An Introduction To RMI With Java

Remote Method Invocation (or RMI for short) allows us to execute methods on remote servers. In this article Nevile introduces us to RMI with a simple example and fundamentals in Java.RMI is the acronym for Remote Method Invocation. As the name suggests, it helps you locate and execute methods of remote objects. It's like placing a class on Machine A and calling methods of that class from Machine B as though they were from the same machine. Confused? Well if you are new to the concepts of enterprise programming then it would take you some time to get this concept.

Rest assured however, as I have written this article to ease you into the methodologies and concepts behind RMI. This article will give you more than the facts or RMI. What I mean by this is that after you finish reading this article you will have an actual working demo of RMI on your machine.

I can tell you from my personal experience that although RMI isn't the only enterprise level solution for accessing objects remotely, it is the easiest solution to implement. RMI is a pure Java solution unlike CORBA where we can have objects from different programming languages interacting. In RMI everything you code will be in JAVA.

Before reading this article I will assume that you know your JAVA basics quite well. You could continue to read on to learn more about RMI, however knowing JAVA would be an added advantage as this article focuses heavily on JAVA coding examples to explain RMI.

The Concept

As I mentioned earlier, RMI is a mechanism through which we can call remote methods as though they were residing on your own machine. This whole process will look transparent to the end user: if I showed you a live demo and didn't tell you it was RMI, then you wouldn't have even realized the method is actually being called from a different machine. Of course, a Java Virtual Machine must be present on both the machines.

Objects which have to be made available to other machines have to be exported to something called a Remote Registry Server so that they can be invoked. So if Machine A wants to call methods of some object on Machine B, then Machine B would have to export that object on its Remote Registry Server. Remote Registry Server is a service that runs on the server and helps client's search and access objects on the server remotely. Now, if an object has to be capable of being exported then it must implement the Remote Interface present in the RMI package. For example, say that you want an object Xyz on machine A to be available for remote method invocation, then it must implement the Remote interface.

RMI uses something called a stub and a skeleton. The stub is present on the client side, and the skeleton the server side. When you call remote methods, you don't just go directly to other machine and say "hey, here's the method name, here are the parameters, just give me back what has to be returned and I am out of here".

There are a number of events that have to take place beforehand which help in the communication of the data. The stub is like a local object on the client side, which acts like a proxy of the object on the server side. It provides the methods to the client which can be invoked on the server. The Stub then sends the method call to the Skeleton, which is present on the server side. The Skeleton then implements the method on the server side.

The Stub and the Skeleton communicate with each other through something called a Remote Reference Layer. This layer gives the stub and skeleton the capability to send data using the TCP/IP protocol. Let's take a quick look at a simple technique called "Binding".

Whenever a client wants to make a reference to any object on the server, have you thought how he would tell the server what object he wants to create? Well, this is where this concept of "Binding" comes in. On the server end we associate a string variable with an object (we have methods to do this. We will learn more about these when we start coding). The client tells the server what object he wants to create by passing that string to the server, thus letting the server know exactly what object you are talking about. All of these strings and objects are stored in the Remote Registry Server on the server.

What Do We Have To Code?

Now, before we jump into the code, let me tell you exactly what we have to code:
The Remote Object: This interface (yes our remote object is actually an interface) will have only method declarations. Hopefully you know that interfaces don't always have to have method bodies, just declarations. The Remote Object will have a declaration for each method that you want to export. This remote object would implement the Remote interface, which is present in the Java.rmi package.
The Remote Object Implementation: This is a class that implements the Remote Object. If you implement the Remote Object, it's common sense that you would override all of the methods in that object, so the remote object implementation class would actually have all of the method bodies of the methods that we want to export. This method will be extended from the UnicastRemoteObject class.
The Remote Server: This is a class that will act like a server to the client wanting to access remote methods. Here's the place where you will bind any string with the object you want to export. The binding process will be taken care of in this class.
The Remote Client: This is a class that will help you access the remote method. This is the end user, the client. You will call the remote method from this class. You will use methods to search and invoke that remote method.
Ok enough of the theory for now. Let's jump into the fun part: the coding...

The Coding
We will start by coding the remote object. Save this code as AddServer.Java in your favorite text editor:

```
import Java.rmi.*;

public interface AddServer extends Remote {

public int AddNumbers(int firstnumber,int secondnumber) throws RemoteException;

}
```

Let's have a look at this code. First of all we import the rmi package to use its contents. We then create an interface that extends the remote interface present in the Java.rmi package. All remote objects must extend the remote interface. We call this remote object AddServer. We have a method called AddNumbers in this remote object, which could be called by a client. We must remember that all remote methods need to throw the RemoteException, which is called whenever some error occurs.

