Unit - 4
Java Networking & JDBC

Topics to Cover:

*) Socket programming in Java - Inet Address and URL classes
*) TCP and UDP Protocols in Java
*) Server Socket and Socket classes Multithreaded Servers
Handling multiple client connection
*) RMI
*) JDBC

Concept overview (Theory part of the unit) and codes

1) Socket Programming in Java

. Socket Programming enables communication between two machines over a network
Types: Connection oriented -> uses TCP, classes
o) Socket (Client)
o) ServerSocket (Server)
connection-less UDP
o) DatagramSocket
o) datagram packet
client Requirements:
*) IP Address of Server
*) Port number
working of TCP Sockets:
*) Server waits with ServerSocket.accept() -> Returns a Socket
*) client initiates connection using Socket(ip, port)
*) Both IP uses input and output Streams to communicate
Socket class (Purpose):
Represents an endpoint for communication

M T W T F S S

Page No.: YOUVA

Date:

ServerSocket class (purpose)

Listens for client requests and establishes connection

Code: Server side

```
import java.io.*;
import java.net.*;
public class SimpleServer {
    public static void main (String[] args) throws Exception {
        ServerSocket ss = new ServerSocket(5000);
        S.O.P ("Server waiting...");
        Socket s = ss.accept();
        DataInputStream in = new DataInputStream
                (s.getInputStream());
        String msg = in.readUTF();
        S.O.P ("Client says :" + msg);
        s.close();
        ss.close();
```

Code: client side

```
import java.io.*;
import java.net.*;
public class SimpleClient {
    public static void main (String[] args) throws Exception {
        Socket s = new Socket ("127.0.0.1", 5000);
        DataOutputStream out = new DataOutputStream(Socket.
                getOutputStream());
        out.writeUTF ("Hello Server");
        out.flush()
        Socket.close(); ss
```

Output

Server is waiting

client says Hello Server

M T W T F S S

Page No.: YOUVA

Date:

2) Inet Address & URL classes

Inet Address : Represents an IP address + host name

Used for :

- Getting IP of a hostname
- DNS lookup

IP Address Types

- a) IPV4 -> 32 bit, Widely used 4 million + addresses
- b) IPV6 -> 128 bit, Supports extremely large address space

TCP/IP

- a) TCP -> reliable, Connection-oriented, Ensures correct ordered delivery
- b) IP -> addressing & routing packets

URL class

*) Uniform Resource Locator -> Represents a web resource

URL Contains:

1) protocol (http, https)
2) Host name / IP
3) port (Optional)
4) File / path

Common methods used in URL :

a) getProtocol()	https
b) getHost()	www.google.com
c) getPort()	-1
d) getFile()	/search?q=Java
e) getPath()	/search
f) getQuery()	q=Java
g) getDefaultPort()	443

Code: Inet example

```
import java.net.*;
public class InetDemo {
    public static void main(String[] args) throws Exception {
        InetAddress ip = InetAddress.getByName("www.google.com");
        S.O.P("Host name :" + ip.getHostName());
        S.O.P("IP Address :" + ip.getHostAddress());
    }
}
```

Output

Host name : www.google.com

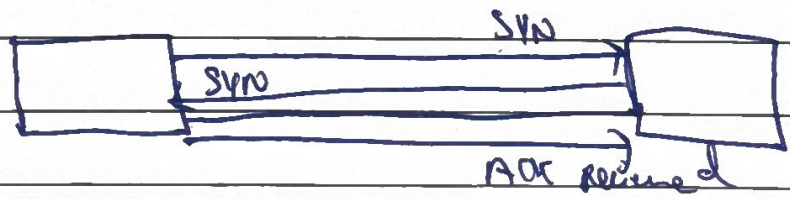
IP Address : 192.250.182.36

Code: URL Example

```
import java.net.*;
public class URLDemo {
    public static void main(String[] args) throws Exception {
        URL url = new URL("...");
        url.getProtocol;
        like wise use other functions
```

TCP vs UDP

1) TCP (Transmission Control Protocol)
   a) connection-oriented
   b) uses 3-way handshake



Connection oriented protocol

a) Reliable, ordered delivery
b) Resends lost packets
c) Suitable for: web pages, file downloads

UDP (User datagram protocol)

