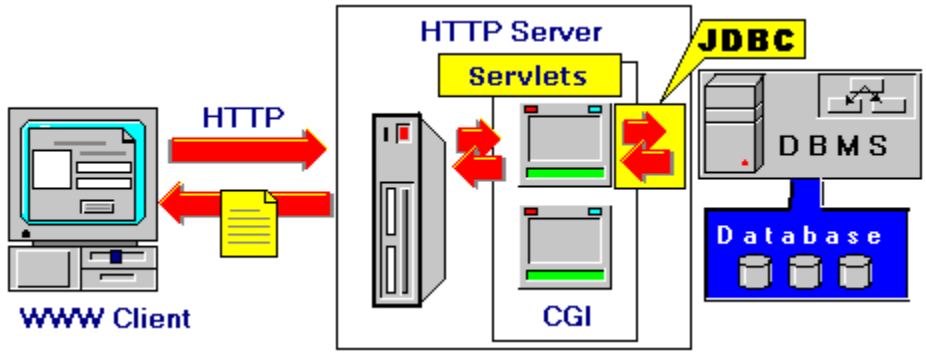
mood-book

UNIT III- JDBC



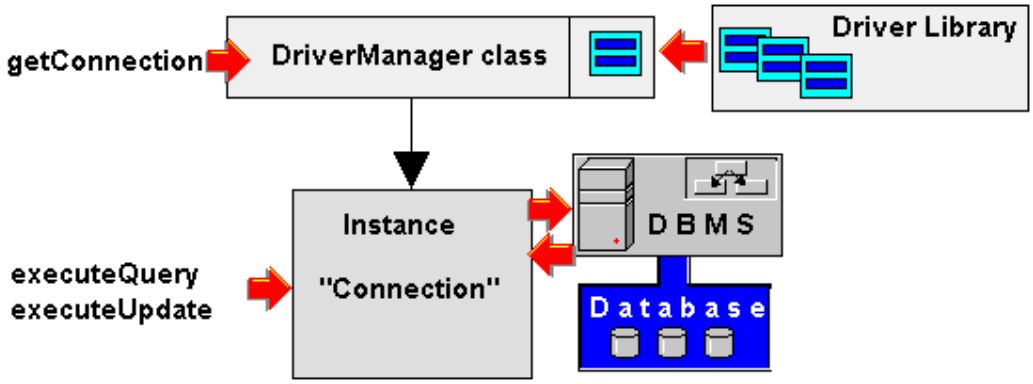
Basic Principles

Basically, the JDBC API is a software library providing a number of Java classes and interfaces that allows programmers to:

- Establish a connection to a wide range of different Relational Database Management Systems (RDBMS). Note, that each major RDBMS, such as Oracle, Microsoft SQL, or MySQL is supported.
- Access a database within a particular system.
- Manipulate the structure of a database by creating, deleting, or altering relations from that database.
- Manipulate the content of a database by inserting, deleting, or updating records in database relations.
- Retrieve the content from a database by querying the database.

Basically, the JDBC operates with two main classes:

1. DriverManager class operates with a library of drivers for different DBMS implementations. The DriverManager class loads requested drivers, physically installs connrection to a database and return an instance of a data class "Connection".
2. An instance of the class "Connection" represent a single connection to a particular database. All the communication to the database is carryed out via this object.



Installing Connection:

Establishing JDBC Connection means obtaining a correct instance of so-called "Connection" class.

Usually, establishing a connection is carried out in two steps:

1. loading an appropriate JDBC driver for the installed RDBMS.
2. installing connection and getting reference to the Connection object

Loading a JDBC driver is very simple and involves just a single line of Java code.

```
...
try
{    Class.forName("com.mysql.jdbc.Driver"); }
catch(ClassNotFoundException exc){exc.printStackTrace();}
...
```

This line of code just notifyes the DriverManager which particular Java class should be loaded as a JDBC driver class.

The next step in establishing a database connection is a message to loaded driver requesting actual connection to the RDBMS. The operation is carryed out by sending message "getConnection" to the driver manager. Note that "DriverManager" returns a "Connection" instance that is used for further processing the database.

```
try
{
...
Connection connection_;
```

```
String dbms = "jdbc:mysql://" + host + "/" + db;
connection_ = DriverManager.getConnection(dbms, username, password);
}
catch(ClassNotFoundException exc){exc.printStackTrace();}
```

Method "getConnection()" accepts three arguments:

1. A so-called Database URL, which encoded using standard URL syntax (protocol + host + object). The protocol part starts always with "jdbc:" folowed by the name of the RDBMS (in our case "mysql") and terminated with "://" symbols. Thus, the protocol part in our example is "jdbc:mysql://". The host part identifies a server where the DBMS is running. In our case (Servlets & DBMS on the same computer) "localhost" can be used to identify the host. Finally, the name of a particular database must be supplied preceeded with the slash character. In our case this would be "/example".
2. A registered username that has the proper privileges for manipulating the database.
3. A password valid for the username.

