

**A Project Report**

**On**

**“Implementation of Client-Server for**

**conversion using TCP”**

**Department of Computer Science & Engineering**

**NATIONAL INSTITUTE OF TECHNOLOGY PATNA**

**University Campus, Bihar - 800005**

**Submitted By:**

**Name**

**Roll No.**

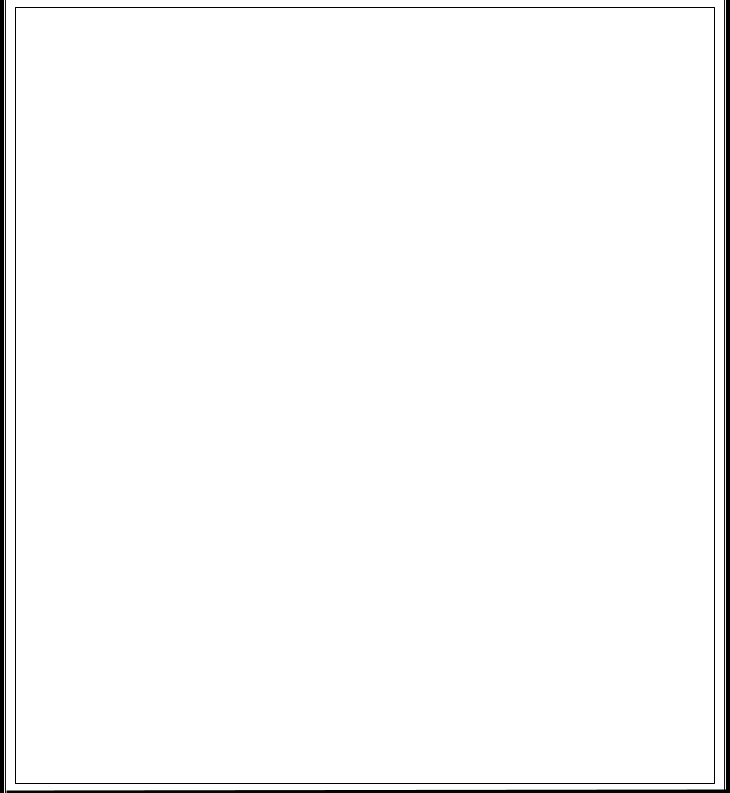
Anshu Kumari

2006039

**Course Code: CSL5403**

**Course Title: Computer Networks Lab**

**Branch: B.Tech CSE – A1**



**TABLE of CONTENTS**

**Contents**

**Page No.**

**1.**

**Introduction to TCP**

**4**

●

Layers of TCP

●

How it works?

●

What is it used for?

●

Importance of TCP

**2.**

**Introduction to Sockets**

**6**

●

Definition

●

Types of Socket

●

Where is it used?

**3.**

**Server**

**7**

●

Definition

●

Types of Server

●

How to make a Server?

**4.**

**Client**

**8**

●

Definition

●

How to make Client?

**5.**

**Client-Server Architecture using TCP**

**6.**

**Code 11**

●

Server Side

●

Client Side

**7.**

**Output 27**

●

Server Side

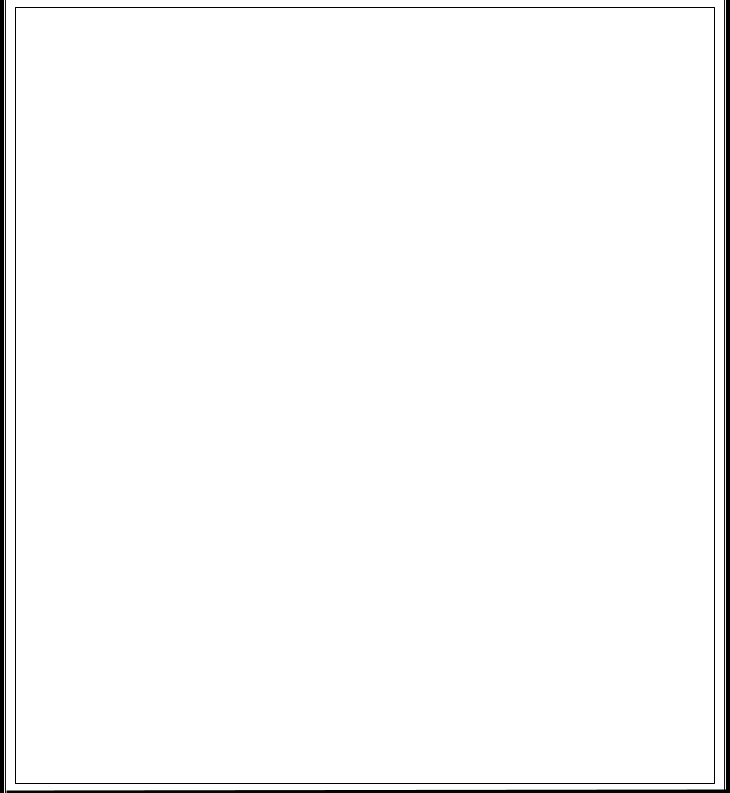
●

Client Side

**8.**

**Learning Outcome 28**

**9**



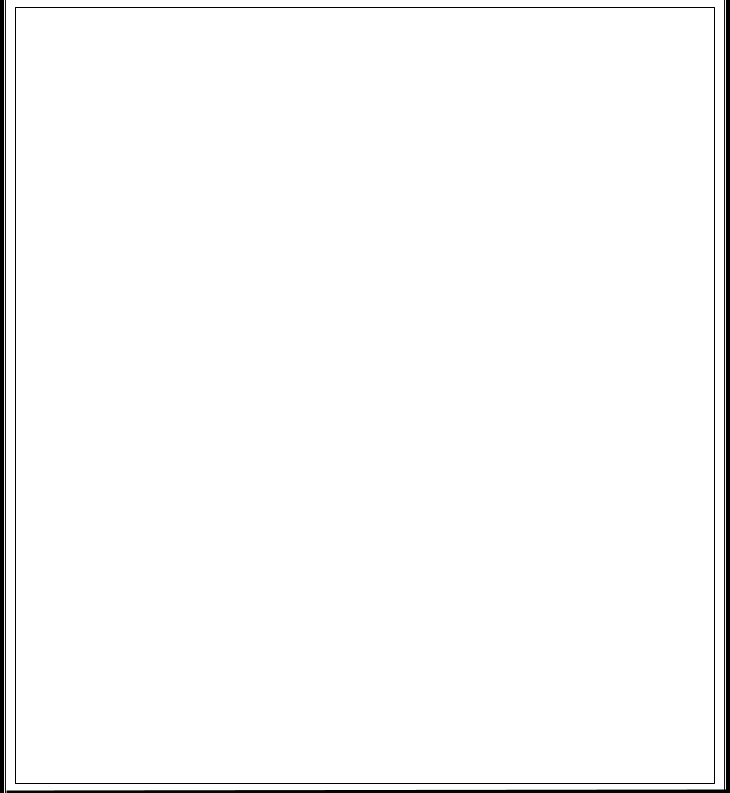
**AIM OF THE EXPERIMENT**

Implement a Client-Server based Prefix to Postfix,

Postfix to

Prefix, Prefix to Infix, Infix to Prefix, Postfix to Infix, and

Infix to Postfix conversion application using TCP.



**1.**

**INTRODUCTION TO TCP**

**What is the Transmission Control Protocol (TCP)?**

TCP

(

Transmission

Control

Protocol)

is

one

of

the

main

protocols

of

the

Internet

protocol

suite.

It

lies

between

the

Application

and

Network

layers,

which

are

used

in

providing

reliable

delivery

services.

It

is

a

connection-oriented

protocol

for

communications

that

helps

in

the

exchange

of

messages

between

the

different

devices

over a network.

**How it Works?**

TCP

is

a

connection-oriented

protocol,

which

means

a

connection

is

established

and

maintained until the applications at each end have finished exchanging messages.

TCP performs the following actions:

●

determines how to break application data into packets that networks can

deliver;

●

sends packets to, and accepts packets from, the network layer;

●

manages flow control;

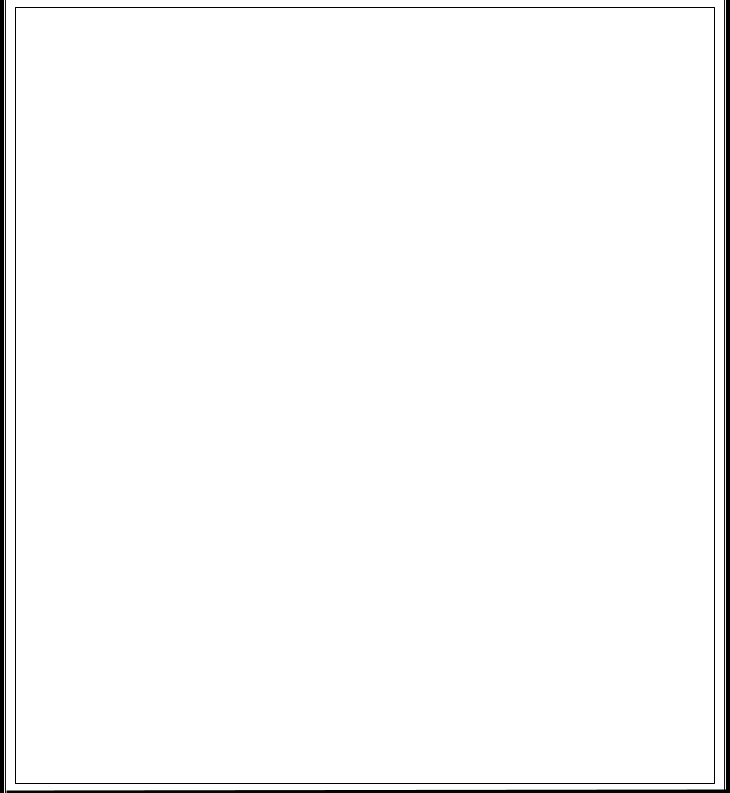
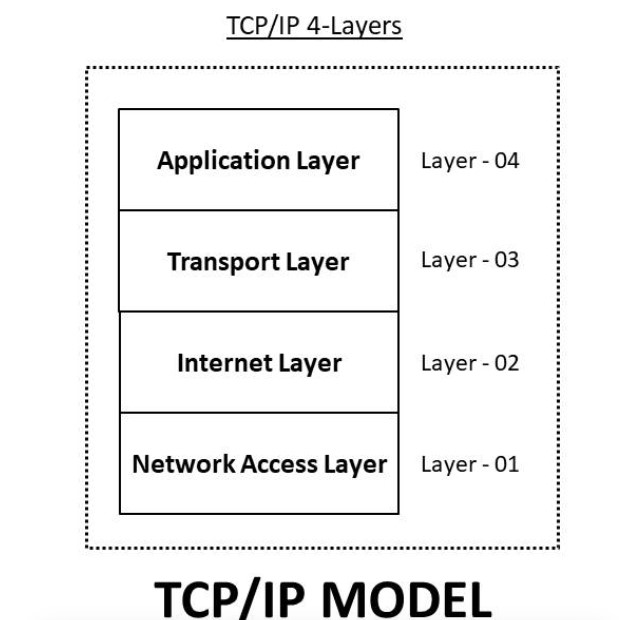
●

handles retransmission of dropped or garbled packets, as it's meant to

provide error-free data transmission; and

●

acknowledges all packets that arrive.