We will now start coding the remote object implementation. This is the class that would implement the remote object and contain all of the method bodies. Save this code as AddServerImpl.Java in your favorite text editor:

```
import Java.rmi.*;

public class AddServerImpl extends UnicastRemoteObject implements AddServer {
public AddServerImpl() {
super();
}
public int AddNumbers(int firstnumber,int secondnumber) throws RemoteException {
return firstnumber + secondnumber;
}
}
```

First of all we import the rmi package to use its contents. We then create a class that extends the UnicastRemoteObject and implements out remote object that we have created. Next, we create a default constructor for our class, which with the help of the super keyword calls the constructor of the base class UnicastRemoteObject. We also see the body for the AddNumbers method, which throws a RemoteException. This way we are actually overriding the method in the remote object that we have created. The method body should be quite self-explainatory. We are receiving two integer parameters, adding them both and then returning their sum.

At this point we have two Java files: the remote object and the remote object implementation. We will now compile both of these file using the Javac command, like this:

Compile Remote Object:
C:\jdk\bin\Javac workingdir\AddServer.Java

Compile Remote Object Implementation:
C:\jdk\bin\Javac workingdir\AddServerImpl.Java

We end up with two Java files and two class files. We are now going to create our stub and skeleton. For creating the stub and skeleton files we have to use the rmic compiler on the remote object implementation file.

Rmic Compile Remote Object Implementation:
C:\jdk\bin\rmic workingdir\AddServerImpl.Java

After following the above step you will see that two newer class files have been created. They are AddServerImpl_Stub.class (the stub which will reside on the client side) and AddServerImpl_Skel.class (the skeleton which will reside on the server side).

Since we have the entire source compiled, let's create our client and server to see this demo in action. Save this code as RmiServer.Java in your favorite text editor:

```
import Java.rmi.*;
import Java.net.*;

public class RmiServer {
public static void main (String args[]) throws RemoteException, MalformedURLException {
AddServerImpl add = new AddServerImpl();
Naming.rebind("addnumbers",add);
}
}
```

Firstly we import the Java.rmi package and the Java.net package. We also use the throws clause to catch any necessary exceptions. Here we make an object out of the remote object implementation and on the very next line we use the rebind method to bind this object with the string addnumbers. This example shows exactly what I mean by binding.

From now on, whenever the client wants to make a reference to the remote object it would use the string addnumbers to do so. The rebind method takes two parameters: first, the string variable and second the object of the remote object implementation class.

Let's create the client. Save this code as RmiClient.Java in your favorite text editor:

```
import Java.rmi.*;
import Java.net.*;

public class RmiClient {
public static void main(String args[]) throws RemoteException, MalformedURLException {
String url="rmi://127.0.0.1/addnumbers";
AddServer add;
add = (AddServer)Naming.lookup(url);
int result = add.AddNumbers(10,5);
System.out.println(result);
}
}
```

First of all we import the Java.rmi package and the Java.net package. We also use the throws clause to catch all necessary exceptions. We then somehow try to make an object out of the remote object by using the lookup method in the Naming class, which is a Static method (that's the reason why we don't need to make an object of the Naming class and call it, we just use the class name).

The lookup method takes the complete URL name of the remote object, which consists of the complete machine I.P. address and the string which the object is associated to. This is the binding name of the object. As you can see, we use the RMI protocol to reference the remote object. This lookup method returns an object to us that we will have to typecast to the remote object data type before it can be used. Typecasting is a process through which we convert data from one data type to the other.

Since we have both our server and client source ready, let's compile them both:

Compile Remote Server:
C:\jdk\bin\Javac workingdir\RmiServer.Java

Compile Remote Client:
C:\jdk\bin\Javac workingdir\RmiClient.Java

Before we start testing our code we have to start the RMI Registry. The RMI registry is the place where all the bound data will be stored. Without it RMI wouldn't work!

Start Rmi Registry Server:
C:\jdk\bin\start rmiregistry

You will notice a new blank DOS prompt window will open. That's the RMI registry running. Make sure you leave it open as it is. Just like client server code, we first run the Rmi server in one DOS Prompt and then the RMI client in the other.

Start RMI Server:
C:\jdk\bin\Java workingdir\RmiServer

Start RMI Client:
C:\jdk\bin\Java workingdir\RmiClient

If everything worked out well then you should get the output of 15. We passed two integer values 10 and 5 to the AddNumbers method, which added them and returned an integer 15 back to us. If you got 15 then you have successfully executed a remote method. Of course, its not remote in the true sense as in this case you are the client and the server, but if you have a network then you can easily try this remotely.