Introduction to Java RMI

Lecture-1

Prof. Hari Mohan Pandey
Assistant Professor, CSE Department
Amity School of Engineering & Technology
hmpandey@amity.edu

Session Objective

- At the end of this lecture learns will be able to
  - Define and explore RMI
  - Develop RMI service application.
  - Explains the features of RMI

Remote Method Invocation (RMI)

- The Java Remote Method Invocation (RMI) application programming interface (API) *enables client and server communications over the net.*

- Remote method invocation allows applications
  - to call object methods located remotely,
  - sharing resources and
  - processing load across systems.

Remote Method Invocation (RMI) Cont.

- RMI allows any *java object type to be used- even if the client or server has never encountered it before*.
- RMI allows both *client and server to dynamically load new object types as required*.
- Remote Method Invocation (RMI) ***facilitates object function calls between Java Virtual Machines (JVMs).***
- JVM can be located on separate computers- yet one JVM can invoke methods belonging to an object stored in another JVM.

RMI Applications Development

- RMI applications often comprise two separate programs, a server and a client.
- A typical server program
  - creates some remote objects,
  - makes references to these objects accessible, and
  - waits for clients to invoke methods on these objects.
- A typical client program
  - obtains a remote reference to one or more remote objects on a server and
  - then invokes methods on them.

RMI Applications Development

- RMI provides the mechanism by which the server and the client communicate and pass information back and forth.

- Such an application is sometimes referred to as a *distributed object application*.

Distributed Object Application

- Distributed object applications need to do the following:
  - **Locate remote objects.**
  - **Communicate with remote objects.**
  - **Load class definitions for objects that are passed around.**

Locate Remote Objects

- Applications can use various mechanisms to obtain references to remote objects.
- For example, an application can register its remote objects with RMI's simple naming facility, the RMI registry.
- Alternatively, an application can pass and return remote object references as part of other remote invocations.

Locate Remote Objects Cont.

- Details of communication between remote objects are handled by RMI.
- To the programmer, remote communication looks similar to regular Java method invocations.
- Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.

Writing RMI Service

- The various steps in the development of a RMI service are as follows:
  - Writing an interface
  - Implementing the interface
  - Implementing the client
  - Running the application
  - Generation of Stub and Skeletons
    - **Install files on client and server machines**
    - **Starting RMI registry**
    - **Running server and client**

Writing an Interface

- The first thing we need to do is to agree upon an interface.
- An interface is a description of the methods we will allow remote clients to invoke.
- The method signature will be as follows:

```
double maxtwo(double a, double b);
```

- Save it under file name **imax2.java** in a directory name **server**.

```
import java.rmi.*;
public interface imax2 extends Remote
{
double maxtwo(double a, double b)throws RuntimeException;
 }
```

- Our interface name is **imax2** and it must extend **java.rmi.Remote**, which indicates that this is a remote service.

Implementing the interface

- In the implementation part we need to write a class which will be implementing the interface created in the first step.
- The class is responsible for providing the definitions of the methods declared in interface.
- In writing this class the real code need to be concerned about is the default constructor.
- Assume the class name is **Max2Class**. Its constructor must be defined as:

```
public Max2Class()throws RemoteException
 {}
```

Implementing the interface Cont.

- We have to declare a default constructor, even when we don’t have any initialization code for our service.
- This is because our default constructor can throw a **java.rmi.RemoteException**, from its parent constructor in **UnicastRemoteObject.**
- The implementation of the interface is given as:

```
public double maxtwo(double a, double b)throws RemoteException
{
        return a>b ? a : b;
}
```

Implementing the interface Cont.

- The complete source code of the file **Max2Class.java** is given below. Save also in the directory **server**.

```
import java.rmi.*;
import java.rmi.server.*;
public class Max2Class extends UnicastRemoteObject implements imax2
{
  public Max2Class()throws RemoteException  {}
  public double maxtwo(double a,double b)throws RemoteException
  {
     return a>b ? a:b;
  }
}
```

Implementing the interface Cont.

- Note the interface class must extend the class **UnicastRemoteObject** class.
- RMI provides some convenience classes that remote object implementations can extend which facilitate remote object creation.
- The class **UnicastRemoteObject** is one of them. The class is used for exporting a remote object and obtaining a stub that communicates to the remote object.

Creating Server

- Create a server class which will act as our RMI Server.
- Save the file under the name **Max2Server.java** in the server directory.
- The code for this class is given below.

```
import java.net.*;
import java.rmi.*;
public class Max2Server{
public static void main(String[] args){
        try{
                Max2Class ref = new Max2Class();
                Naming.rebind("max2ser", ref);
        }
        catch (Exception e){
                System.out.println("Exception:" + e);
        }
}}
```

Creating Server Cont.

- The crux of the code is the two statements re-written below:

  Max2Class ref = new Max2Class();

  Naming.rebind("max2ser", ref);

- Reference ref is of the class **Max2Class** created earlier.
- Naming is the class in **java.rmi** package. Its declaration is as follows:

  public final class Naming extends Object

- The Naming class provides method for storing and obtaining references to remote objects in a remote object registry.