In

the

Open

Systems

Interconnection

[(](https://www.techtarget.com/searchnetworking/definition/OSI)

[OS](https://www.techtarget.com/searchnetworking/definition/OSI)

[I](https://www.techtarget.com/searchnetworking/definition/OSI)

)

communication

model,

TCP

covers

parts

of Layer 4, the transport layer, and parts of Layer 5, the session layer.

When

a

web

server

sends

an

[HTM](https://www.theserverside.com/definition/HTML-Hypertext-Markup-Language)

[L](https://www.theserverside.com/definition/HTML-Hypertext-Markup-Language)

file

to

a

client,

it

uses

the

hypertext

transfer

protocol

[(](https://www.techtarget.com/whatis/definition/HTTP-Hypertext-Transfer-Protocol)

[HTT](https://www.techtarget.com/whatis/definition/HTTP-Hypertext-Transfer-Protocol)

[P](https://www.techtarget.com/whatis/definition/HTTP-Hypertext-Transfer-Protocol)

)

to

do

so.

The

HTTP

program

layer

asks

the

TCP

layer

to

set

up

the

connection

and

send

the

file.

The

TCP

stack

divides

the

file

into

data

packets,

numbers them and then forwards them individually to the IP layer for delivery.

Although each packet in the transmission has the

same source and destination

[I](https://www.techtarget.com/whatis/definition/IP-address)

[P](https://www.techtarget.com/whatis/definition/IP-address)

[addres](https://www.techtarget.com/whatis/definition/IP-address)

[s](https://www.techtarget.com/whatis/definition/IP-address)

, packets may be sent along multiple routes.

The TCP program layer in the

client computer waits until all packets have arrived. It then acknowledges those it

receives and asks for the retransmission of any it does not, based on missing packet

numbers. The TCP layer then assembles the packets into a file and delivers the file

to the receiving application.

**What is it used ?**

TCP is used for organizing data in a way that ensures

the secure transmission

between the server and client. It guarantees the integrity of data sent over the

network, regardless of the amount. For this reason, it is used to transmit data from

other higher-level protocols that require all transmitted data to arrive.

Examples of these protocols include the following:

●

Secure Shell (

[SS](https://www.techtarget.com/searchsecurity/definition/Secure-Shell)

[H](https://www.techtarget.com/searchsecurity/definition/Secure-Shell)

)

, File Transfer Protocol (FTP),

[Telne](https://www.techtarget.com/searchnetworking/definition/Telnet)

[t](https://www.techtarget.com/searchnetworking/definition/Telnet)

:

For

peer-to-peer file sharing, and, in Telnet's case, logging into another user's

computer to access a file.

●

Simple Mail Transfer Protocol (

[SMT](https://www.techtarget.com/whatis/definition/SMTP-Simple-Mail-Transfer-Protocol)

[P](https://www.techtarget.com/whatis/definition/SMTP-Simple-Mail-Transfer-Protocol)

)

, Post Office

Protocol (POP),

Internet Message Access Protocol (IMAP): For sending and receiving

email.

●

HTTP: For web access.

These examples all exist at the application layer of the

[TCP/IP stac](https://www.techtarget.com/searchnetworking/definition/TCP-IP)

[k](https://www.techtarget.com/searchnetworking/definition/TCP-IP)

and send data

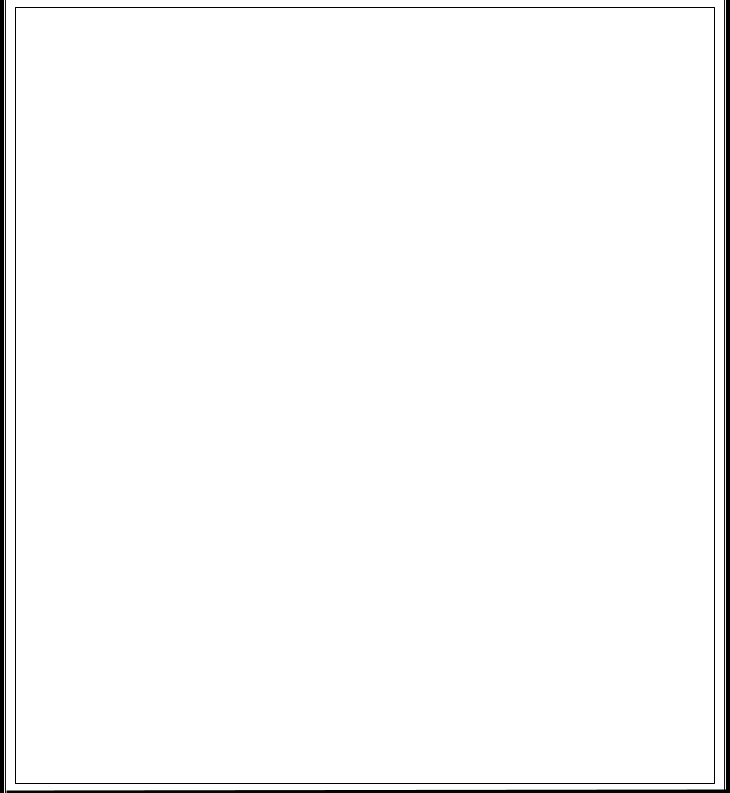
downwards to TCP on the transport layer.

# IImportance of TCP

TCP is important because it establishes the rules and standard procedures for the way information is communicated over the internet. It is the foundation for the internet as it currently exists and ensures that data transmission is carried out uniformly, regardless of the location, hardware or software involved.

TCP is flexible and highly scalable, meaning new protocols can be introduced to it, and it will accommodate them. It is also nonproprietary, meaning no one person or company owns it.

# INTRODUCTION TO SOCKETS

Sockets allow communication between two different processes on the same or different machines. To be more precise, it's a way to talk to other computers using standard Unix file descriptors. In Unix, all I/O actions are done by writing or reading to a file descriptor. A file descriptor is just an integer associated with an open file and it can be a network connection, a text file, a terminal, or something else.

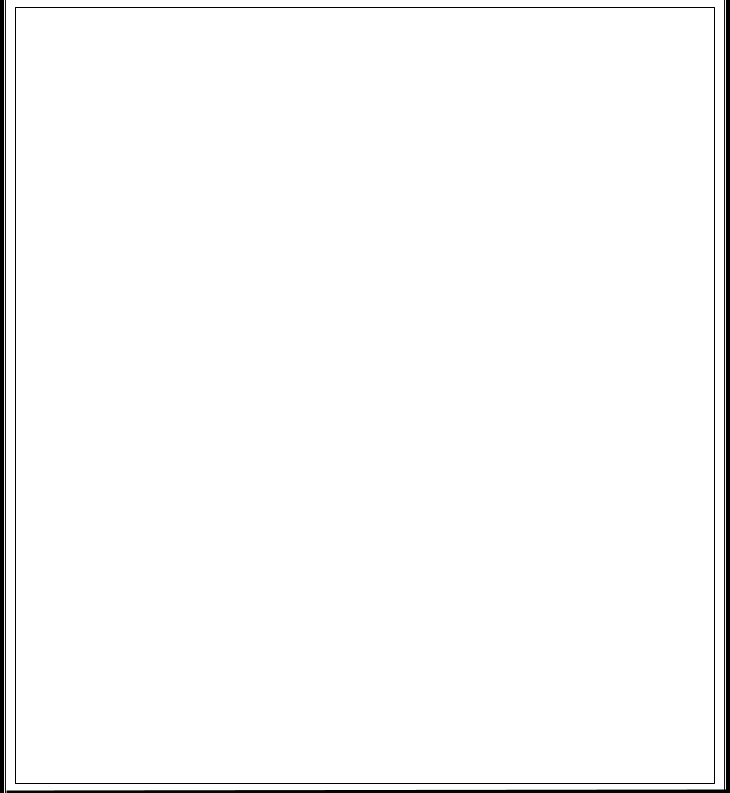
## Types of Socket

There are three types of sockets available to the users. The first two are most commonly used and last two are rarely used.

1. **Stream Sockets**: Delivery in a networked environment is guaranteed. If you send through the stream socket three items "A, B, C", they will arrive in the same order "A, B, C". These sockets use TCP (Transmission Control Protocol) for data transmission. If delivery is impossible, the sender receives an error indicator. Data records don’t have any boundaries.
2. **Datagram Sockets**: Delivery in a networked environment is not guaranteed.

They're connectionless because you don't need to have an open connection as in Stream Sockets - you build a packet with the destination information and send it out. They use UDP (User Datagram Protocol).

1. **Raw Sockets**: provides users access to the underlying communication protocols which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols.



**Where it is used?**

A

Unix

Socket

is

used

in

a

client

server

application

framework.

A

server

is

a

process

which

does

some

function

on

request

from

a

client.

Most

of

the

application

level

protocols

like

FTP,

SMTP

and

POP3

make

use

of

Sockets

to

establish

connection

between client and server and then for exchanging data.