* Connection-less
* No guarantee of delivery (fire and forget)
* Faster than TCP
* Suitable for: video streaming, gaming, VOIP
* Info → Byte array → Datagram Socket → Send to port

Code: Server.

```
import java.net.*;
class DatagramServer {
    public static void main(String[] args) throws Exception {
        DatagramSocket dgs = new DatagramSocket();
        String Msg = "Hi Client!";
        InetAddress ip = InetAddress.getByName("localHost");
        DatagramPacket dgp = new DatagramPacket(Msg.getbyte(),
                                Msg.length(), ip, port:3000);
        dgs.send(dgp);
        dss.close();
    }
}
```

1) Create Socket
2) Create Buffer Array
3) Convert String

client:

```
import java.net.*;
class DatagramClient{
    public static void main(String[] args) throws Exception{
        DatagramSocket dgs = new DatagramSocket(port: 5000);
        byte[] buffer = new byte[1024];
        DatagramPacket dgp = new DatagramPacket(buffer, length: 1024);
        dgs.receive(dgp);
        String msg = new String(dgp.getData());
        S.O.p(msg);
        dgs.close();
```

Output: Hi client!

Server Socket & Socket class

Server Socket:

used on server side to listen for client connections

Important methods

1) Bind() - binds socket to IP & to port
2) accept()
3) getLocalPort
4) getInetAddress
5) close()

Socket:

Represents the client-server communication endpoint

M T W T F S S

Page No.: YOUVA

Date:

Multithreaded Server

Need:

To handle multiple clients simultaneously

How it works:

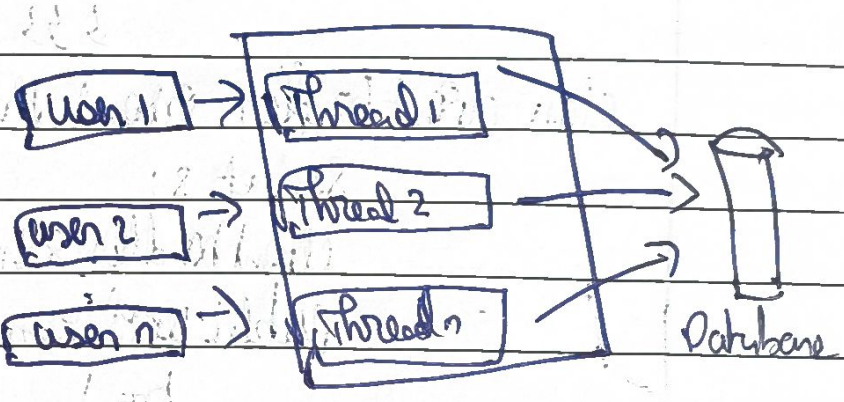
- Server accepts connection
- Creates one thread per client
- Each thread independently handles communication

Advantages:

1) Faster response
2) Users don't wait
3) Errors in one thread don't affect other

Disadvantages:

1) Complex code
2) Managing threads



Handling multiple client connections

*) Server runs an infinite loop calling accept()
*) For every new client:
    *) A new thread / Runnable executes communication logic
*) Main server thread continues listening

Client Code:

```
import java.io.*;
import java.net.*;
public class MultiClient {
    public static void main(String[] args) throws Exception {
        Socket s = new Socket("127.0.0.1", 6000);
        DataOutputStream out = new "" (s.getOutputStream());
        out.writeUTF("Hello from client");
        out.flush();
    } }
        s.close();
```

Code: Server

```
import java.io.*;
import java.net.*;
public class multiserver {
        public static void main (String [] args) throws Exception {
            ServerSocket ss = new "" (6000);
            S.O.P ("Server waiting, Running");
            while (true) {
                ss
                Socket s = ss.accept();
                new clientHandler (Socket).start();
            }}
class clientHandler extends thread {
        Socket s;
        clientHandler (Socket s) { this.socket = s; }
        public void run () {
            try {
                DIS in = new DIS (S.getInputStream());
                String msg = in.readUTF();
                S.O.P ("Client :" +msg);
                s.close();
            } Catch (Exception e) {
                S.O.P (e);
```