Creating Server Cont.

- Each method of the Naming class takes as one of its arguments a name that is a **java.lang.String** in URL format (without the scheme component) of the form:

  //host: port/name

  - Where **host** is the host (remote or local) where the registry is located,
  - **port** is the port number on which the registry accepts calls, and
  - **name** is a simple string un-interpreted by the registry.

- Both host and port are optional. If **host** is omitted, the host defaults to the local host.
- If **port** is omitted, then the port default to 1099, the “well-known” port that RMI’s registry, **rmiregistry**, uses.

Implementing the Client

- The client receives an instance of the interface we defined earlier, and not the actual implementation. Some behind-the scenes work is going on, but this is completely transparent to the client.

```
String url = "rmi://127.0.0.1/max2ser";
imax2 mi =(imax2)Naming.lookup(url);
```

- To identify a service, we specify an RMI URL. The URL contains the hostname on which the service is located, and the logical name of the service. This returns an **imax2** instance, which can then be used just like a local object reference. We can call the methods just as if we'd created an instance of the remote **Max2Server** ourselves.

```
//call remote method
System.out.println("Maximum: "+mi.maxtwo(20.4,23.4));
```

Implementing the Client

```
import java.rmi.*;
public class Max2Client
{
    public static void main(String args[])
   {
        try
        {
                String url = "rmi://127.0.0.1/max2ser";
                imax2 mi = (imax2)Naming.lookup(url);
                System.out.println("Maximum is:"+mi.maxtwo(10.5,20.5));
        }
        catch (Exception e)
        {
              System.out.println("Exception: " + e);
        }
   }
}
```

Running the application

A. Generating Stub and Skelton:

- To generate stubs and skeletons, you use a tool called the RMI compiler, which is invoked from the command line, as shown here, into the server directory:

rmic Max2Class

- This command generates two new files: **Max2Class_Skel.class** (**skeleton**) and **Max2Class_Stub.class** (**stub**).

Running the application

B. Install files on client and server machines

- Onto the server directory the following files must be present: **imax2.class** (interface class file), **Max2Server.class** (the server class), **Max2Class.class** (interface implemented class), **Max2Class_Skel.class** (Skelton), **Max2Class_Stub.class** (**stub**).

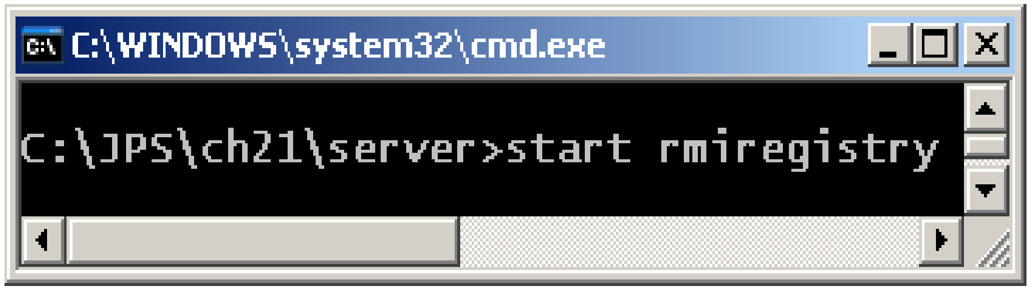
- Onto the client directory the following files must be present: **imax2.class** (interface class file), **Max2Client** (the client file) and **Max2Class_Stub.class** (**stub**).

Running the application

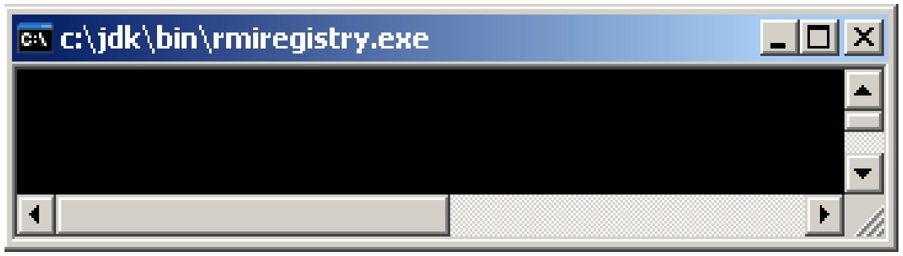
C. Starting RMI registry

- The JDK provides a program called rmiregistry, which executes on the server machine. It maps names to object reference. Start the RMI Registry from the command line as shown here:

start rmiregistry



```
C:\JPS\ch21\server>start rmiregistry
```



Running the application

D. Running server and client

- Move onto the server directory and start the server in a separate window as:

```
java Max2Server
```

- Now move onto the client directory and start the client in a separate window as:

```
java Max2Client
```

- Output you will get is:

```
Maximum is 20.5
```