Working with a Database

In order to actually work with a databaase, a special "Statement" class is used.

In order to create an instance of such "Statement" class, a message "createStatement" is sent to the previously created instance of JDBC connection.

```
try
{
Statement statement = connection_.createStatement();
}
catch(SQLException exc)
{
exc.printStackTrace();
}
```

If an error occurs during the execution of the createStatement() method a SQLException will be thrown. Instances of the Statement Class provides a public interface to insert, update, or retrieve data from a database. Depending on a particular database operation, an appropriate method should be invoked. For instance,

- executeUpdate() can be used to insert data into a relation
- executeQuery() can be used to retrieve data from a database

...

```
try
{
String insert_sql_stmt = "INSERT INTO " + table + " VALUES(" + values + ")";
statement.executeUpdate(insert_sql_stmt);
}
catch(SQLException exc){exc.printStackTrace();} ...
```

Other methods of the "statement" class can be also applyed to its instances.

if we need to retrieve the keys automatically generated by the "executeUpdate" statement, we need to pass the "Statement.RETURN_GENERATED_KEYS" argument in advance.

```
try
{
String sql = "INSERT INTO " + table + " VALUES(" + values + ")";
statement.executeUpdate(sql,Statement.RETURN_GENERATED_KEYS);
ResultSet keys = statement.getGeneratedKeys();
}
catch(SQLException exc){exc.printStackTrace();}
```

Similarly, to retrieve data from a database we need to obtain an instance of the Statment class, and then to invoke executeQuery() method on this instance. This method takes a string containing SQL source as an argument.

```
try
{
String sql = "SELECT ...";
ResultSet query_result = statement.executeQuery(sql);
```

```
}   catch(SQLException exc){exc.printStackTrace();}
```

Note, that the "sql" argument should contain a valid SQL Select statement. The executeQuery() method returns an instance of the ResultSet class. Generally, execution of any JDBC statement that returns data from a database, results in an instance of the ResultSet class. This instances may be seen as a number of rows (tuples) that hold the current results. The number and type of columns in this object corresponds to the number and types of columns returned as the result from the database system.

Consider the following sample database:

```
Customer(cn,cname,ccity);
Product(pn,pname,pprice);
Transaction(cn,pn,tdate,tqnt);
```

```
...
try
{
String sql = "SELECT * FROM Customer;";
ResultSet query_result = statement.executeQuery(sql);
...
```

The "executeQuery" command will result in obtaining an instance of the ResultSet class which will hold all tuples from the Customer table as rows, each row will contain 3 values: "cn", "cname" and "ccity". Normally, the SQL statement exlicitly defines the "ResultSet" internal structure. Once when we set a current row of the ResultSet, we can retrieve values by means of a number of methods. The methods correspond to a column type. Thus, to retrieve the value of a string column, we invoke a getString() method. Similarily, to retrive an integer value we simply invoke a getInt() method.

```
try
{
String sql = "SELECT cname, pname, qnt";
sql = sql + " FROM Customer, Product, Transaction";
```

```
sql = sql + " where Customer.ccity = \"Graz\" And";
sql = sql + " Customer.cn = Transaction.cn And";
sql = sql + " Transaction.pn = Product.pn";
ResultSet query_result = statement.executeQuery(sql);
while(query_result.next())
{
String customerName = query_result.getString("cname");
String productTitle = query_result.getString("pname");
int productQuantity = query_result.getInt("cid");
... }
```

Socket Programming

The basic java classes (java.net package) that supports Socket programming, namely:

- InetAddress
- Socket
- ServerSocket
- DatagramSocket
- DatagramPacket
- MulticastSocket

1. The InetAddress class:

The java.net.InetAddress class is used to represent an IP address as an object. It is also used to resolve from host name to IP address (via the DNS) and vice versa.

No Constructors, instead, factory methods are provided.

The following are some of the methods of the **InetAddress** class.

Static InetAddress getByName(String host)	Takes a hostname and returns InetAddress object representing its IP address.
Static InetAddress[] getAllByName(String host)	Takes a hostname and returns an array of InetAddress objects representing its IP addresses.
static InetAddress getLocalHost()	Returns an InetAddress object representing the IP address of the local host

String getHostAddress()	Returns IP address string of this InetAddress object
string getHostName()	Returns the hostname of this InetAddress object.
boolean isLoopbackAddress()	Checks if the InetAddress is a loopback address.
boolean isMulticastAddress()	Checks if the InetAddress is an IP multicast address.
String toString()	Converts this InetAddress to a String.

