

## Experiment No : 01

**Experiment Name :** Write a JAVA Program to Display Image using JFrame

**Theory :** Java provides a comprehensive set of libraries for building Graphical User Interfaces (GUIs), and one of the key components in GUI development is the JFrame class. This lab explores the creation of a simple Java program to display an image using the Swing library and JFrame. The primary objectives include understanding the basic principles of GUI programming and gaining familiarity with key Swing components.

The JFrame class is a fundamental component of Swing, representing the main window of a GUI application. It provides functionalities such as title setting, size configuration, and default close operations. Through JFrame, developers can create a window to host various GUI components.

### Program Structure:

- ❖ **ImageDisplay Class:** The ImageDisplay class is defined, extending the JFrame class. In the constructor, essential properties of the JFrame, such as title, size, and default close operation, are configured. Additionally, an instance of ImageIcon is created to represent the image, and a JLabel is used to display the image within the JFrame.
- ❖ **Main Method:** The main method serves as the entry point for the program. It ensures that the GUI-related tasks are executed on the Event Dispatch Thread (EDT) using SwingUtilities.invokeLater. An instance of the ImageDisplay class is created, and the JFrame is set to be visible.

### Source Code:

```
import java.awt.FlowLayout;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
public class JFrame extends JFrame {
    private ImageIcon image1;
    private JLabel label1;
    private ImageIcon image2;
    private JLabel label2;
    JFrame(){
        setLayout(new FlowLayout());
        image1 = new ImageIcon(getClass().getResource("jdk.jpeg"));
```

```
label1 = new JLabel(image1);
add(label1);
image2 = new ImageIcon(getClass().getResource("python.png"));
label2 = new JLabel(image2);
add(label2);
}
public static void main(String args[]) {
Jframe gui = new Jframe();
gui.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
gui.setVisible(true);
gui.pack();
gui.setTitle("Image Program");}
```

**Output:**



## Experiment No : 02

**Experiment Name :** Write a JAVA Program for generating Restaurant Bill

**Theory :** The objective of this lab is to develop a Java program that generates a restaurant bill based on items ordered by the customer. The program utilizes fundamental concepts of Java programming, including data structures, input/output handling, and basic arithmetic operations. By completing this lab, students gain practical experience in software development and learn how to implement a simple billing system.

### Program Structure:

- ❖ **Menu Initialization:** The program initializes a menu containing various food items and their corresponding prices. This information is stored in a Java Map data structure, with the item names as keys and the prices as values.

- ❖ **Order Placement:** Using the menu, the program prompts the user to place an order by selecting items from the menu and specifying the quantities. The user input is processed to record the ordered items and their respective quantities, which are stored in another Map data structure.
- ❖ **Bill Calculation:** Based on the recorded order, the program calculates the total bill by multiplying the price of each item by its quantity and summing up the results.
- ❖ **Displaying the Bill:** Finally, the program displays the generated bill, listing the ordered items, their quantities, individual prices, and the total bill amount.

#### Source Code :

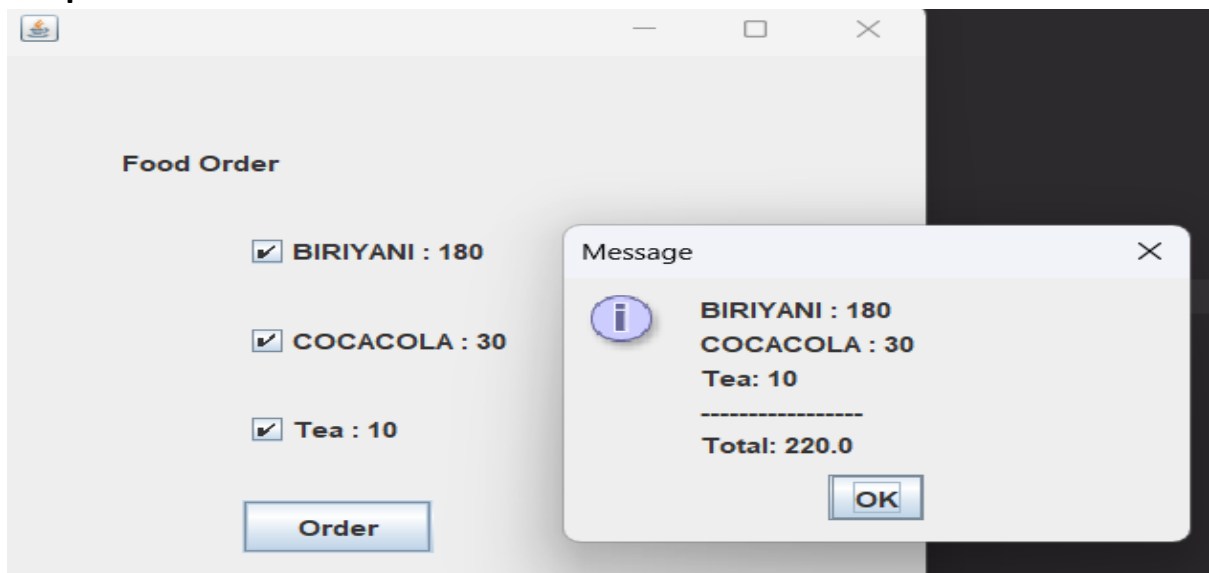
```
import javax.swing.*;
import java.awt.event.*;
public class billrest extends JFrame implements ActionListener{
    JLabel l;
    JCheckBox cb1,cb2,cb3;
    JButton b;
    billrest(){
        l=new JLabel("Food Order");
        l.setBounds(50,50,300,20);
        cb1=new JCheckBox("BIRIYANI : 180");
        cb1.setBounds(100,100,150,20);
        cb2=new JCheckBox("COCACOLA : 30");
        cb2.setBounds(100,150,150,20);
        cb3=new JCheckBox("Tea : 10");
        cb3.setBounds(100,200,150,20);
        b=new JButton("Order");
        b.setBounds(100,250,80,30);
        b.addActionListener(this);
        add(l);add(cb1);add(cb2);add(cb3);add(b);
        setSize(400,400);
        setLayout(null);
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
    public void actionPerformed(ActionEvent e){
        float amount=0;
```

```

String msg="";
if(cb1.isSelected()){
amount+=180;
msg="BIRIYANI : 180\n";
}
if(cb2.isSelected()){
amount+=30;
msg+="COCACOLA : 30\n";
}
if(cb3.isSelected()){
amount+=10;
msg+="Tea: 10\n";
}
msg+="-----\n";
JOptionPane.showMessageDialog(this,msg+"Total: "+amount);
}
public static void main(String[] args) {
new billrest();
}
}

