Java RMI Tutorial

In this lab exercise you are required to build a simple networked and distributed application using the Java Remote Method Invocation technology. The application should return the date on the server computer to any client that requests it.

Building a distributed application using RMI involves six steps.

1. Define a remote interface
2. Implement the remote interface and the server
3. Develop a client that uses the remote interface
4. Generate stubs and skeletons
5. Start the RMI registry
6. Run the server and the client

We'll examine each step as we develop our sample distributed application. For our example we'll develop a small arithmetic server that will read in two arrays of integers and perform some operations on these arrays.

Step 1: Define a remote interface
A remote interface definition specifies the methods offered by the server to each of its clients. By looking at a remote interface the programmer of a client will be able to tell what methods the server provides and how to call them. The interface includes the names of the methods and the types of their parameters, together these are called the **method signature.**

To indicate that an object in an interface is actually a remote object, the object must implement a remote interface. A remote interface has the following characteristics:

- The remote interface must be declared public
- The remote interface extends **`java.rmi.Remote`**
- Each method declared in the remote interface must declare **`java.rmi.RemoteException`** in its throws clause.

The remote interface for example is shown below; notice how all the above characteristics are implemented. The interface contains just one method, **`add`**, which returns an array of integers to the caller.

```
// Arith.java

public interface Arith extends java.rmi.Remote
{
    int[] add(int a[], int b[]) throws java.rmi.RemoteException;
}
```

Exercise:

- ***Type in the given code and save the program, and then compile our remote interface.***

Step 2: Implementing the remote interface

The second step in the development cycle of a distributed application using RMI is implementing the remote interface that we just defined in step 1. This is done by writing a class that implements the above interface. The implementation class needs to do the following.

- Specify the remote interface being implemented
- Define the constructor for the remote object
- Implement the methods that can be invoked remotely.
- Create an instance of the security manager and install it
- Create one (or more) instance of a remote object
- Register one (or more) remote objects with the RMI registry.

The implementation of the remote interface for the arithmetic server is shown below.

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

// Server.java
public class ArithImpl extends UnicastRemoteObject implements Arith
{
    private String name;

    public ArithImpl(String s) throws RemoteException
    {  name = s;    }

    public int[] add(int a[], int b[]) throws RemoteException
    {
       int c[] = new int[10];
       for (int i=0; i<10; i++)
          c[i] = a[i] + b[i];
       return c;
    }

    public static void main(String argv[])
    {
       System.setSecurityManager(new RMISecurityManager());

       try
       {
          ArithImpl obj = new ArithImpl("ArithServer");
          Naming.rebind("//localhost/ArithServer", obj);
          System.out.println("ArithServer bound in registry");
       }
       catch (Exception e)
       {
          System.out.println("ArithImpl err: " + e.getMessage());
          e.printStackTrace();
       }
    }
}
```

Exercise:

- ***Type in the given code and save the program, and then compile.***

- **In the previous step I outlined 6 required features, look at the list again and highlight where each feature is implemented in the given code.**

Step 3: Developing a Client that uses the remote interface

In this step we'll develop a client that remotely invokes any of the remote interfaces methods. In this case we only have one, ***add***. The code shows a sample implementation of a client that uses the add method of the remote interface.

```
// ArithApp.java

import java.rmi.*;
import java.net.*;

public class ArithApp
{

    public static void main(String argv[])
    {
       int a[] = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
       int b[] = {2, 2, 2, 2, 2, 2, 2, 2, 2, 2};
       int result[] = new int[10];

       try
       {
       // replace hostname by the machine the server is running on
          Arith obj = (Arith)Naming.lookup("//hostnam/ArithServer");
          result = obj.add(a, b);
       }

       catch (Exception e)
       {
          System.out.println("ArithApp exception:"+e.getMessage());
          e.printStackTrace();
       }

       System.out.println("The sum of the arrays is: ");

       for (int j=0; j<10; j++)
          System.out.print(result[j]+"    ");

       System.out.println();
    }
}
```

Exercise:

- ***Type in the given code and save the program, and then compile.***

Step 4: Generating stubs and Skeletons
Once all of the code has been written we are ready to generate the stubs and skeletons. The stubs and skeletons are used to connect the client and server together.

RMI stubs and skeletons are generated from the command line using the rmic compiler, which is of the form: **`rmic <classname>`**

So for use we would use the command**: `rmic ArithImpl`**

Exercise:

- ***Type in the given command on the command line.***

This command will generate two files, ArthImpl_Skel.class (server end) and ArithImpl_stub.class (Client end)

Step 5: Starting the RMI Registry

You're done with development at this point; you have built all the code you need to for the application. Now you're setting up the environment so that you can run it.

The RMI registry is a naming service that allows clients to obtain a reference to a remote object. rmiregistry is a program that comes with the JDK; you can find it in the "bin" directory of your JDK installation. Under Windows, you can simply say "start rmiregistry" on the command line, which will cause the RMI registry to be started in its own DOS window. The RMI registry must be started before you can start your server.

Exercise:

- ***Type in the given command on the command line.***

Step 6: Starting the RMI Registry
Once the RMI registry is running and everything else is in place, we can fire up our server and client applications.

Exercise:

- ***Run the server (ArithImpl) and then the Client (ArithApp). What is the output?***
- ***Get the program to run across a network connection***
- ***Add in an extra method, subtract, that will subtract the second array from the first.***
- ***Connect with multiple clients to a single sever***

RMI Security

The following problem may happen when running the server:

```
C:\test3>java myRMIServer
Exception occurred: java.security.AccessControlException: access denied
(java.net.SocketPermission 127.0.0.1:1099 connect,resolve)
```

Or you may get the following error when running the client:

```
C:\test3>java myRMIClient 127.0.0.1
Exception occured: java.security.AccessControlException: access denied (java.net.SocketPermission
127.0.0.1:1099 connect,resolve)
```

The solution to this problem is to modify your security policy to allow the activities to take place. A full explanation is available on the Sun MicroSystems RMI tutorial page.

In summary, you'll need to do the following:

1. Create a new security policy file.
   For example to allow everything to happen:

   ```
   grant {
       permission java.security.AllPermission;
   };
   ```

   Save this code to a file called ***mysecurity.policy***.

2. When you run the client or the server, pass the location of your new security policy file in as an argument. This allows you to run under a new policy without having to modify your system policy files.

You can then **run the server** with the command line

```
java -Djava.security.policy=c:\myfolder\mysecurity.policy myRMIServer
```

and **run the client** with

```
java -Djava.security.policy=c:\myfolder\mysecurity.policy myRMIClient 127.0.0.1
```

You' replace ***c:\myfolder\mysecurity.policy*** with the path to your own properties file.