Example 1: IP Address Resolver.

The following example first prints the Address of the current machine and then in a loop reads a host or an address from a user and resolving it.

```
import java.net.*;        import java.io.*;

public class IPAddressResolver {
        public static void main (String args[]) {
                try {
        BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
                        InetAddress myself = InetAddress.getLocalHost();
                        System.out.println("MyAddress is :"+myself);
                        while (true) {
                                System.out.print("host name or IP to resolve - <exit> to quit: ");
                                String input = stdin.readLine();
                                if (input.equalsIgnoreCase("exit")              break;
                                InetAddress address = InetAddress.getByName(input);
                                if (isHostName(input))
                                System.out.println("IP Addrsss is: "+ address.getHostAddress());
                                else
                                System.out.println("Host name is: "+ address.getHostName());
                                System.out.println("addresses for "+input+ " are: ");
                                InetAddress[] addresses =
InetAddress.getAllByName(address.getHostName());
                                for (int i = 0; i < addresses.length; i++)
```

```
                    System.out.println(addresses[i]);
            }       }
        catch (UnknownHostException e) {
            System.out.println("Exception: "+e);          }
        catch (Exception ex) {
            System.out.println("Exception: "+ex);         }   }

    private static boolean isHostName(String input) {
        char[] ca = input.toCharArray();
        for (int i = 0; i < ca.length; i++) {
            if (!Character.isDigit(ca[i]) && ca[i] != '.')
                return true;        }
        return false;   } }
```

TCP Sockets (Stream Sockets)

Java provides two classes for creating TCP sockets, namey, Socket and ServerSocket.

The java.net.Socket class:

This class is used by clients to make a connection with a server

Socket constructors are:

Socket(String hostname, int port)

Socket(InetAddress addr, int port)

Socket(String hostname, int port, InetAddress localAddr, int localPort)

Socket(InetAddress addr, int port, InetAddress localAddr, int localPort)

Creating socket

Socket client = new Socket(“www.microsoft.com", 80);

Note that the Socket constructor attempts to connect to the remote server - no separate connect() method is provided.Data is sent and received with output and input streams. The Socket class has the following methods, that retruns **InputStream** and the **OutputStream** for reading and writing to the socket

public InputStream getInputStream()

public OutputStream getOutputStream()

There's also a method to close a socket:

public synchronized void close()

The following methods are also provided to set socket options:

void setReceiveBufferSize()

void setSendBufferSize()

void setTcpNoDelay()

void setSoTimeout()

The java.net.ServerSocket class

The **ServerSocket** class is used to by server to accept client connections

The constructors for the class are:

public ServerSocket(int port)

public ServerSocket(int port, int backlog)

public ServerSocket(int port, int backlog, InetAddress networkInterface)

Creating a ServerSocket

ServerSocket server = new ServerSocket(80, 50);

Note: a closed ServerSocket cannot be reopened

ServerSocket objects use their accept() method to connect to a client

public Socket accept()

accept() method returns a **Socket** object, and its ***getInputStream()*** and ***getOutputStream()*** methods provide streams for reading and writing to the client.

Note: There are no getInputStream() or getOutputStream() methods for ServerSocket

Example 2: The following examples show how to create TcpEchoServer and the corresponding TcpEchoClient.

```
import java.net.*;    import java.io.*;    import java.util.*;
public class TcpEchoServer
{
        public static void main(String[] args)
        {
```

```
                int port = 9090;
                try {
                ServerSocket server = new ServerSocket(port);
                        while(true) {
    System.out.println("Waiting for clients on port " + port);
    Socket client = server.accept();
    System.out.println("Got connection from "+client.getInetAddress()+":"+client.getPort());
BufferedReader reader = new BufferedReader(new
InputStreamReader(client.getInputStream()));
PrintWriter     writer = new PrintWriter(client.getOutputStream());
writer.println("Welcome to my server");       writer.flush();

String message = reader.readLine();
while (!(message == null || message.equalsIgnoreCase("exit"))) {
System.out.println("MessageReceived: "+message);
writer.println(message);    writer.flush();
message = reader.readLine();                    }
client.close();    } }
catch(Exception ex) {
                        System.out.println("Connection error: "+ex);
                } }   }

public class TcpEchoClient
{
        public static void main(String[] args) {     int port = 9090;
                try {
                        String host = InetAddress.getLocalHost().getHostName();
                Socket client = new Socket(host, port);
                        PrintWriter writer = new PrintWriter(client.getOutputStream());
BufferedReader reader = new BufferedReader(new
InputStreamReader(client.getInputStream()));
```

```
BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
System.out.println(reader.readLine()); //read welcome message
                    String message;
                    while (true) {
                            System.out.print("Enter message to echo or Exit to end : ");
                            message = stdin.readLine();
                            if (message == null || message.equalsIgnoreCase("exit"))
                                    break;
                            writer.println(message);        writer.flush();
                            System.out.println("Echo from server: "+reader.readLine()); }
                    client.close();    }
catch (Exception ex) {
                    System.out.println("Exception: "+ex);                }        }    }
```

