Unit – I Java RMI

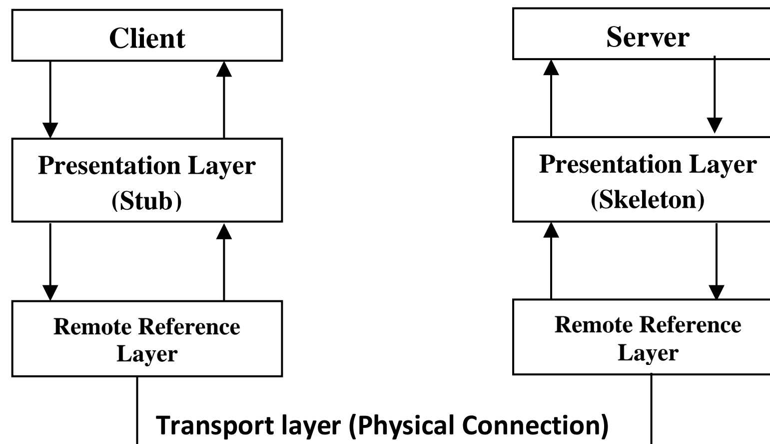
Introduction to Remote Method Invocation (RMI)

- RMI is the action of invoking a method of remote interface on a remote object. It is used in distributed object systems.
- RMI allows a Java object that execute on one machine to invoke a method of a Java object that executes on another machine.
- RMI uses object serialization to marshal and unmarshal parameters.

1.1 Architecture of RMI

RMI architecture consists of four layers namely:

- Application layer
- Presentation layer or Stub and Skeleton layer
- Session layer or Remote Reference layer and
- Transport layer



- ✓ Application layer stands for the client and Server. A client invokes a method on a remote server object actually makes use of the stub or proxy for the remote object as a contact to remote object.
- ✓ The stub and skeleton layer or proxy layer is used for marshalling and unmarshalling the data that is transferred through the network. Marshalling is the process of converting Java objects in to sequence of bytes(streams) and unmarshalling does the reverse
- ✓ Remote Reference Layer is responsible for carrying out the semantics of the invocation
- ✓ Transport layer handles the actual machine-to-machine communication and keeping track of remote object.
- ✓ Stub is a client-side communication logic and skeleton is server-side communication logic

1.2 The RMI Package

The interfaces and classes that are responsible for specifying the remote behavior of the RMI system are defined in the *java.rmi* package hierarchy. The relationship between these interfaces and classes are shown in the following figure.

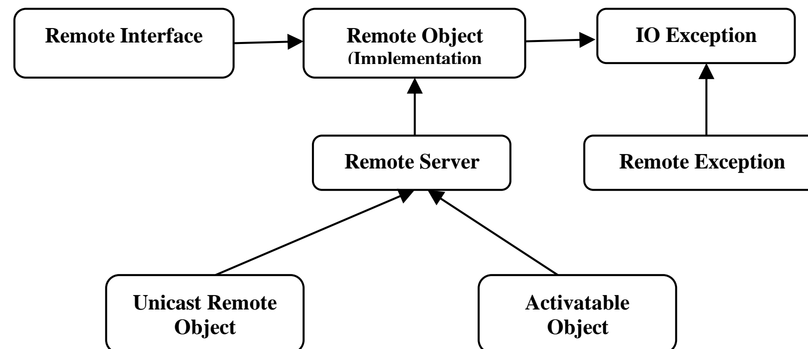
The Remote Interface

The Remote interface declares the set of methods that can be invoked by the client from a remote Java object. It must satisfy the following requirements

- The remote interface must be public
- It must at least extend *java.rmi.Remote* interface
- Each method declared in the interface must include *java.rmi.RemoteException* in its throws clause
- Any remote object is used in remote method declaration , it must be declared as the remote interface and not as the implementation class of that interface.

