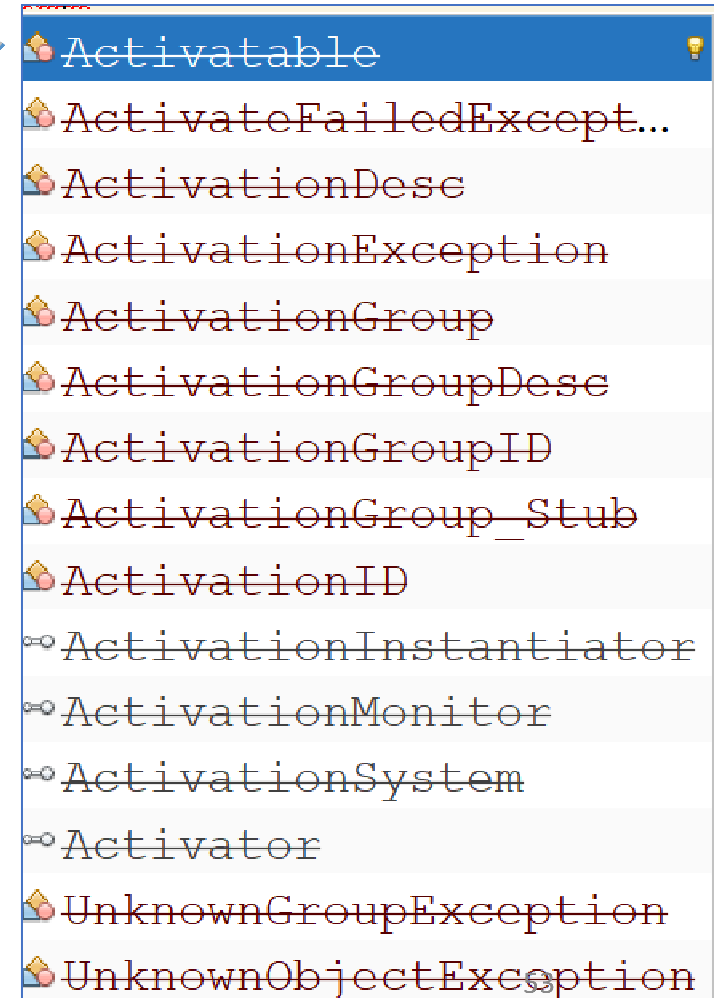
1. **RemoteServer Class**
2. **RMISocketFactory Class**
3. **UnicastRemoteObject Class**

Find out more details about them.

java.rmi.activation Package

The java.rmi.activation is a java.rmi sub package that supports remote object activation, an advanced topic that we may wish to come back to it at a later time. However this package has a number of classes and interfaces shown below, that you may not want to explore their functionalities since they are dreprecated.

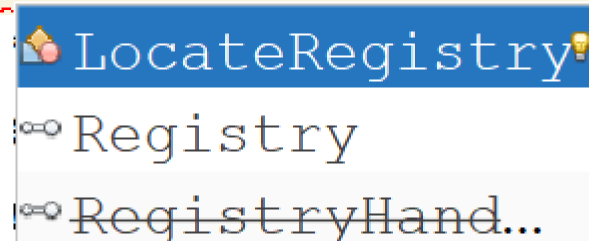
```
import java.rmi.activation.
```



java.rmi.registry Package

defines an interface (Registry) for accessing a registry service, and the `LocateRegistry` class, which can create a new RMI registry (if the rmiregistry application has not been run), as well as locating an existing registry.

```
import java.rmi.registry.
```



Registry Interface

defines methods for accessing, creating, or modifying registry entries of an RMI object registry. Similar method signatures are used for the `java.rmi.Naming` class, but there are some small changes.

The `java.rmi. Naming` class is designed to support registry entries on any remote registry, whereas the `Registry` interface deals with a specific registry.

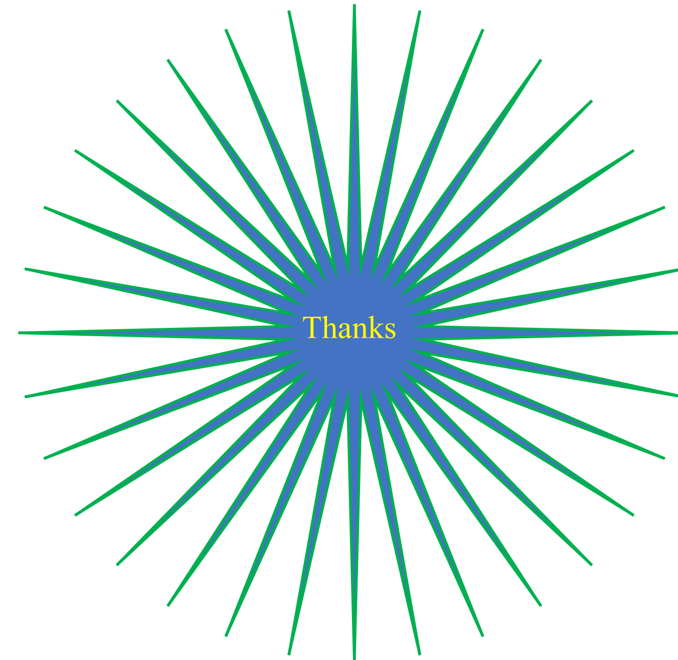
This means that some methods don't require as many parameters, and others don't require a hostname and port to be specified in the URL for the registry entry.

Summary

Summary

1. Recap on Creating Stub and Skeleton Classes
2. Creating an RMI Server(LightBulbServer.java),
3. Creating an RMI Client(LightBulbClient.java),
4. Running the RMI System,
5. Remote Method Invocation Packages and Classes

Thank you for Listening



References

Java™ Network Programming and Distributed Computing, (David R.,Michael R. 2002), Publisher : Addison Wesley; ISBN: 0201710374