Example 3: Multi-Client Tcp Servers

The following example shows how to create a multi-client TCP server.

```
import java.net.*;
import java.io.*;
import java.util.*;

public class MultiClientTcpEchoServer
{       public static void main(String[] args)
        {               int port = 9090;
                try {     ServerSocket server = new ServerSocket(port);
                        while(true) { System.out.println("Waiting for clients on port " + port);
                                Socket client = server.accept();
                                ConnectionHandler handler = new ConnectionHandler(client);
                                handler.start();          }
                } catch(Exception ex) {
                        System.out.println("Connection error: "+ex);          }        }    }
```

```
class ConnectionHandler extends Thread {
        private Socket client;          BufferedReader reader;          PrintWriter writer;
        static int count;
        public ConnectionHandler(Socket client) {
                this.client = client;
     System.out.println("Got connection from"+client.getInetAddress()+":"+client.getPort());
    count++;
    System.out.println("Active Connections = " + count);  }
        public void run() {
                String message=null;


                try {
reader = new BufferedReader(new InputStreamReader(client.getInputStream()));
writer = new PrintWriter(client.getOutputStream());
writer.println("Welcome to my server"):                writer.flush();
message = reader.readLine();
while (!(message == null || message.equalsIgnoreCase("exit"))) {
writer.println(message);     writer.flush(); message = reader.readLine(); }
client.close();     count--;
System.out.println("Active Connections = " + count);
} catch (Exception ex) {        count--;
System.out.println("Active Connections = " + count);        }     }     }
```

UDP Sockets (Datagram Sockets)

Java provides two classes for creating UDP Sockets:

- **DatagramPacket** class, used to represent the data for sending and receiving.
- **DatagramSocket** class for creating a socket used to send and receive DatagramPackets.

The java.net.DatagramPacket class:

Constructor for receiving:

public DatagramPacket(byte[] data, int length)

Constructor for sending :

public DatagramPacket(byte[] data, int length, InetAddress addr, int port)

Some methods provided by the DatagramPacket class are:

public synchronized void setAddress(InetAddress addr)

public synchronized void setPort(int port)

public synchronized void setData(byte data[])

public synchronized void setLength(int length)

The java.net.DatagramSocket class:

Constructor for sending:

public DatagramSocket()

Constructor for receiving :

public DatagramSocket(int port)

public DatagramSocket(int port, InetAddress addr)

Sending UDP Datagrams involves the following steps:

- Convert the data into byte array.
- Create a **DatagramPacket** using the array
- Create a **DatagramSocket** using the packet and then call ***send()*** method

Receiving UDP Datagrams involves the following steps:

- Construct a **DatagramSocket** object on the **port** on which you want to listen
- Pass an empty **DatagramPacket** object to the DatagramSocket's ***receive()*** method
  public synchronized void receive(DatagramPacket dp)
- The calling thread blocks until a datagram is received
- dp is filled with the data from that datagram
- After receiving, use the ***getPort()*** and ***getAddress()*** on the received packet to know where the packet came from. Also use ***getData()*** to retrieve the data, and ***getLength()*** to see how many bytes were in the data