```

### Output :



## Experiment No : 03

**Experiment Name :** Write a JAVA Program to Create a Student form in GUI

### Theory :

The goal of this lab is to design and implement a graphical user interface (GUI) for a student information form using Java's Swing library. The lab aims to familiarize students with GUI programming concepts, event handling, and the creation of interactive forms in Java.

Graphical User Interfaces (GUIs) are essential components in modern software applications, providing users with an interactive and user-friendly experience. Java's Swing library offers a rich set of tools for creating GUIs. This lab focuses on the creation of a student form, which typically includes input fields for personal information.

### Program Structure :

- ❖ **JFrame and Layout:** The program utilizes the JFrame class as the main window for the GUI. A GridLayout is used to organize and arrange the components in a grid structure.
- ❖ **Form Components:** The form components include JTextField for inputting text, JRadioButton for selecting gender, JComboBox for selecting the grade, and a JButton for submitting the form.
- ❖ **Action Handling:** An ActionListener is implemented to handle the action when the submit button is clicked. The entered information is then processed, and a dialog is displayed to present the submitted data.
- ❖ **Implementation:** The Java code provided in the previous response serves as the implementation of the student form. It utilizes Swing components to create an interactive and user-friendly form.

### Source Code :

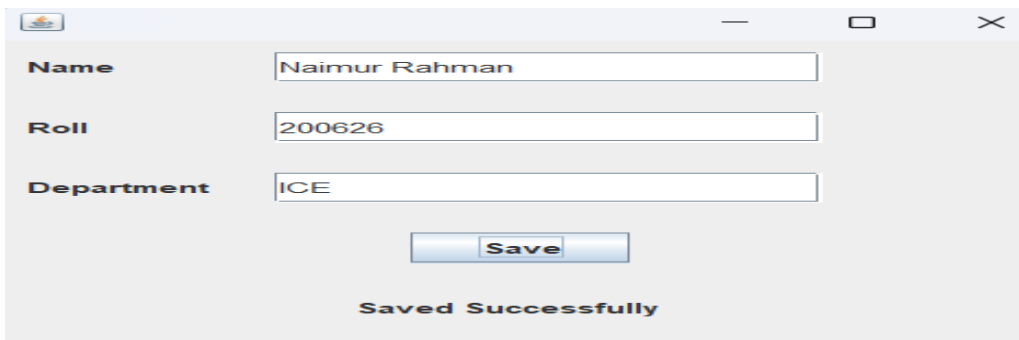
```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
public class studentinfo implements ActionListener {
```

```

private static JLabel success;
private static JFrame frame;
private static JLabel label1,label2,label3;
private static JTextField userText1, userText2, userText3;
public static void main(String[] args) {
    frame = new JFrame();
    panel = new JPanel();
    frame.setSize(400, 300);
    label1= new JLabel("Name");
    label1.setBounds(10,10,80,25);
    userText1 = new JTextField("Enter Your Name");
    userText1.setBounds(100,10,200,25);
    panel.add(userText1);
    JTextField userText2 = new JTextField("Enter Your Name");
    userText2.setBounds(100,60,200,25);
    panel.add(userText2);
    JTextField userText3 = new JTextField("Enter Your Name");
    button.setBounds(150, 160, 80, 25);
    button.addActionListener(new studentinfo());
    panel.add(button);
    success = new JLabel("");
    success.setBounds(130,210,300,25);
    panel.add(success);
    frame.setVisible(true);
}
@Override
public void actionPerformed(ActionEvent e) {
    success.setText("Saved Successfully");
}
}

```

## Output :



The screenshot shows a Java Swing window with a light gray background. It contains three text input fields arranged vertically. The first field is labeled 'Name' and contains the text 'Naimur Rahman'. The second field is labeled 'Roll' and contains the text '200626'. The third field is labeled 'Department' and contains the text 'ICE'. Below these fields is a blue button with the text 'Save'. At the bottom of the window, the text 'Saved Successfully' is displayed in a bold, black font.