**3**

**. SERVER**

In computing, a server is a computer program or a device that provides functionality

for called clients which are other programs or devices. This architecture is called the

client–server model. A single overall computation is distributed across multiple

processes or devices. Servers can provide various functionalities called services.

These services include sharing data or resources among multiple clients, or

performing computation for a client. Multiple clients can be served by a single server,

and a single client can use multiple servers. A client process may run on the same

device. It can also connect over a network to a server to run on a different device.

Examples of servers may include database servers, mail servers, print servers, file

servers, web servers, application servers, and game servers.

Most frequently client–server systems are implemented by the request–response

model., i.e., a client sends a request to the server. In this model server performs some

action and sends a response back to the client, typically with a result or

acknowledgement. Designating a computer as server-class hardware means that it is

specialized for running servers on it. This implies that it is more powerful and reliable

than standard personal computers. But large computing clusters may be composed of

many relatively simple, replaceable server components.

**Types of Server**

There are two types of servers you can have:

**1**

**. Iterative Server**

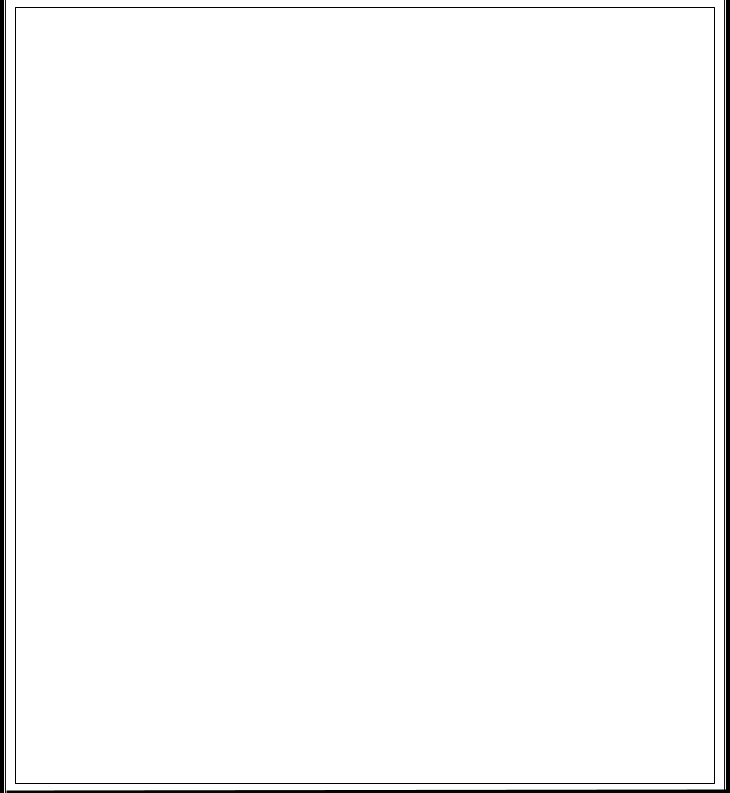
:

This is the simplest form of

server where a server process serves

one client and after completing first request then it takes request from another client.

Meanwhile another client keeps waiting.



**2**

**. Concurrent Servers**

:

This type of server runs multiple concurrent processes to

serve many requests at a time. Because one process may take longer and another client

cannot wait for so long. The simplest way to write a concurrent server under Unix is

to fork a child process to handle each client separately.

**How to make a server:**

The steps involved in establishing a socket on the server side are as follows:

1

. Create a socket with the socket() system call.

2

. Bind the socket to an address using the bind() system call. For a server socket on

the Internet, an address consists of a port number on the host machine.

3

. Listen for connections with the listen() system call.

4

. Accept a connection with the accept() system call. This call typically blocks until a

client connects with the server.

5

. Send and receive data using the read() and write() system calls.

**4**

**. CLIENT**

A client and server networking model is a model in which computers such as servers

provide the network services to the other computers such as clients to perform a user

based tasks. This model is known as client-server networking model.

A client is a program that runs on the local machine requesting service from the

server. A client program is a finite program means that the service started by the user

and terminates when the service is completed.

**How to make client:**

The system calls for establishing a connection are somewhat different for the client

and the server, but both involve the basic construct of a socket. The two processes

each establish their own sockets. The steps involved in establishing a socket on the

client side are as follows:

1

. Create a socket with the socket() system call.

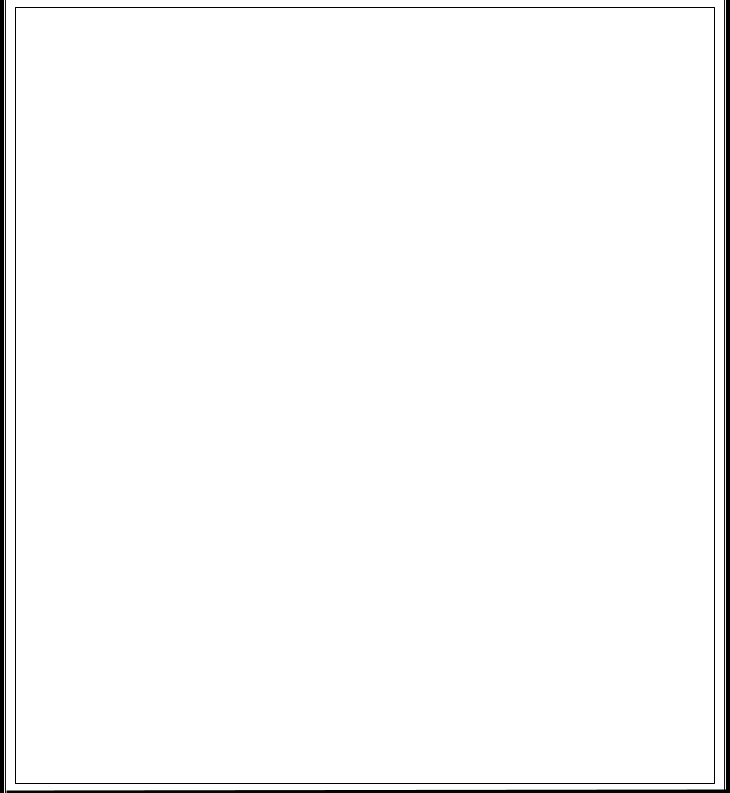
2

. Connect the socket to the address of the server using the connect() system call.

3

. Send and receive data. There are a number of ways to do this, but the simplest is to

use the read() and write() system calls.



**5**

**. CLIENT-SERVER ARCHITECTURE USING TCP**

The Transmission Control Protocol (TCP) provides full transport-layer services to

applications. TCP is a reliable stream transport protocol. The term stream, in this

context, means connection oriented:

A connection must be established between both ends of a transmission before either

can transmit data. At the sending end of each transmission, TCP divides a stream of

data into smaller units called segments. Each segment includes a sequence number for

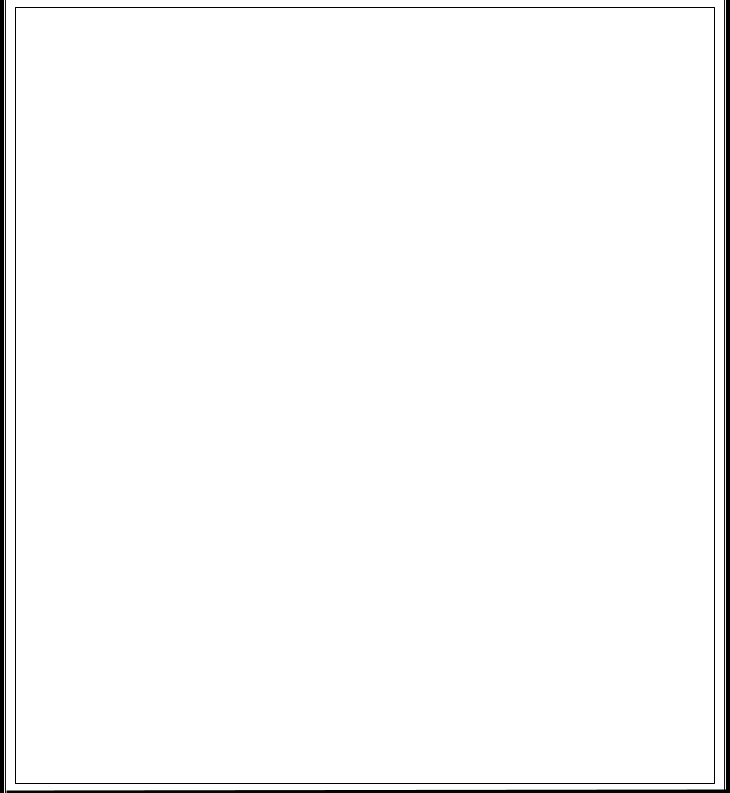
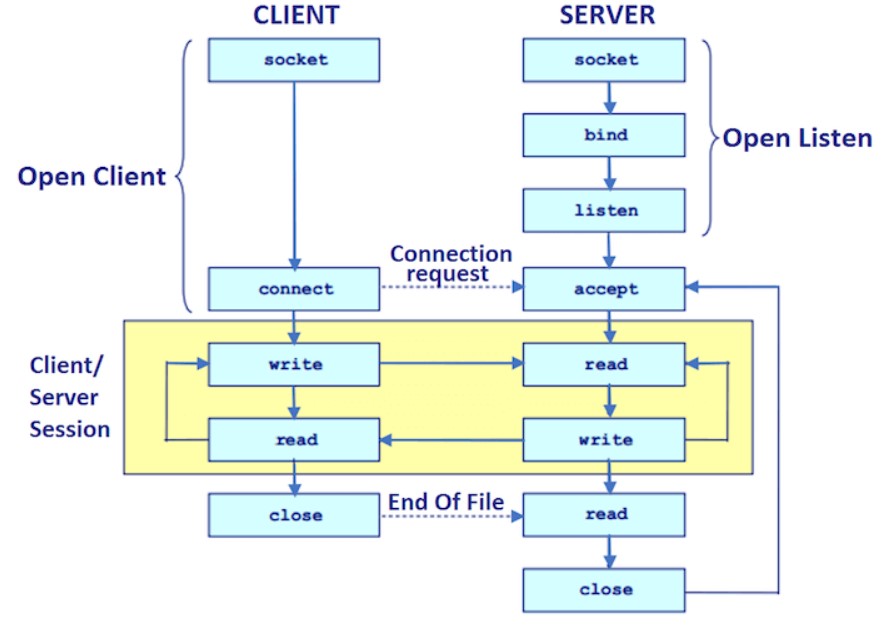
reordering after receipt, together with an acknowledgment number for the segments

received. Segments are carried across the internet inside of IP datagrams. At the

receiving end, TCP collects each datagram as it comes in and reorders the

transmission based on sequence numbers. TCP client and server architecture is shown

in Figure.