The received packet could be truncated to fit the buffer

Example 4: The following examples show how to create a UdpEchoServer and the corresponding UdpEchoClient.

```
import java.net.*;
import java.io.*;

public class UdpEchoServer
{
        static final int port = 9999;
        static final int packetSize = 1024;

public static void main(String args[]) throws SocketException{
                DatagramPacket packet;
                DatagramSocket socket;
                byte[] data;
                int clientPort;
                InetAddress address;
                String str;
                int recvSize;
                socket = new DatagramSocket(port);
                while(true){
                        data = new byte[packetSize];
                        // Create packets to receive the message
                        packet = new DatagramPacket(data,packetSize);
                        System.out.println("to receive the packets or port: "+port);
                        try{
                                socket.receive(packet);
                        }catch(IOException ie){
                                System.out.println(" Could not Receive:"+ie.getMessage());
```

```
                    System.exit(0);
                }
                // get data about client in order to echo data back
                address = packet.getAddress();
                clientPort = packet.getPort();
                recvSize = packet.getLength();
                str = new String(data,0,recvSize);
        System.out.println("Message  from "+ address+":"+clientPort+": "+str.trim());
                // echo data back to the client
        packet = new DatagramPacket(data,recvSize,address,clientPort);
try{            socket.send(packet);  }
catch(IOException ex){
        System.out.println("Could not Send "+ex.getMessage());
        System.exit(0);
                }               }       }   }

import java.net.*;
import java.io.*;

public class UdpEchoClient {
        static final int packetSize = 1024;
        static BufferedReader stdin = new BufferedReader(new
InputStreamReader(System.in));
        public static void main(String args[]) throws UnknownHostException,
SocketException{
                DatagramSocket socket;
                DatagramPacket packet;
                InetAddress address;
                String messageSend;
                String messageReturn;
                byte[] data;
```

```
int port;
try {
        System.out.print("Enter server name: ");
        address = InetAddress.getByName(stdin.readLine());
        System.out.print("Enter server port: ");
        port = Integer.parseInt(stdin.readLine());
        while (true) {
        System.out.print("Enter message for the server or press enter to exit: ");
        messageSend = stdin.readLine();
        if(messageSend.length() == 0){ System.exit(0); }
                socket = new DatagramSocket();
                data = messageSend.getBytes();
                packet = new DatagramPacket(data,data.length,address,port);
                socket.send(packet);
                //packet is reinitialized to use it for recieving
                data = new byte[packetSize];
                packet = new DatagramPacket(data,data.length);
                socket.receive(packet);
                messageReturn = new String(data,0,packet.getLength());
                System.out.println("Message Returned : "+ messageReturn);
        }
}catch(IOException iee){
        System.out.println("Could not receive :       "+iee.getMessage() );
        System.exit(0);
}       }     }
```

3. Multicast Sockets

Multicasting is achieved in Java using the same **DatagramPacket** class used for normal Datagrams. This is used together with the **MulticastSocket** class. The **MulticastSocket** class is

a subclass of the **DatagramSocket** class, with the following methods for joining and leaving a multicast group added.

void joinGroup(InetAddress mcastaddr)

void leaveGroup(InetAddress mcastaddr)

void setTimeToLive(int ttl)

Example 5: The following examples shows a sample Multicast sender and receiver.

```
import java.io.*;
import java.net.*;

public class MulticastReceiver {

  static MulticastSocket receiver;
  static InetAddress group;

  public static void main(String[] args) {
    try {
          group = InetAddress.getByName("224.100.0.5");
           receiver = new MulticastSocket(9095);
           System.out.println("Joined group at 224.100.0.5");
           receiver.joinGroup(group);
        while (true) {
            byte[] buffer = new byte[1024];
             DatagramPacket recvPacket = new DatagramPacket(buffer, buffer.length);
             receiver.receive(recvPacket);
 String message = new String(buffer, 0, recvPacket.getLength());
if (message.length() == 0) {
        receiver.leaveGroup(group);   receiver.close();      break;                  }
             System.out.println(message);   }      }
     catch (Exception ex) {
```

```
        System.out.println("Exception: "+ ex);        }    }    }

import java.io.*;
import java.net.*;
public class MulticastSender {
  public static void main(String[] args) {
    MulticastSocket sender;
    InetAddress group;
        try {
      group = InetAddress.getByName("224.100.0.5");
      sender = new MulticastSocket(9095);
      sender.setTimeToLive(32);
        BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
      while (true) {
                        System.out.print("Enter message for Multicast: ");
      String message = stdin.readLine();
                        if (message.length() == 0) {
                                sender.leaveGroup(group);
                                sender.close();
                                break;
                        }
  DatagramPacket packet = new DatagramPacket(message.getBytes(), message.length(),
group, 9095);
      sender.send(packet);
                }    }
   catch (Exception ex) {    System.out.println("Exception: "+ ex);     }      }      }
```

Remote Method Invocation

- A primary goal of RMI is to allow programmers to develop distributed Java programs
- RMI is the Java Distributed Object Model for facilitating communications among distributed objects

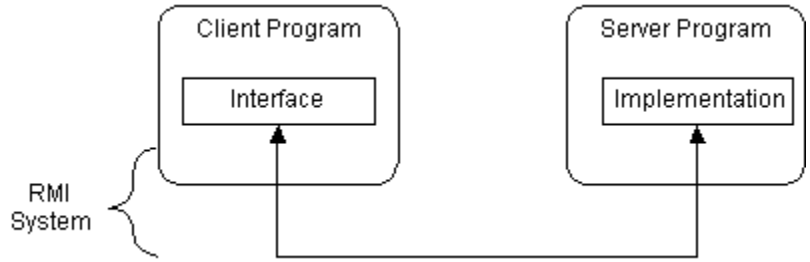
- RMI is a higher-level API built on top of sockets
- Socket-level programming allows you to pass data through sockets among computers
- RMI enables you not only to pass data among objects on different systems, but also to invoke methods in a remote object

The Differences between RMI and Traditional Client/Server Approach

- RMI component can act as both a client and a server, depending on the scenario in question.
- RMI system can pass functionality from a server to a client and vice versa. A client/server system typically only passes data back and forth between server and client.