For example

```
public interface BankInterface extends java.rmi.Remote
{
    public void deposit(int accno, float amount) throws java.rmi.RemoteException;
    public void withdraw(int accno, float amount) throws java.rmi.RemoteException;
    public float balance(int accno) throws java.rmi.RemoteException;
}
```



The RemoteException class

The *java.rmi.RemoteException* class is the super class of exceptions thrown by RMI-runtime environment. It is thrown when the following failures occur:

- Failure in communication
- Failure while marshalling / unmarshalling parameters
- Protocol errors

The RemoteObject class (Interface implementation class)

The functionalities of the RMI server are provided by the implementation class. The implementation class has the following functionalities

- It provides the implementation for the remote objects methods
- UnicastRemoteObject class provides the methods needed to create remote objects and export them. It defines single remote object that is valid only when the process is alive.
- Activatable subclass identifies whether the remote object is a simple remote object or an activatable remote object that executes when invoked

For example

```
public class BankImp extends UnicastRemoteObject implements BankInterface
{
private float balance = 0.0;
public BankImp(float balance)
{ this.balance = balance; }
public void deposit(int accno, float amount) throws java.rmi.RemoteException
{……………………}
public void withdraw(int accno, float amount) throws java.rmi.RemoteException
{ …………………..}
public float balance(int accno, float amount) throws java.rmi.RemoteException;
{…………………..}
}
```

Remote Server

The *RemoteServer* class is the common super class to server implementations and provides the framework to support a wide range of remote reference semantics. Specifically, the functions needed to create and export remote objects are provided abstractly by *RemoteServer* and concretely by its subclass(es).

Parameter Passing in RMI

- Parameter in RMI call are written to a stream that is a subclass of ObjectOutputStream class, in order to serialize the parameters to the destination of the remote call.

- Parameter that are objects are written to the stream using ObjectOutpusSteam's writeObject() which in turn calls the replaceObject()
- If the object passed is a remote object and is exported at run-time then its stub is passed
- If it is not exported at run-time then the object itself is passed.
- If the object is not a remote object then it is simply returned

1.3 Creation of RMI application

The steps involved in the creation of an RMI system are listed below

Step 1: Create a remote interface which extends Remote interface is treated as Remote Object

For example

```
import java.rmi.*;
public interface Hello extends Remote
{ public String sayHello() throws RemoteException; }
```

Step 2: Create a class that implements the remote interface. All the remote objects must extend the UnicastRemoteObject which provides functionality that is needed to make objects available from remote mechanism

For example

```
import java.rmi.*;
public class HelloImp extends UnicastRemoteObject implements Hello
{
HelloImp() throws RemoteException
{  super();  }
public String sayHello() throws RemoteException
{return “Hello”; }
}
```

Step 3: The server program uses createRegistry method of LocateRegistry class to create rmiregistry within the server JVM with the port number passed as an argument. The rebind method of Naming class is used to bind the remote object to the new name Naming.rebind(“rmi://machineName:portNo/name”,ObjRef); Where the first argument is a string that names the server and the second argument is a reference to an instance of implementation class.

For example

```
import java.rmi.*;
import java.rmi.server.*;
public class Server
{
     public static void main(String args[]) {
   HelloImp h = new HelloImp();
try   {
   UnicastRemoteObject.exportObject(h);
   LocateRegistry.createRegistry(4000);
   Naming.rebind("rmi://sys-name:4000/Hello",h);
   System.out.println("Waiting for a Client at port number 4000...");
     }
 catch(Exception e)
 { e.printStackTrace(); }
 }
 }
```

Step 4: Compile all the Server side classes and interfaces as

- javac *.java <enter>
- rmic HelloImp <enter>

  HelloImp_stub.class and HelloImp_skeleton classes are created
- start RMIRegistry 4000<enter>

➢java Server <enter>
Now Server is ready to service

Step 5: Keep the stub and interface class files either on a web server or copy them to client machine

Step 5: Develop a client program and execute it

```
import java.rmi.*;
import java.rmi.server.*;
class HelloClient{
  public static void main(String args[])  {
   try   {
   System.setProperty("java.security.policy","all.policy");
   System.setProperty("java.rmi.server.codebase","http://localhost:8080/");
   System.setSecurityManager(new RMISecurityManager());
   Hello r = (Hello)Naming.lookup("rmi://sys-name:4000/Hello");
   String s = r.sayHello();
   System.out.println(s);
   }
  catch(Exception e)
   { e.printStackTrace(); }
   }
}
```