**Algorithmic steps for client server programming using TCP.**

**TCP SERVER:**

1

. Using create(), Create TCP socket.

2

. Using bind(), Bind the socket to the server address.

3

. Using listen(), put the server socket in a passive mode, where it waits for the client

to approach the server to make a connection

4

. Using accept(), At this point, connection is established between client and server,

and they are ready to transfer data.

5

. Go back to Step

3.

**TCP CLIENT:**

1

. Create a TCP socket.

2

. Connect newly created client socket to server.

**SYSTEM REQUIREMENTS**

**Hardware Requirements:**

•

2

GHz x86 processor

•

256

MB of system memory (RAM

)

•

100

MB of hard-drive space

•

Monitor to display output

•

Keyboard/Mouse for data input

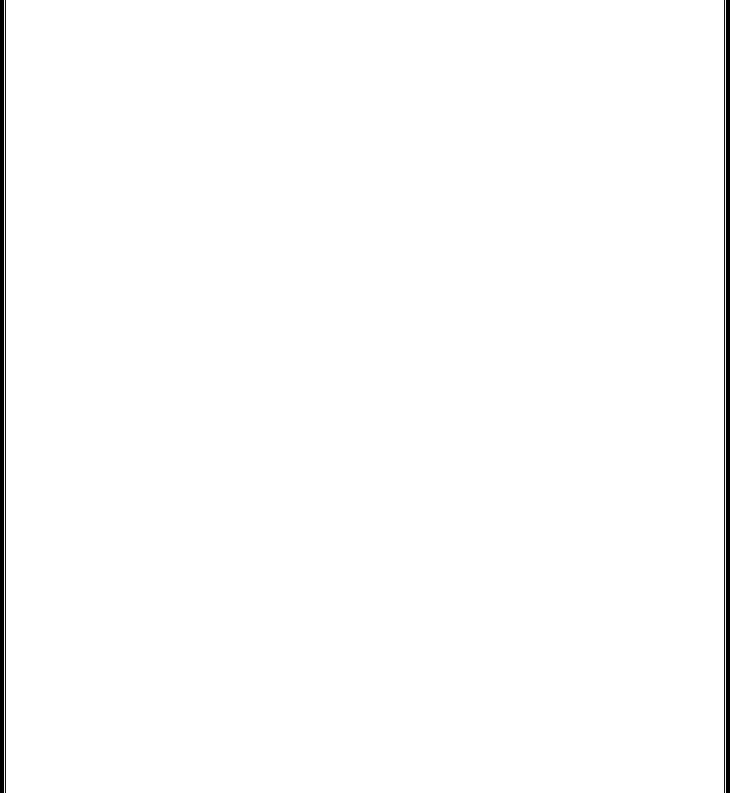
**Software Requirements:**

•

Java Compiler (javac compiler)

•

MS Word(Documentation)



**6.**

**CODE**

**6.1**

**. Server side implementation**

// Java program to illustrate Server

Side Programming

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.IOException;

import java.net.ServerSocket;

import java.net.Socket;

import java.util.Stack;

import java.util.StringTokenizer;

import java.util.\*;

class Server\_CNProj

{

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

// Convert prefix to Postfix

expression

static String preToPost(String

pre\_exp)

{

Stack<String> s = new

Stack<String>();

// length of expression

int length = pre\_exp.length();

// reading from right to left

for (int i = length - 1; i >= 0;

i--)

{

// check if symbol is operator

if

(

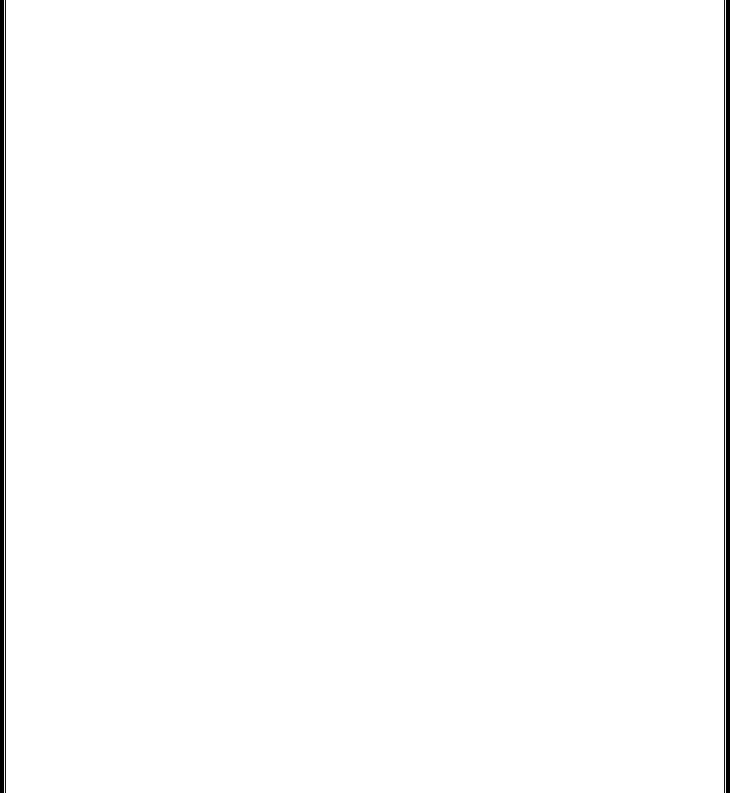
isOperator(pre\_exp.charAt

(i)))

{

// pop two operands from

stack



String op1 = s.peek();

s.pop();

String op2 = s.peek();

s.pop();

// concat the operands and

operator

String temp = op1 + op2 +

pre\_exp.charAt(i);

// Push String temp back

to stack

s.push(temp);

}

// if symbol is an operand

else {

// push the operand to the

stack

s.push(pre\_exp.charAt(i) +

"");

}

}

// stack contains only the

Postfix expression

return s.peek();

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Convert postfix to Prefix

expression

static String postToPre(String

post\_exp)

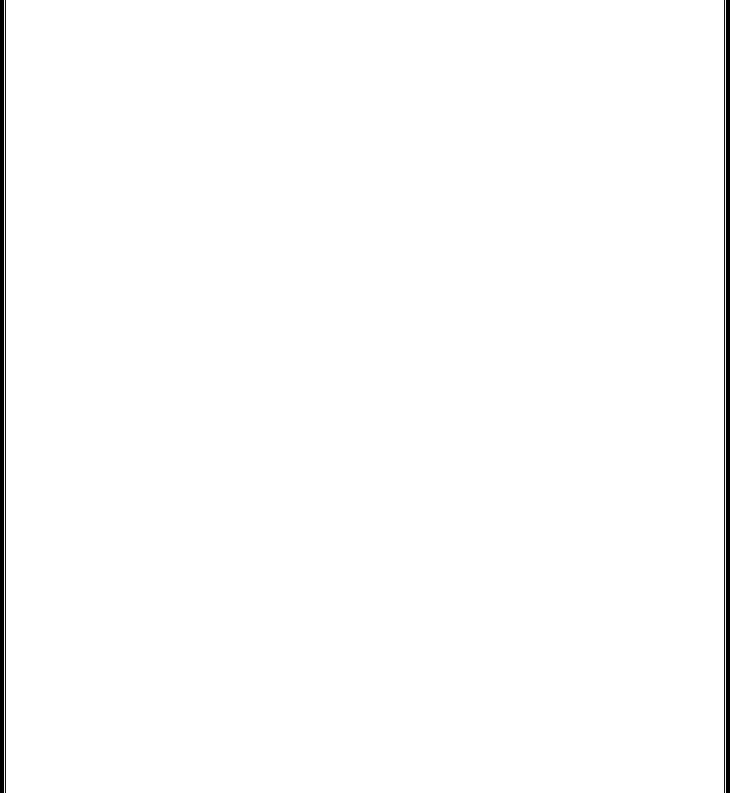
{

Stack<String> s = new

Stack<String>();

// length of expression

int length = post\_exp.length();



// reading from right to left

for (int i = 0; i < length; i++) {

// check if symbol is operator

if

(

isOperator(post\_exp.charAt(i)))

{

// pop two operands from

stack

String op1 = s.peek();

s.pop();

String op2 = s.peek();

s.pop();

// concat the operands and

operator

String temp

=

post\_exp.charAt(i)

+

op2 + op1;

// Push String temp back

to stack

s.push(temp);

}

// if symbol is an operand

else {

// push the operand to the

stack

s.push(post\_exp.charAt(i)

+

"");