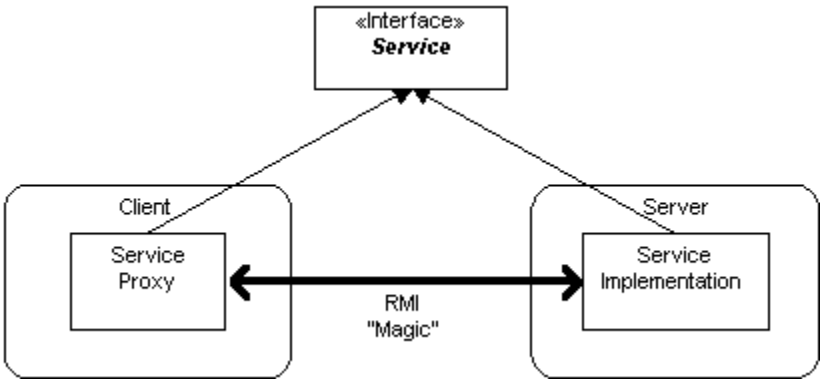
Interfaces

- In RMI, the definition of a remote service is coded using a Java interface.
- The implementation of the remote service is coded in a class
- The key to understanding RMI is to remember that interfaces define behavior and classes define implementation.



RMI supports two classes:

1. The first class is the implementation of the behavior, and it runs on the server.
2. The second class acts as a proxy for the remote service and it runs on the client.

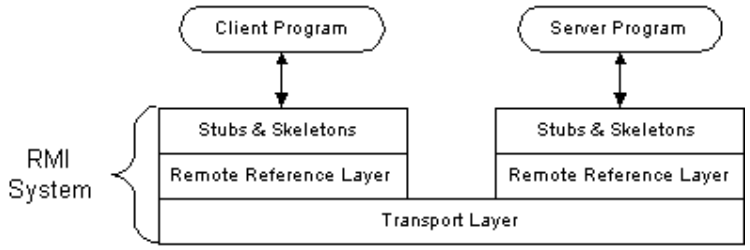


A client program makes method calls on the proxy object, RMI sends the request to the remote JVM, and forwards it to the implementation. Any return values provided by the implementation are sent back to the proxy and then to the client's program.

RMI Architecture Layers

The RMI implementation is built from three abstraction layers.

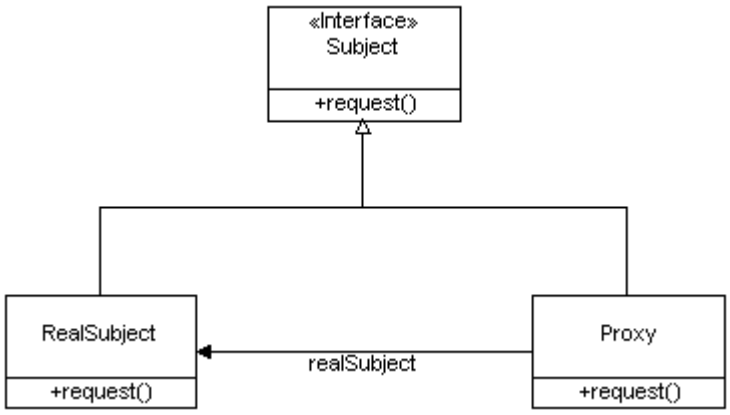
- The first is the Stub and Skeleton layer. This layer intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service.
- The next layer is the Remote Reference Layer. This layer understands how to interpret and manage references made from clients to the remote service objects. In JDK 1.1, this layer connects clients to remote service objects that are running and exported on a server. The connection is a one-to-one (unicast) link. In the Java 2 SDK, this layer was enhanced to support the activation of dormant remote service objects via Remote Object Activation.
- The Transport Layer is based on TCP/IP connections between machines in a network. It provides basic connectivity, as well as some firewall penetration strategies.



