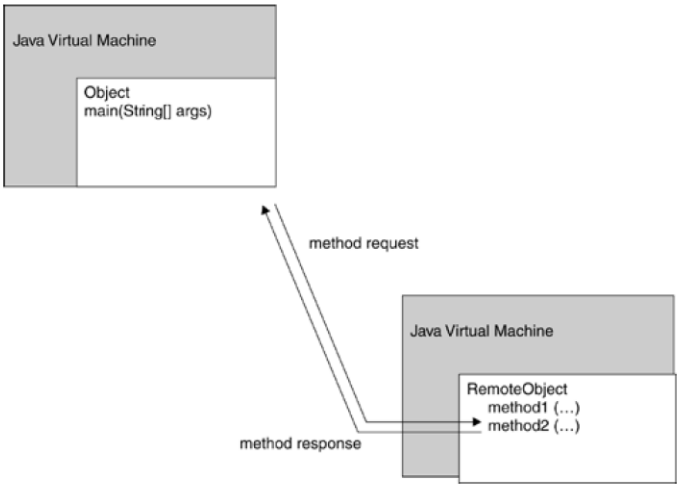
REMOTE METHOD INVOCATION (RMI)

- Remote Method Invocation (RMI) is a distributed systems technology that allows one Java Virtual Machine (JVM) to invoke object methods that will be run on another JVM located elsewhere on a network.
- This technology is extremely important for the development of large-scale systems, as it makes it possible to distribute resources and processing load across more than one machine.



- An object running on one JVM invokes a method of an object hosted by another JVM.
- Communication like this does not have to be a one-way process, either—a remote object method can return data as well as accept it as a parameter.

How does RMI work?

- Systems that use RMI for communication typically are divided into two categories: clients and servers.
- A server provides an RMI service, and a client invokes object methods of this service.
- RMI servers must register with a lookup service, to allow clients to find them, or they can make available a reference to the service in some other fashion.
- Included as part of the Java platform is an application called *rmi-registry*, which runs as a separate process and allows applications to register RMI services or obtain a reference to a named service.
- Once a server has registered, it will then wait for incoming RMI requests from clients.

- Associated with each service registration is a name (represented as a string), to allow clients to select the appropriate service.
- If a service moves from one server to another, the client need only look up the registry again to find the new location. This makes for a more fault tolerant system
- If the service is unavailable because a machine is down, a system administrator could launch a new instance of the service on another system and have it register with the RMI registry.
- Providing the registry remains active, you can have your servers go online and offline or move from host to host.
- The registry doesn't care which host a service is offered from, and clients get the service location directly from the registry.

RMI Steps:

Step1:

- **Create and compile** an interface that specifies the methods that will have remote access.
- Must be **public**
- Must **extend** the interface java.rmi.Remote
- Each remote method must have a **throws** java.rmi.RemoteException clause
- Interface must reside on both the server and the client side
- Any method parameters or return value of a reference type must implement Serializable.

Step 2:

- Write and compile a class that implements the remote interface in 1.
- Must extend java.rmi.server.UnicastRemoteObject
- Must implement the remote interface in 1
- Its constructors must be defined explicitly since they each may throw java.rmi.RemoteException
- Resides on the server side only

Step 3:

- Create the stub class using the rmic tool.
  rmic ImplementationClass
- Leave the stub on the server side
- Put a copy of the stub class on the client side

Step 4:

- Write and compile a server class that Instantiates an object of the implementation class in 2;
  This is the remote object
- Binds the remote object implObj to a unique identifier (a String) for the rmi registry
  java.rmi.Naming.rebind("uniqueID", implObj);

Note: The implementation class and the server class can be combined into one class, particularly if there is no need for the server class to extend another class.

Step 5:

- Write and compile a client class that
- Requests an object from the remote server using its hostname and the unique identifier of the object, and then casts that object to the interface type from 1.
- InterfaceType it = (InterfaceType)java.rmi.Naming.lookup("rmi://localhost/uniqueID");
  1099 is the default port and if used, its specification here can be omitted.
- If the client is running on the same machine as the remote object (the server), use localhost.

Step 6:

- Start the bootstrap rmi registry in the background on the server side.
- Start rmiregistry

Step 7:

- Execute the server class on the machine that the client named, **java ServerClass**

Note: The rmi registry and the server must run on the same machine and in the same directory.

Example Code:

Create search_city interface

```
import java.rmi.*;
public interface search_city extends Remote
 {
public String find_city(String city_name) throws RemoteException;
}
```

Create Imp class. // Server class

```
import java.rmi.*;
import java.rmi.registry.Registry;
import java.rmi.server.*;
// Hello Server.
public class impl extends UnicastRemoteObject implements search_city
{
String str;
impl() throws RemoteException
{
str="";
}
public String find_city(String city_name) throws RemoteException
{
int j=0;
String A[][]={ {"lahore","021"},{"karachi","022"},{"islamabad","023"},{"peshawar","024"}};

for(int i=0;i<=3;i++)
{
if(city_name.equals(A[i][j]))
{
j++;
str=A[i][j];
```

```
break;
}
else
{
j++;

}

}
return str;
}
public static void main(String args[]) throws RemoteException
{
  try {
         Registry r = java.rmi.registry.LocateRegistry.createRegistry(1092);
         impl obj=new impl();
         r.rebind("Search_City", obj);
         System.out.println("Server is connected and ready for operation.");
        }catch (Exception e) {
         System.out.println("Server not connected: " + e);
       }
}
}
```

Create client class

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class Client
{
    public static void main(String[] argv) {
        Client C=new Client();
        C.connetRemote();
       }
   private void connetRemote()  {
       try {
       Registry reg = LocateRegistry.getRegistry("localhost",1092);
       search_city city = (search_city) reg.lookup("Search_City");
       System.out.println(city.find_city("lahore"));
       } catch(Exception ee)
       {
          System.out.println("Exception:   "+ee);
       }
   } }
```