}

}

// concatenate all strings in

stack and return the

// answer

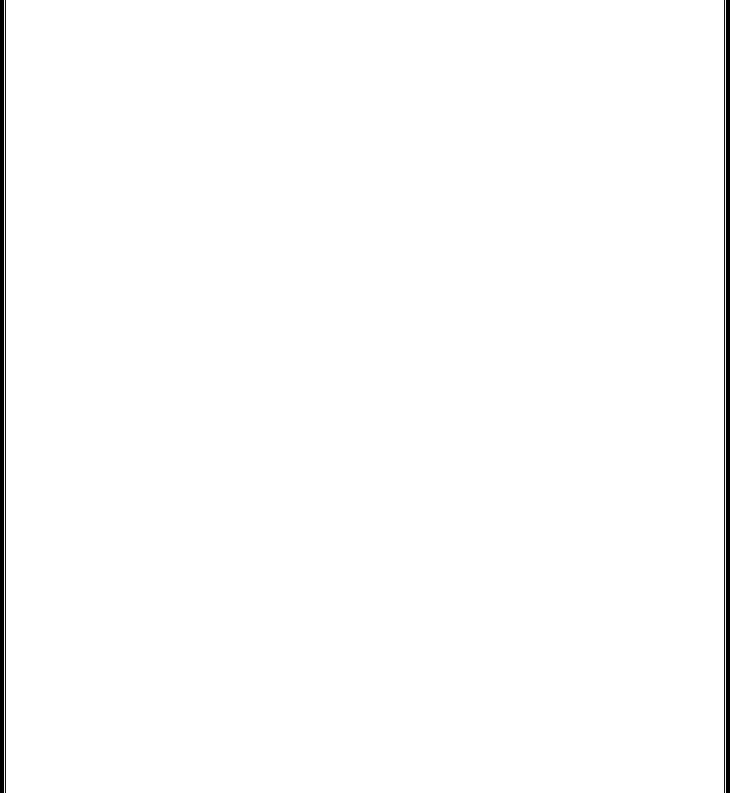
String ans = "";

for (String i : s)

ans += i;

return ans;

}



//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Convert prefix to Infix

expression

static String preToIn(String str)

{

Stack<String> stack = new

Stack<>();

// Length of expression

int l = str.length();

// Reading from right to left

for(int i = l - 1; i >= 0; i--)

{

char c = str.charAt(i);

if (isOperator(c))

{

String op1 = stack.pop();

String op2 = stack.pop();

// Concat the operands and

operator

String temp = "(" + op1 + c +

op2 + ")";

stack.push(temp);

}

else

{

// To make character to string

stack.push(c + "");

}

}

return stack.pop();

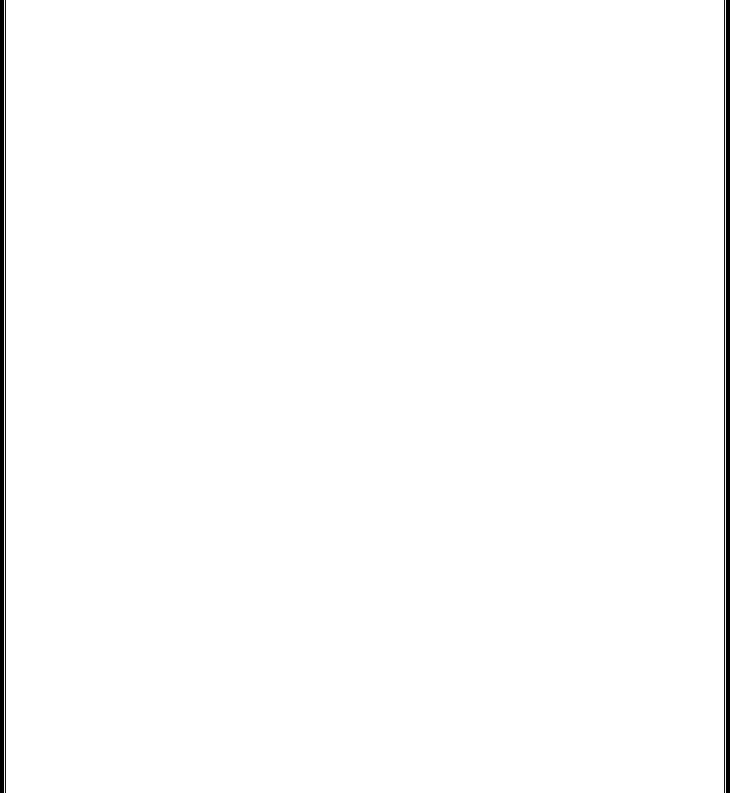
}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*

static boolean isalpha(char c)

{



if (c >= 'a' && c <= 'z' || c >= 'A'

&& c <= 'Z')

{

return true;

}

return false;

}

static boolean isdigit(char c)

{

if (c >= '0' && c <= '9')

{

return true;

}

return false;

}

static boolean isOperator(char c)

{

return (!isalpha(c) &&

!

isdigit(c));

}

static int getPriority(char C)

{

if (C == '-' || C == '+')

return 1;

else if (C == '\*' || C == '/')

return 2;

else if (C == '^')

return 3;

return 0;

}

// Reverse the letters of the word

static String reverse(char str[], int

start, int end)

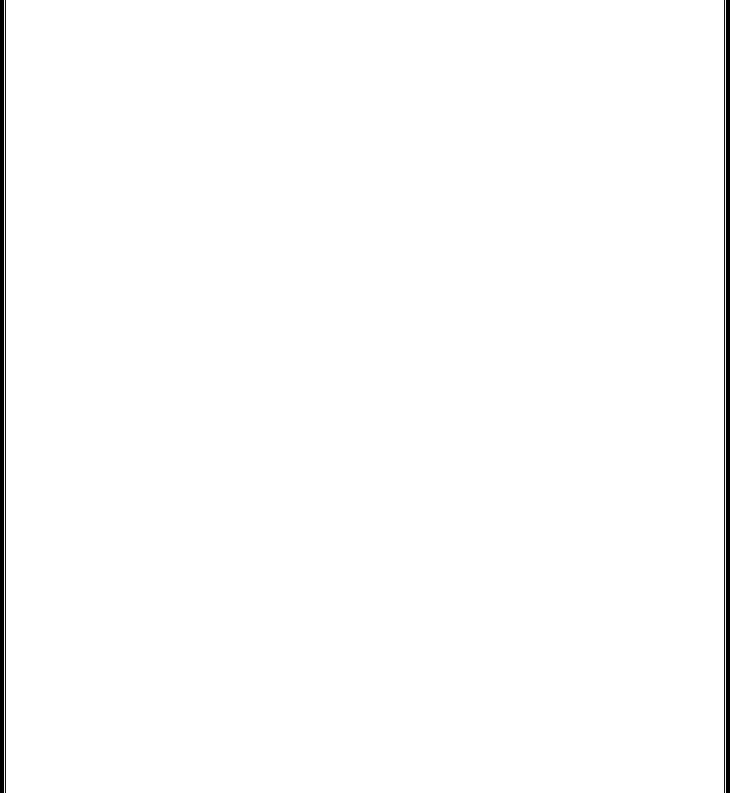
{

// Temporary variable to store

character

char temp;

while (start < end)



{

// Swapping the first and last

character

temp = str[start];

str[start] = str[end];

str[end] = temp;

start++;

end--;

}

return String.valueOf(str);

}

static String infixToPostfix(char[]

infix1)

{

System.out.println(infix1);

String infix = '(' +

String.valueOf(infix1) + ')';

int l = infix.length();

Stack<Character> char\_stack =

new Stack<>();

String output="";

for (int i = 0; i < l; i++)

{

// If the scanned character is an

// operand, add it to output.

if (isalpha(infix.charAt(i)) ||

isdigit(infix.charAt(i)))

output += infix.charAt(i);

// If the scanned character is an

// ‘(‘, push it to the stack.

else if (infix.charAt(i) == '(')

char\_stack.add('(');

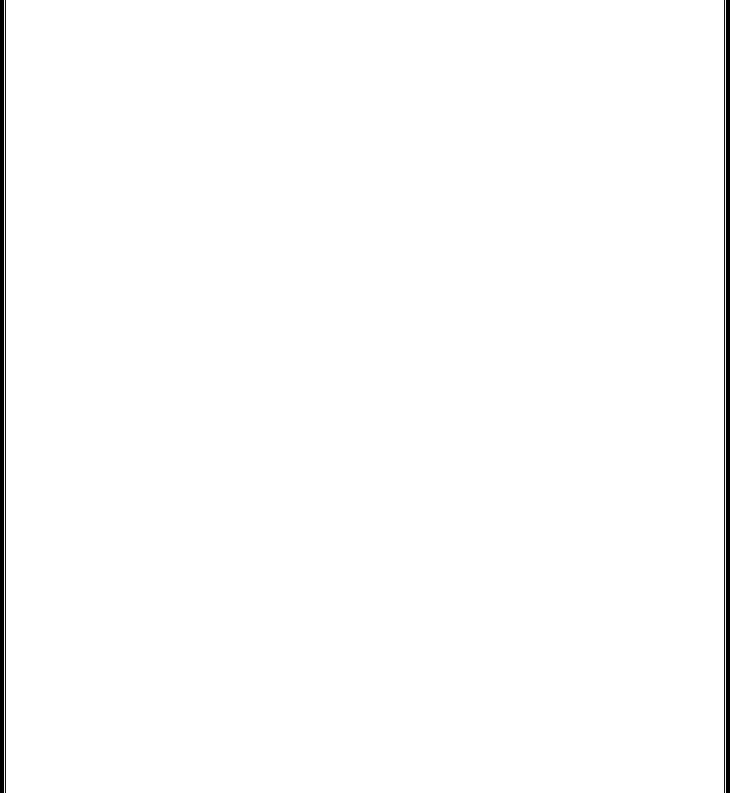
// If the scanned character is an

// ‘)’, pop and output from the

stack

// until an ‘(‘ is encountered.

else if (infix.charAt(i) == ')')



{

while (char\_stack.peek() != '(')

{

output += char\_stack.peek();

char\_stack.pop();

}

// Remove '(' from the stack

char\_stack.pop();

}

// Operator found

else {

if

()))

(

isOperator(char\_stack.peek

{

while

((

getPriority(infix.charAt(i))

<

getPriority(char\_stack.peek()))

||

(

getPriority(infix.charAt(i))

<=

getPriority(char\_stack.peek())

&& infix.charAt(i) ==

'^'))

{

output += char\_stack.peek();

char\_stack.pop();

}

// Push current Operator on

stack

char\_stack.add(infix.charAt(i));