1. ***A server object is registered with the RMI registry;***
2. ***A client looks through the RMI registry for the remote object;***
3. ***Once the remote object is located, its stub is returned in the client;***
4. ***The remote object can be used in the same way as a local object. The communication between theclient and the server is handled through the stub and skeleton.***

Stub and Skeleton Layer

In this layer, RMI uses the Proxy design pattern. In the Proxy pattern, an object in one context is represented by another (the proxy) in a separate context. The proxy knows how to forward method calls between the participating objects. The following class diagram illustrates the Proxy pattern.



In RMI's use of the Proxy pattern, the stub class plays the role of the proxy, and the remote service implementation class plays the role of the RealSubject.

A skeleton is a helper class that is generated for RMI to use. The skeleton understands how to communicate with the stub across the RMI link. The skeleton carries on a conversation with the stub; it reads the parameters for the method call from the link, makes the call to the remote service implementation object, accepts the return value, and then writes the return value back to the stub.

Remote Reference Layer

The Remote Reference Layers defines and supports the invocation semantics of the RMI connection. This layer provides a RemoteRef object that represents the link to the remote service implementation object.

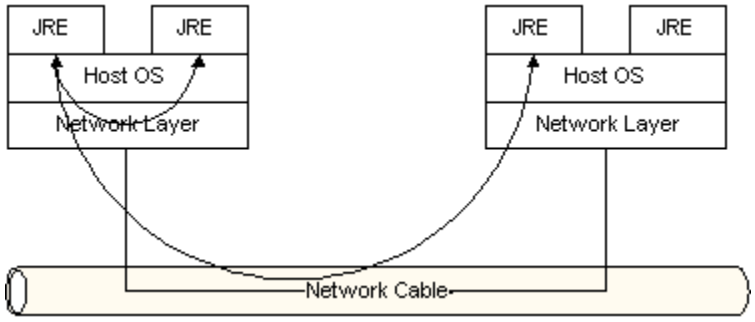
The stub objects use the invoke() method in RemoteRef to forward the method call. The RemoteRef object understands the invocation semantics for remote services.

The JDK 1.1 implementation of RMI provides only one way for clients to connect to remote service implementations: a unicast, point-to-point connection. Before a client can use a remote service, the remote service must be instantiated on the server and exported to the RMI system.

The Java 2 SDK implementation of RMI adds a new semantic for the client-server connection. In this version, RMI supports activatable remote objects. When a method call is made to the proxy for an activatable object, RMI determines if the remote service implementation object is dormant. If it is dormant, RMI will instantiate the object and restore its state from a disk file. Once an activatable object is in memory, it behaves just like JDK 1.1 remote service implementation objects.

Transport Layer

The Transport Layer makes the connection between JVMs. All connections are stream-based network connections that use TCP/IP. Even if two JVMs are running on the same physical computer, they connect through their host computer's TCP/IP network protocol stack. The following diagram shows the unfettered use of TCP/IP connections between JVMs.



As you know, TCP/IP provides a persistent, stream-based connection between two machines based on an IP address and port number at each end. Usually a DNS name is used instead of an IP address; this means you could talk about a TCP/IP connection between flicka.magelang.com:3452 and rosa.jguru.com:4432. In the current release of RMI, TCP/IP connections are used as the foundation for all machine-to-machine connections.

On top of TCP/IP, RMI uses a wire level protocol called Java Remote Method Protocol (JRMP). JRMP is a proprietary, stream-based protocol that is only partially specified is now in two versions. The first version was released with the JDK 1.1 version of RMI and required the use of Skeleton classes on the server. The second version was released with the Java 2 SDK. It has been optimized for performance and does not require skeleton classes.Some other changes with the Java 2 SDK are that RMI service interfaces are not required to extend from java.rmi.Remote and their service methods do not necessarily throw RemoteException.

RMI can use many different directory services, including the Java Naming and Directory Interface (JNDI). RMI itself includes a simple service called the RMI Registry, rmiregistry. The RMI Registry runs on each machine that hosts remote service objects and accepts queries for services, by default on port 1099.

On a host machine, a server program creates a remote service by first creating a local object that implements that service. Next, it exports that object to RMI. When the object is exported, RMI creates a listening service that waits for clients to connect and request the service. After exporting, the server registers the object in the RMI Registry under a public name.

On the client side, the RMI Registry is accessed through the static class Naming. It provides the method lookup() that a client uses to query a registry. The method lookup() accepts a URL that specifies the server host name and the name of the desired service. The method returns a remote reference to the service object. The URL takes the form:

rmi://<host_name>
    [:<name_service_port>]
        /<service_name>

where the host_name is a name recognized on the local area network (LAN) or a DNS name on the Internet. The name_service_port only needs to be specified only if the naming service is running on a different port to the default 1099.