Advantages:

- RMI supports for distributed programming
- Facilitates seamless remote invocation on objects
- Facilitates usage of applets on client side

Disadvantages:

- RMI supports only Java to Java communication , (ie) cross-platform communication is not possible
- Naming Server is not powerful enough for industrial applications
- No security in the communication
- Poor concurrency control
- No transaction control

1.4 CORBA

- Common Object Request Broker Architecture (CORBA) is standard middleware architecture for distributed object system.
- Its specification was created by Object Management Group (OMG).
- CORBA allows a distributed, heterogeneous collection of objects to interoperate transparent to the object location, hardware, operating system, programming language and communication protocol.
- CORBA integrates different types of machines and it is preferred middleware for large enterprises.

1.4.1 OBJECT MANAGEMENT ARCHITECTURE (OMA)

- The Primary goals of OMGs are the reusability, portability and interoperability of object-based software in distributed, heterogeneous environments.
- The OMA provides the conceptual infrastructure upon which all OMG specifications are based.
- It describes objects as immutable, whose services are exposed through interfaces.
- It consists of two parts, namely, OMG Object Model and OMG Reference Model.

THE OMG OBJECT MODEL

This model describes the objects across a heterogeneous environment. The OMG Object Model is based on objects, operations, types and subtyping. The standard of OMG object model is CORBA. The OMG Object Model consists of the following parts:

1. **Object**: It is an encapsulated entity that provides services to the clients.
2. **Request**: It is an action created by a client directed to a target object that includes information on the operation to be performed.
3. **Object Creation and Destruction**: Based on the client's requests objects are created or deleted.
4. **Types**: It is an identifiable entity with values.
5. **Interface**: It is the specification of operation that a client can request from an object.
6. **Operation**: It is an identifiable entity that defines what a client can request from the object.

The OMG Reference Model

It deals with the interconnection between distributed objects in a heterogeneous environment to promote interoperability between them. The OMG Reference Model consists of the following components

1. **Object Request Broker**: It facilitates communication between clients and objects. It enables objects to transparently send/receive requests/response in a distributed environment.
2. **Object Services**: It is a collection of services that support basic function for using and implementing objects. Services are independent of application domains.
3. **Common Facilities**: It is a collection of services that many applications can share, but is not as fundamental as the object services.
4. **Domain Interface**: These interfaces fill roles similar to object services and common facilities but are oriented towards specific application domains.
5. **Application Interfaces**: These are interfaces developed specifically for a given application. They are user-defined, application-specific and have proprietary interfaces.

1.5 CORBA ARCHITECTURE

CORBA is an open distributed object computing infrastructure standardized by the OMG. The core of the CORBA architecture is the ORB that acts as the object bus over which objects transparently interact with other objects located locally or remotely. A CORBA object is represented to the outside world by an interface with a set of methods. The interface defined in an IDL file serves as a contract between a server and its clients. The main components of CORBA are

1. **Object Request Broker (ORB)**: ORB provides the communication and activation infrastructure for distributed object applications. To make a request, the client specifies the target object by using an object reference (OR). When a CORBA object is created, an OR is also created. ORB core facilitates the following transparencies in client-server communication:
   - **Object location**: Here the client has no knowledge of whether the target object is in-process or out-of-process in the same machine or a different machine.
   - **Object implementation**: The client does not know the language, platform or hardware in which object is implemented.
   - **Object execution state**: The client does not know whether the object is active or inactive. ORB transparently activates object when required by the client.

- **Object communication mechanism**: ORB can use memory, RPC and TCP/IP for communication.