}

}

}

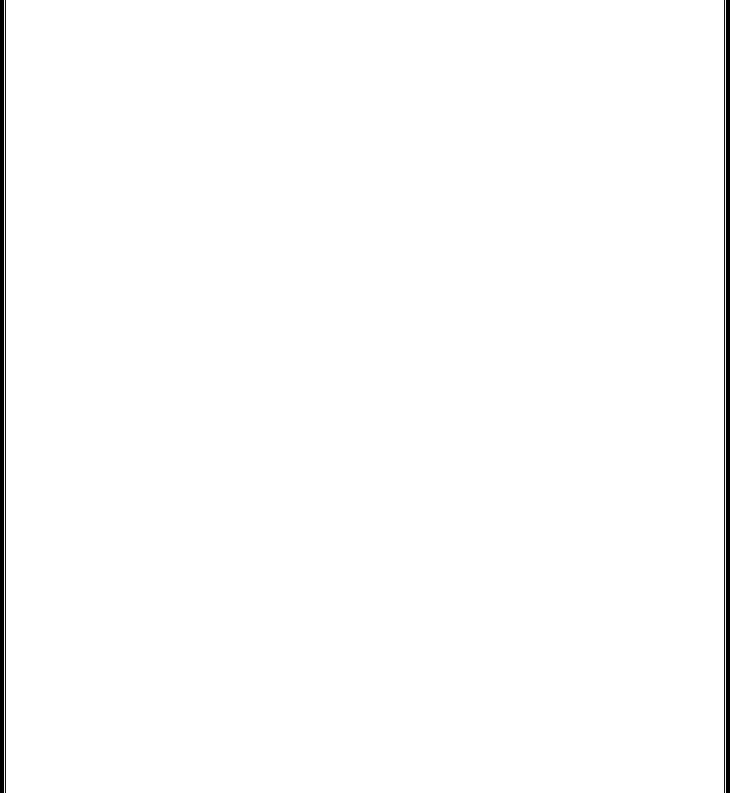
while(!char\_stack.empty()){

output += char\_stack.pop();

}

return output;

}



static String infixToPrefix(char[]

infix)

{

/\*

\* Reverse String Replace (

with ) and vice versa Get Postfix

Reverse Postfix \*

\*/

int l = infix.length;

// Reverse infix

String infix1 = reverse(infix, 0, l -

1)

;

infix = infix1.toCharArray();

// Replace ( with ) and vice versa

for (int i = 0; i < l; i++)

{

if (infix[i] == '(')

{

infix[i] = ')';

i++;

}

else if (infix[i] == ')')

{

infix[i] = '(';

i++;

}

}

String prefix =

infixToPostfix(infix);

// Reverse postfix

prefix =

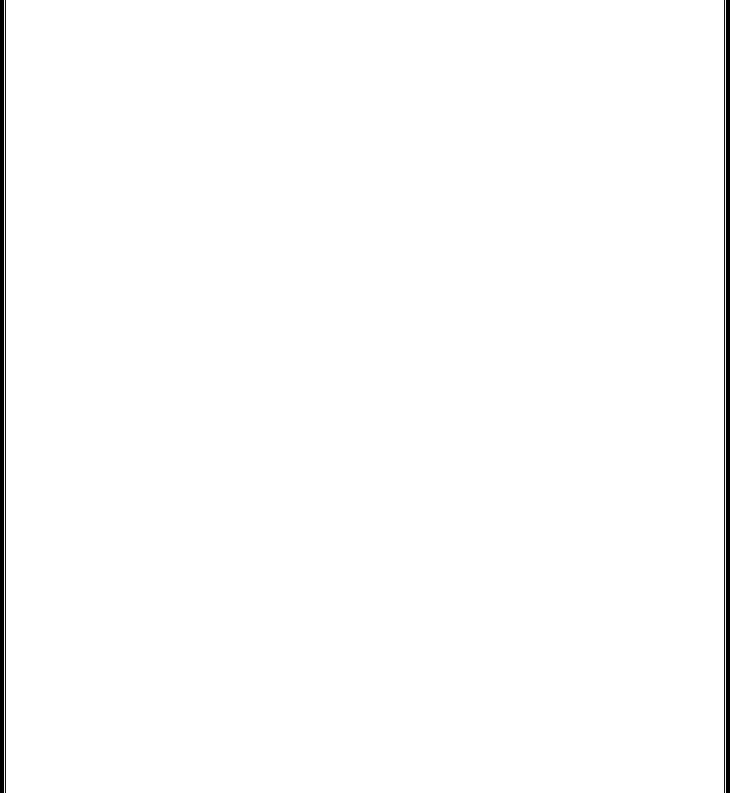
reverse(prefix.toCharArray(), 0, l-1);

return prefix;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*



static boolean isOperand(char x)

{

return (x >= 'a' && x <= 'z') ||

(

x >= 'A' && x <= 'Z');

}

// POSTFIX to INFIX

static String postToIn(String exp)

{

Stack<String> s = new

Stack<String>();

for (int i = 0; i < exp.length();

i++)

{

// Push operands

if (isOperand(exp.charAt(i)))

{

s.push(exp.charAt(i) + "");

}

// We assume that input is

// a valid postfix and expect

// an operator.

else

{

String op1 = s.peek();

s.pop();

String op2 = s.peek();

s.pop();

s.push("(" + op2 +

exp.charAt(i) +

op1 + ")");

}

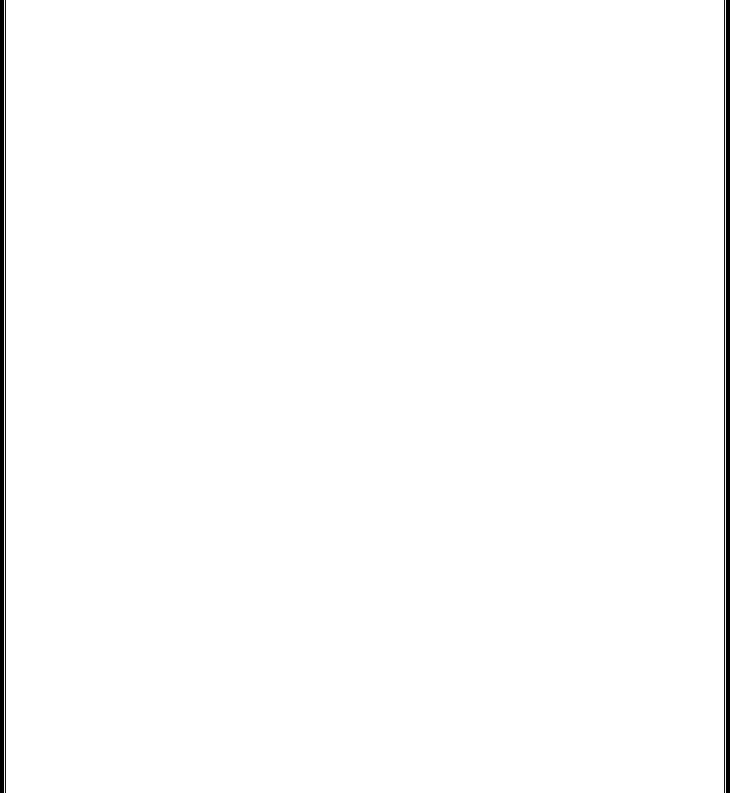
}

// There must be a single element

// in stack now which is the

required

// infix.



return s.peek();

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

static int Prec(char ch)

{

switch (ch) {

case '+':

case '-':

return 1;

case '\*':

case '/':

return 2;

case '^':

return 3;

}

return -1;

}

// The main method that converts

// given infix expression

// to postfix expression.

static String infixToPostfix(String

exp)

{

// initializing empty String for

result

String result = new String("");

// initializing empty stack

Deque<Character> stack

=

new

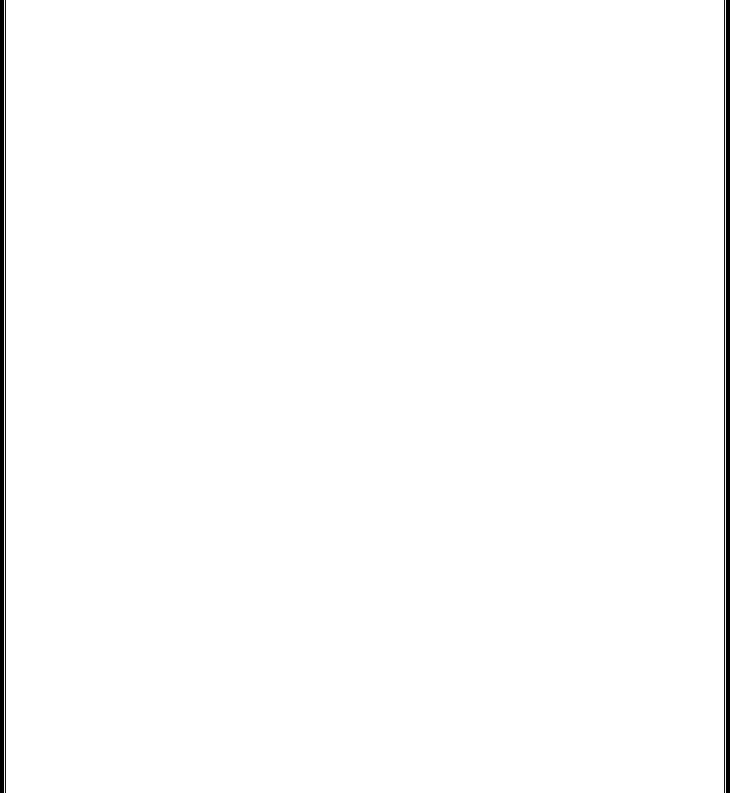
ArrayDeque<Character>();

for (int i = 0; i < exp.length();

++

i)

{



char c = exp.charAt(i);

// If the scanned character is

an

// operand, add it to output.

if

(

Character.isLetterOrDigit(c

))

result += c;

// If the scanned character is

an '(',

// push it to the stack.

else if (c == '(')

stack.push(c);

// If the scanned character is

an ')',

// pop and output from the

stack

// until an '(' is encountered.

else if (c == ')') {

while (!stack.isEmpty()

&& stack.peek() !=

'(') {

result += stack.peek();

stack.pop();

}

stack.pop();

}

else // an operator is

encountered

{

while (!stack.isEmpty()

&& Prec(c) <=

Prec(stack.peek())) {

result += stack.peek();