Output : Multiclient Server Running

Interconversion of code (Dont use this

Code : Server side

```
class multithreading extends Thread {
    public void run()
    {
    try {
        S.O.p ("Thread " + Thread.currentThread().getID() + "is running");
    }
    catch (Exception e ){
        S.O.p ("Exception is caught "); } }
public class multithread {
        public static void main (String[] args) {
            int n = 8;
            for (int i=0; i<n; i++) {
                multithreading obj = new ....();
                obj.start();
    } }
```

Code: client side

```
public class server {
    public static void Main (String[] args) throws IO Exception {
        ServerSocket ss = null;
        boolean listening = true;
        try { ServerSocket = new ServerSocket(4444); }
        Cath { IO exception e }
    while (listening)
            new Thread (new clientHandler (ServerSocket.
                                accept()).start();
        ss.close();
    }
}
```

M T W T F S S

Page No.: YOUVA

Date:

Introduction to RMI (Remote Method Invocation)

*) Purpose : Allows methods calls on object located in another JVM
*) Use : distributed applications

Stub & Skeleton (Concept)

Stub (Client side)
Acts as proxy for remote object
- o) Sends method call
- o) Sends parameters
- o) waits for result
- o) Returns result

Skeleton (Server side)
- o) Receives request
- o) Invokes Actual method
- o) Sends result back

Steps to Build RMI Application
1.) create remote interface (extends Remote)
2) Implement the remote interface
3) Generate Stub classes (using rmic)
4) Start RMI Registry
5) Create & run RMI Server
6) Create & run RMI client

RMI Registry

*) Acts as a name service where remote objects are registed and looked up by clients
Key Actions
- o) Server binds Stub to a name
- o) client performs lookup(name) to get Stub
- o) Client calls methods on Stub

Default port : 1099

M T W T F S S

Page No :

Date:

YOUVA

RMI Object Serialization

Serialization:

Process of converting an object into byte stream for:

- o) network transfer
- o) file storage

used in RMI for

- o) Sending objects as parameters
- o) Receiving objects as return values

Serialization Rules

- o) class must implement Serializable
- o) transient fields are not serialized
- o) can customize with writeObject() & readObject()

RMI Code:

Interface:

```
import java.rmi*
public Interface Adder extends Remote{
        int add (int a, int b) throws RemoteException;
}
```

Implementation:

```
import java.rmi.Server.*;
Import java.rmi*
public class AdderRemote extends UnicastRemoteObject implements Adder{
        AdderRemote() throws RemoteException {}
        public int add (int a, int b) {
                return a+b;
        }
}
```

Server program:

```
Import java.rmi.*;
public class RMIServer {
        public static void main (String [] args) throws exception {
                Adder stub = new AdderRemote ();
                naming.rebind ("rmi://localHost/add", stub);
                S.O.p ("RMI Server ready");
```

output:

RMI Server ready

Client program:

```
import java.rmi.*;
public class RMIclient {
        public static void main (String[] args) throws exception {
                Adder stub = (Adder) naming.lookup ("");
                S.O.p ("Sum = " + stub.add (10, 5));
        }
}
```

Sample output

Sum = 15

M T W T F S S

Page No.: YOUVA

Date:

JDBC overview

Definition: Java API to interact with relational databases

Uses: ○ Connect to DB
○ Execute SQL
○ Insert / update / delete data

Popular JDBC Interfaces:
○) Driver
○) Connection
○) Statement
○) Prepared Statement
○) Callable Statement
○) Result Set
○) ResultSet metaData
○) DatabaseMetaData

Types:
*) JDBC - ODBC
*) Native - API
*) Network protocol
*) Thin driver

Code:

```
import java.sql.*;
public class JDBCSelect{
        public static void (String[] args){
         try{
            Class.forname("com.mysql.cj.jdbc.Driver");
            Connection con = DriverManager.getConnection(
                                    JDBC.mysql:///, "root", "pw");
            Statement smt = con.createStatement();
            ResultSet RS = stms.executedquery("select * from book");
```

```
while (rs.next()){
        S.O.p (rs.getInt("id") + rs.getString("name"));
    }
    con.close();
}catch(Exception e){
        sop(e); }}}
```

Output

JallaBook John 500

Inserting value:

```
PS = Con. prepareStatement("Insert into books values (?,?,?,?)");
PS. SetInt(1,1);
PS. SetString(2, "Rahul");
ps. setInt(3, 85);
int rows = PS. executeUpdate();
```

→ place holders

deleting is also same, ? we will give conditions