2. **OMG Interface Definition Language (OMG IDL**): An object's interface specifies the operations and types that the object supports. OMG IDL types are mapped onto programming languages like C, C++, COBOL, Java, Smalltalk and Perl.
3. **Interface Repository (IFR)**: The CORBA IFR allows the IDL-types system to be accessed and written programmatically at run-time. So it is used in dynamic object invocation.
4. **Stubs and Skeletons**: OMG IDL language compilers generate programming language type interfaces, client-side stubs and server-side skeletons.
5. **IDL Compiler**: It is a tool which generates static stubs [Static Invocation Interface (SII)] and skeleton [Static Skeleton Interfaces (SSI)] from the interface definition.
6. **Dynamic Interface Invocation (DII**): The DII allows clients to generate requests at run-time. It is useful when an application has no compile-time knowledge of the interface it accesses.
7. **Object Adapter (OA)**: It serves as the glue between CORBA object implementations and the ORB.
8. **Inter-ORB Protocol**: To support interoperability, CORBA proposes General Interoperable Protocol (GIOP) and Interoperable OR (IOR). GIOP defines a common mechanism of communication between ORB in general terms.
9. **ORB Interface**: This ORB interface provides standard operations to initialize and shutdown the ORB, convert object references to strings and back, and create argument lists for requests made through the DII.
10. **Implementation Repository (IMR)**: IMR contains information that allows an ORB to activate servers on demand.

1.6 OMG CORBA IDL

OMG IDL is a stable standard that was established in 1991. It was adopted by ISO due to its stability and credibility. Its main aim is to bring about language independence in a heterogeneous environment. The following are some of the characteristics of IDLs:

- **Multiple Language Bindings**: A single software interface can be bound to different languages such as COBOL, C, C++, Java, Smalltalk, Ada, Perl, etc.
- **Platform Independence**: IDL enables platform independence as the interface defined by IDL is represented consistently on any ORB and platform.
- **Implementation Independence**: IDL is a pure specification and not an implementation. Hence IDL does not have implementation-dependent features such as looping and control structures.
- **Increased Reuse**: The quality of IDL designs is critically important. Quality IDL designs are reusable across many object implementations

  and hence facilitate interoperability and adaptability of the applications. It thus facilities software maintenance.

CORBA OBJECT LIFECYLCE

There are four events in CORBA object lifecycle. The first two are for CORBA objects and the last two are for servants. They are:

- **Creation**: CORBA objects are created using factory objects. These objects offer operations to create new objects.

- **Deletion**: The lifecycle of a CORBA object ends with the delete event. CORBA objects are deleted using delete () operation in their IDL interface.
- **Activation**: It makes the servant available to process requests for a particular CORBA object.
- **Deactivation**: It unbinds the servant from the CORBA object. It also results in destruction of the servant.

1.7 CORBA SERVICES

CORBA Services are collections of system-level services packaged with IDL specified interfaces. The services can be used to create a component, name a component and introduce a component into the environment. OMG has published standards for 15 object services:

1. **Life Cycle Service**: Defines operations for creating, copying, moving and deleting components on the bus.
2. **Persistence Service**: Provides a single interfaces for storing components persistently on a variety of storage servers including RDBMSs and simple files, etc.
3. **Naming Service**: Allows components on the bus to locate other components by name.
4. **Event Service**: Allows components on the bus to dynamically register or unregister their interest in specific events. It defines well known event channel that collects and distributes events among components that know nothing of each other.
5. **Concurrency Control Service**: Provides a lock manger that can obtain locks on behalf of either transactions or threads.
6. **Transaction Service**: Provides two phase commit coordination among recoverable components using either flat or nested transactions.
7. **Relationship Service**: Provides a way to create dynamic associations or links between components that know nothing of each other.
8. **Externalization Service**: Provides a standard way for getting data into and out of a component using a stream like mechanism.
9. **Query Service**: Provide query operations for objects. It's a superset of SQL
10. **Licensing Service**: Provides operations for metering the use of components to ensure fair compensation for their use.
11. **Properties Service**: Provides operations that let you associate named values with any component.
12. **Time Service**: Provides interfaces for synchronizing time in a distributed object environment.
13. **Security Service**: Provides a complete framework for distributed object security. It supports authentication, access control lists, confidentiality, etc.
14. **Trader Service**: Provides yellow pages for objects; it allows objects to publicize their services and bid for jobs.
15. **Collection Service**: Provides CORBA interfaces to generically create and manipulate the most common collection.