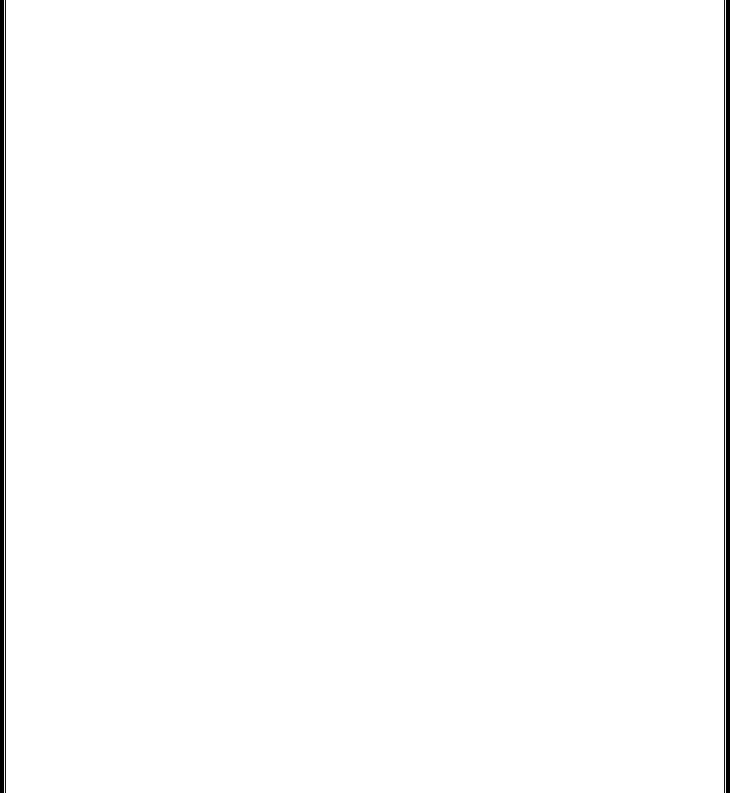
stack.pop();

}

stack.push(c);

}

}



// pop all the operators from the

stack

while (!stack.isEmpty()) {

if (stack.peek() == '(')

return "Invalid

Expression";

result += stack.peek();

stack.pop();

}

return result;

}

public static void main(String

args[]) throws IOException

{

// Step 1: Establish the socket

connection.

ServerSocket ss = new

ServerSocket(4444);

Socket s = ss.accept();

// Step 2: Processing the

request.

DataInputStream dis = new

DataInputStream(s.getInputStream()

)

;

DataOutputStream dos = new

DataOutputStream(s.getOutputStrea

m());

while (true)

{

// wait for input

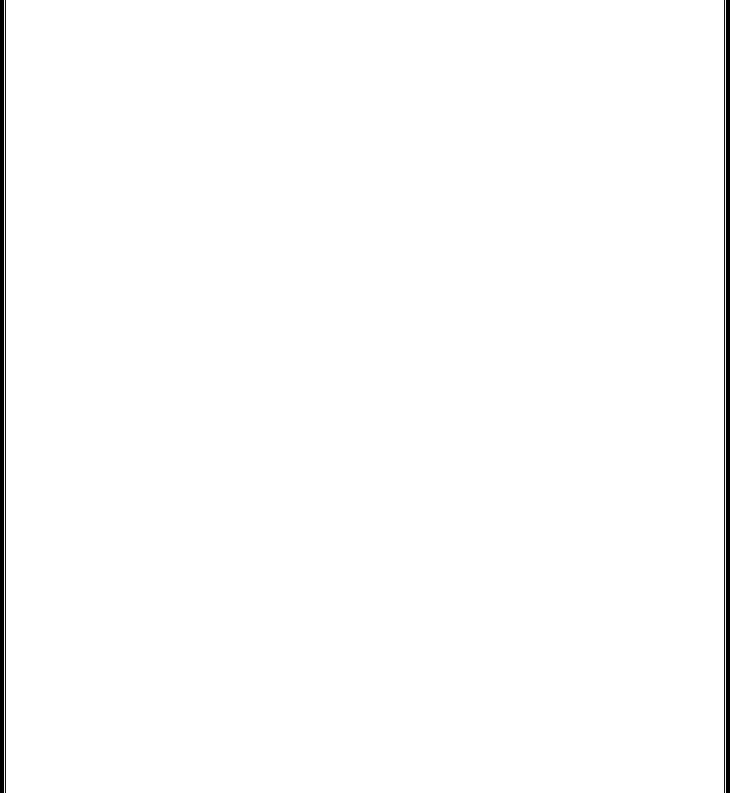
String input = dis.readUTF();

if(input.equals("bye"))

break;

System.out.println("Input

received : " + input);



// Use StringTokenizer to

break the equation into form and

// expression

StringTokenizer st = new

StringTokenizer(input);

String form1 =

st.nextToken();

String exp = st.nextToken();

String form2 =

st.nextToken();

String Result;

// perform the required

operation.

if (form1.equals("PREFIX")

&& form2.equals("POSTFIX"))

{

Result = "Postfix : "+

preToPost(exp);

}

else if

(

form1.equals("POSTFIX") &&

form2.equals("PREFIX"))

{

Result = "Prefix : "+

postToPre(exp);

}

else if

(

form1.equals("PREFIX") &&

form2.equals("INFIX"))

{

Result = "Infix : " +

preToIn(exp);

}

else if

(

form1.equals("INFIX") &&

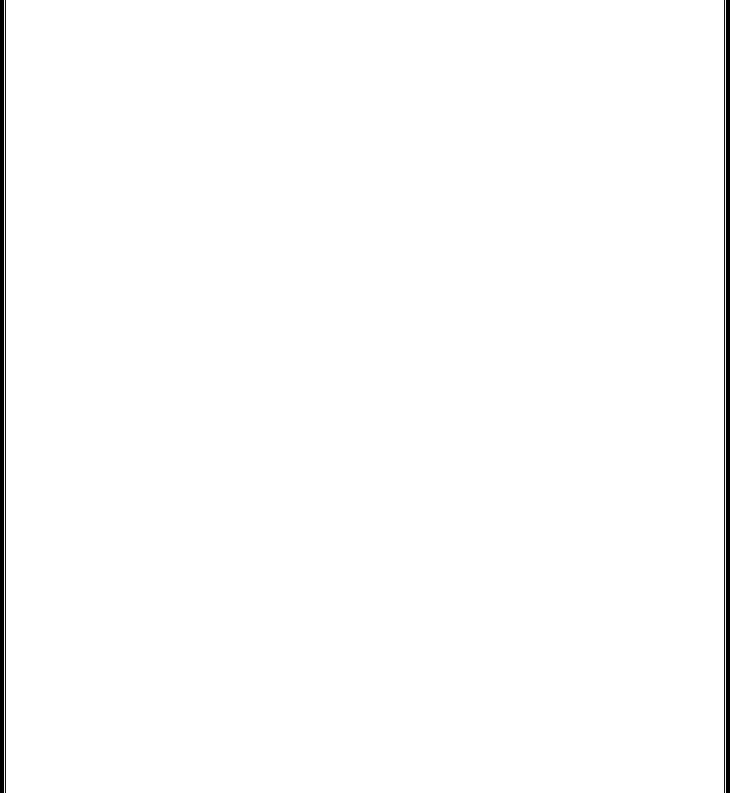
form2.equals("PREFIX"))

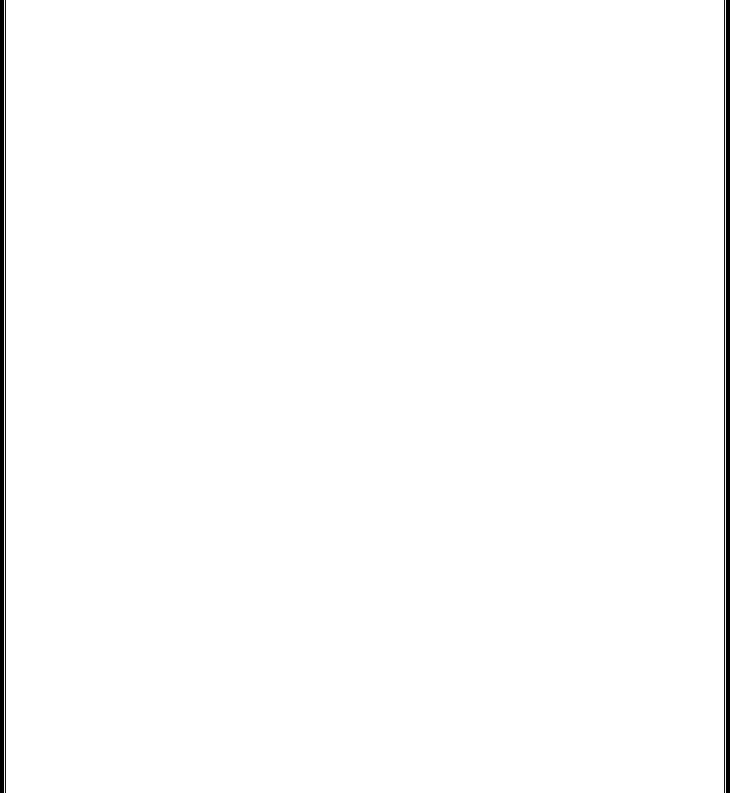
{

Result = "Prefix : " +

infixToPrefix(exp.toCharArray());

|  |
| --- |
| }  else if  (form1.equals("POSTFIX") && form2.equals("INFIX"))  {  Result = "Infix : "+  postToIn(exp); }  else  {  Result = "Postfix : "+  infixToPostfix(exp);  }  System.out.println("Sending  the result...");  // send the result back to the  client. dos.writeUTF(Result);  }  }  } |





**6.2**

**. Client side implementation**

// Java program to illustrate Client Side Programming

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.IOException;

import java.net.InetAddress;

import java.net.Socket;

import java.net.UnknownHostException;

import java.util.Scanner;

class Client\_CNProj

{

public static void main(String[] args) throws IOException

{

InetAddress ip = InetAddress.getLocalHost();

int port = 4444;

Scanner sc = new Scanner(System.in);

// Step 1: Open the socket connection.