Build a simple remote calculator service and use it from a client program.

A working RMI system is composed of several parts.

- Interface definitions for the remote services
- Implementations of the remote services
- Stub and Skeleton files
- A server to host the remote services
- An RMI Naming service that allows clients to find the remote services
- A class file provider (an HTTP or FTP server)
- A client program that needs the remote services

To simplify things, you will use a single directory for the client and server code. By running the client and the server out of the same directory, you will not have to set up an HTTP or FTP server to provide the class files.Assuming that the RMI system is already designed, you take the following steps to build a system:

1. Write and compile Java code for interfaces
2. Write and compile Java code for implementation classes
3. Generate Stub and Skeleton class files from the implementation classes
4. Write Java code for a remote service host program
5. Develop Java code for RMI client program
6. Install and run RMI system

1. Interfaces: First, write and compile the Java code for the service interface. The Calculator interface defines all of the remote features offered by the service:

```
public interface Calculator extends java.rmi.Remote {

   public long add(long a, long b) throws java.rmi.RemoteException;

public long sub(long a, long b) throws java.rmi.RemoteException;

public long mul(long a, long b) throws java.rmi.RemoteException;

public long div(long a, long b) throws java.rmi.RemoteException; }
```

2. Implementation

Next, write the implementation for the remote service. This is the CalculatorImpl class:

```
public class CalculatorImpl extends  java.rmi.server.UnicastRemoteObject
   implements Calculator {
    //Implementations must have an explicit constructor in order to declare the
   //RemoteException exception
   public CalculatorImpl() throws java.rmi.RemoteException {
      super();  }
  public long add(long a, long b) throws java.rmi.RemoteException { return a + b; }
 public long sub(long a, long b) throws java.rmi.RemoteException { return a - b; }
 public long mul(long a, long b) throws java.rmi.RemoteException { return a * b; }
   public long div(long a, long b) throws java.rmi.RemoteException { return a / b; } }
```

The implementation class uses UnicastRemoteObject to link into the RMI system. In the example the implementation class directly extends UnicastRemoteObject. This is not a requirement. A class that does not extend UnicastRemoteObject may use its exportObject() method to be linked into RMI.

When a class extends UnicastRemoteObject, it must provide a constructor that declares that it may throw a RemoteException object. When this constructor calls super(), it activates code in UnicastRemoteObject that performs the RMI linking and remote object initialization.

3. Stubs and Skeletons: You next use the RMI compiler, rmic, to generate the stub and skeleton files. The compiler runs on the remote service implementation class file.

```
>rmic CalculatorImpl
```

4. Host Server: Remote RMI services must be hosted in a server process. The class CalculatorServer is a very simple server that provides the bare essentials for hosting.

```
import java.rmi.Naming;
```

```
public class CalculatorServer {
  public CalculatorServer() {
  try {
     Calculator c = new CalculatorImpl();
     Naming.rebind("rmi://localhost:1099/CalculatorService", c);
   } catch (Exception e) { System.out.println("Trouble: " + e); } }
public static void main(String args[]) { new CalculatorServer(); } }
```

5.Client The source code for the client follows:

```
import java.rmi.Naming;  import java.rmi.RemoteException;
import java.net.MalformedURLException; import java.rmi.NotBoundException;
public class CalculatorClient {
   public static void main(String[] args) {
      try {
         Calculator c = (Calculator) Naming.lookup("rmi://localhost/CalculatorService");
         System.out.println( c.sub(4, 3) );
         System.out.println( c.add(4, 5) );
         System.out.println( c.mul(3, 6) );
         System.out.println( c.div(9, 3) );
      }
      catch (MalformedURLException murle) {
         System.out.println();
         System.out.println("MalformedURLException");
         System.out.println(murle);        }
```

```
        catch (RemoteException re) {
            System.out.println();
            System.out.println("RemoteException");
            System.out.println(re);         }
    catch (NotBoundException nbe) {
            System.out.println();
            System.out.println("NotBoundException");
            System.out.println(nbe);         }
        catch ( java.lang.ArithmeticException ae) {
            System.out.println();
            System.out.println("java.lang.ArithmeticException");
            System.out.println(ae);         }     } }
```

6.Running the RMI System

You are now ready to run the system! You need to start three consoles, one for the server, one for the client, and one for the RMIRegistry.

Start with the Registry. You must be in the directory that contains the classes you have written. From there, enter the following: > rmiregistry

If all goes well, the registry will start running and you can switch to the next console.

In the second console start the server hosting the CalculatorService, and enter the following:

>java CalculatorServer

It will start, load the implementation into memory and wait for a client connection.

In the last console, start the client program. >java CalculatorClient