Socket s = new Socket(ip, port);

// Step 2: Communication-get the input and output

stream

DataInputStream dis = new

DataInputStream(s.getInputStream());

DataOutputStream dos = new

DataOutputStream(s.getOutputStream());

while (true)

{

// Enter the equation in the form-

// "i.e.INFIX 3+4 POSTFIX"

System.out.print("Enter the form, expression,

converted form : ");

String inp = sc.nextLine();

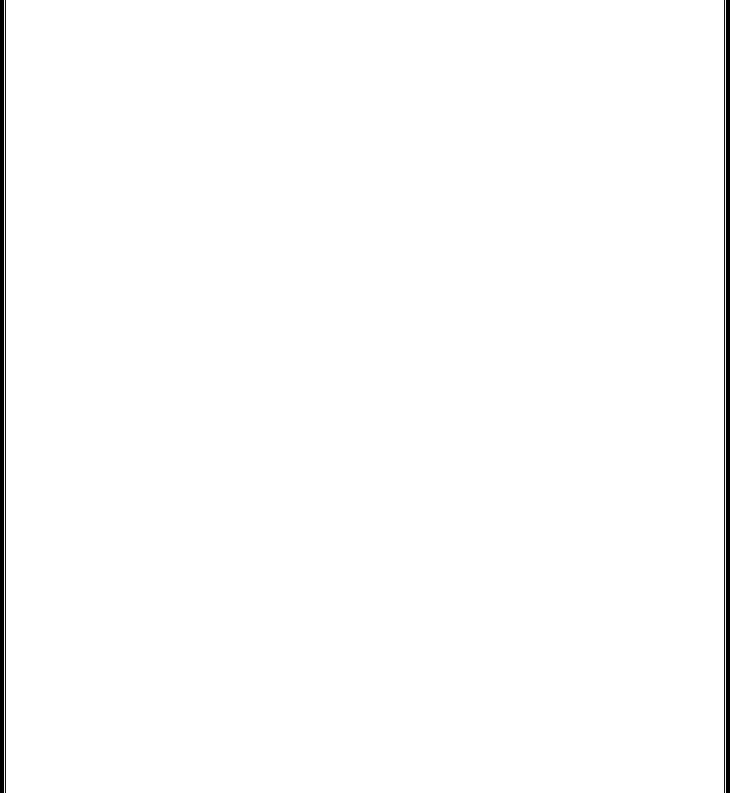
if (inp.equals("bye"))

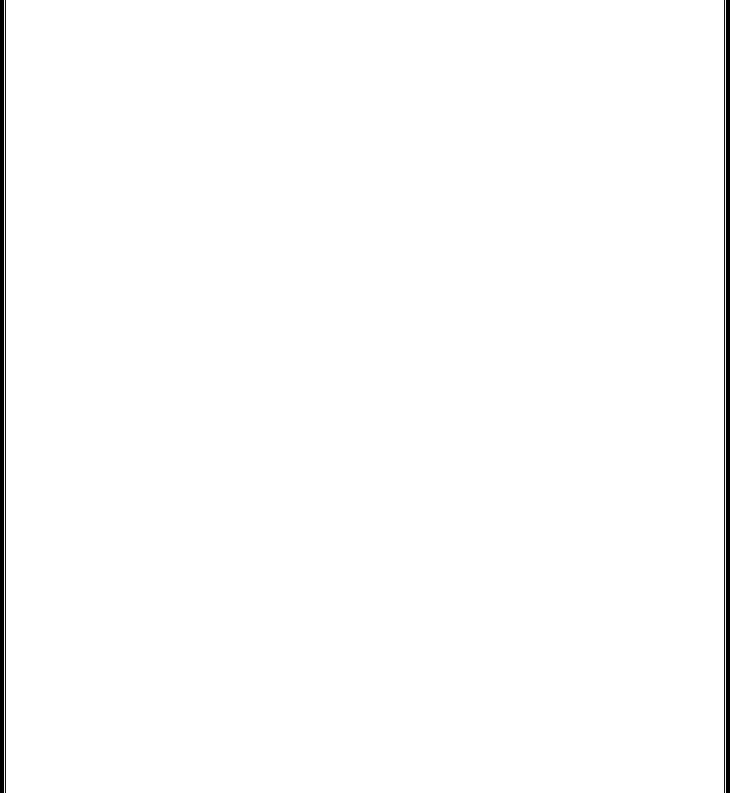
break;

// send the equation to server

dos.writeUTF(inp);

|  |
| --- |
| // wait till request is processed and sent back to  client  String ans = dis.readUTF();  System.out.println(ans);  }  }  } |



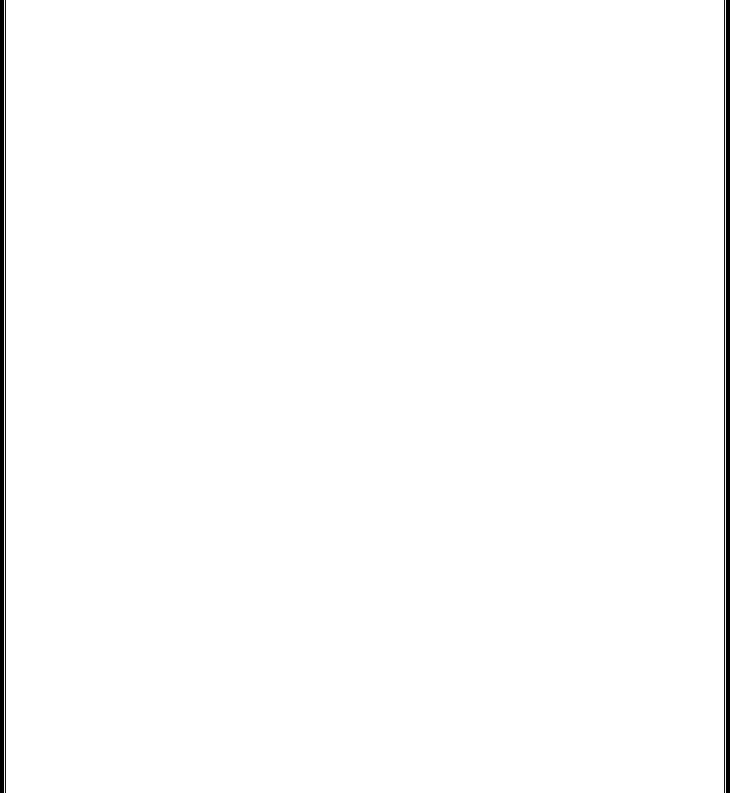
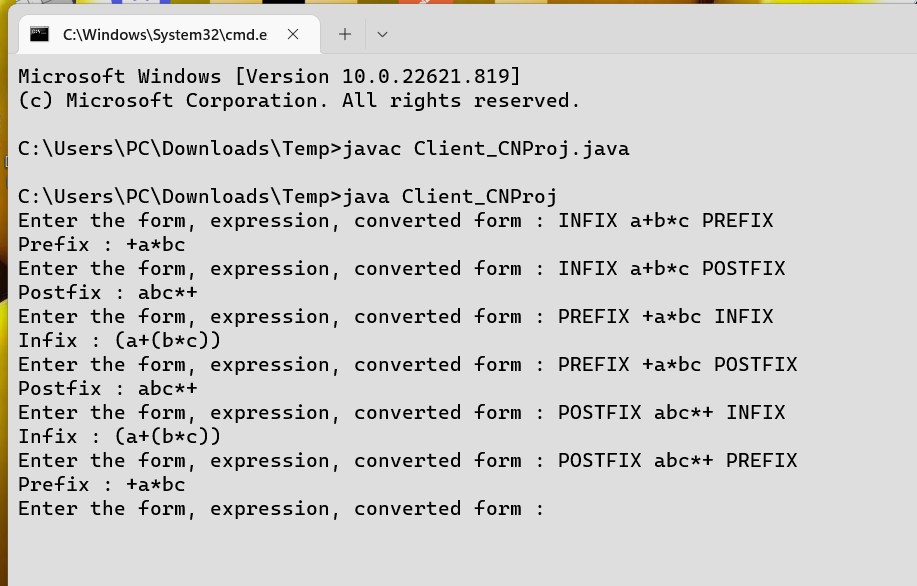
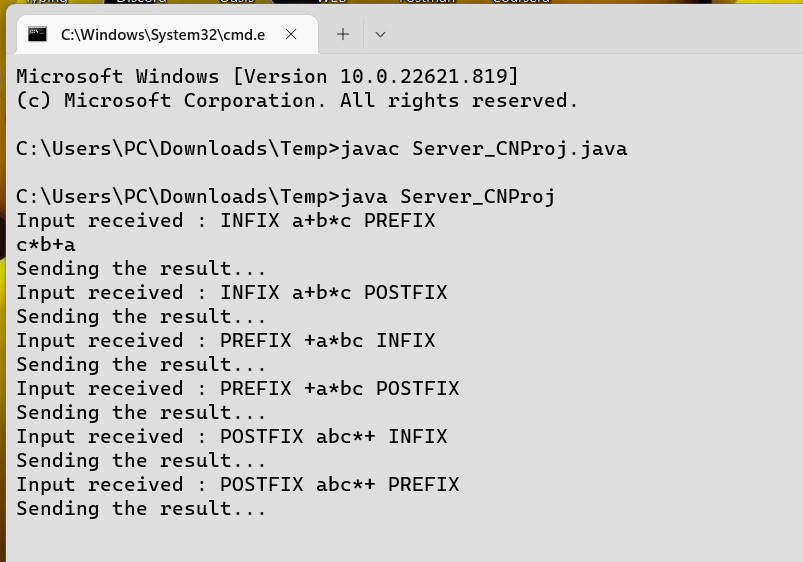


**7**

**. OUTPUT**

**Server side -**

**Client Side-**



**8.**

**LEARNING OUTCOME**

➢

Implementation of Sockets using Transmission Control

Protocol(TCP).

➢

Client Server Acrchitecture using Transmission Control

Protocol(TCP).

➢

Various functions for the connection establishment.

➢

Connection establishment between client and Server.

➢

Conversion of given expression from-

●

Prefix to Postfix conversion

●

Postfix to Prefix conversion

●

Prefix to Infix conversion

●

Infix to Prefix conversion

●

Postfix to Infix conversion

●

Infix to Postfix conversion