## Experiment No : 04

**Experiment Name :** Write a JAVA Program to simple calculator in GUI.

**Theory :** The goal of this lab is to design and implement a graphical user interface (GUI) for a student information form using Java's Swing library. The lab aims to familiarize students with GUI programming concepts, event handling, and the creation of interactive forms in Java. Graphical User Interfaces (GUIs) are essential components in modern software applications, providing users with an interactive and user-friendly experience. Java's Swing library offers a rich set of tools for creating GUIs. This lab focuses on the creation of a student form, which typically includes input fields for personal information.

### Program Structure

- ❖ **JFrame and Layout:** The program utilizes the JFrame class as the main window for the GUI. A GridLayout is used to organize and arrange the components in a grid structure.
- ❖ **Form Components:** The form components include JTextField for inputting text, JRadioButton for selecting gender, JComboBox for selecting the grade, and a JButton for submitting the form.
- ❖ **Action Handling:** An ActionListener is implemented to handle the action when the submit button is clicked. The entered information is then processed, and a dialog is displayed to present the submitted data.
- ❖ **Implementation:** The Java code provided in the previous response serves as the implementation of the student form. It utilizes Swing components to create an interactive and user-friendly form.

Source Code :

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class test extends JFrame implements ActionListener {
    protected Container c;
    protected Font f2 = new Font("Times New Roman", Font.ROMAN_BASELINE, 38);
    protected Font f1 = new Font("Times New Roman", Font.BOLD, 48);
    protected JButton btn[]=new JButton[18];
    protected JPanel pnl1, pnl2;
    protected JTextArea res;
    protected static int i,r,n1,n2;
    protected static String text="";
```

```

char op;
boolean opf=false;
test() {
    this.setBounds(20, 29, 526, 635);
    this.setTitle("Calculator");
    Rectangle rctngl = new Rectangle(280, 50, 850, 635);
    this.setMaximizedBounds(rctngl);
    this.setLocationRelativeTo(null);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    pnl1 = new JPanel();
    pnl1.setLayout(new GridLayout());
    res = new JTextArea("");
    res.setFont(f1);
    pnl1.add(res);
    pnl2 = new JPanel();
    pnl2.setBounds(5, 110, 500, 480);
    pnl2.setLayout(new GridLayout(4, 4));
    for(int i=0;i<=9;i++)
    {
        btn[i]=new JButton(i+"");
    }
    for(int i=7;i<=9;i++) {
        pnl2.add(btn[i]);
    }
    btn[13]= new JButton("/");
    pnl2.add(btn[13]); // division
    for(int i=4;i<=6;i++) {
        pnl2.add(btn[i]);
    }
    btn[12] = new JButton("x");
    pnl2.add(btn[12]);
    for(int i=1;i<=3;i++) {
        pnl2.add(btn[i]);
    }
    btn[11] = new JButton("-");
    pnl2.add(btn[11]);
    btn[15] = new JButton("AC");
    btn[15].setForeground(Color.RED);
    pnl2.add(btn[15]); // AC
    pnl2.add(btn[0]); // zero
    btn[10] = new JButton("+");
    pnl2.add( btn[10]); // Addition
    btn[14] = new JButton("=");
    pnl2.add(btn[14]); // Result
    for(int i=0;i<=15;i++) {
        btn[i].setFont(f2);
        btn[i].setBackground(Color.MAGENTA); }
}

```



```

public void actionPerformed(ActionEvent e) {
    JButton pb=(JButton)e.getSource();
    if(pb==btn[15]){
        r=n1=n2=0;
        text="";
        opf=false;
        res.setText("");
    }
    else if(pb==btn[14]){
        n2 = Integer.parseInt(text);
        System.out.println(n2);
//        n2=3;
        eval();
        res.append(" = "+r);
        text="";
    }
    else{for(int j=10;j<=13;j++){
        if(pb==btn[j]){
            n1 = Integer.parseInt(res.getText());
            if(pb==btn[10]){ op='+'; res.append(" + "); opf=true;}
            if(pb==btn[11]){ op='-'; res.append(" - "); opf=true;}
            if(pb==btn[12]){ op='*'; res.append(" x "); opf=true;}
            if(pb==btn[13]){ op='/'; res.append(" / "); opf=true;}
        }

        if(opf==false){
            for(i=0;i<=9;i++){
                if(pb==btn[i]){
                    String t=res.getText();
                    t+=i;
                    res.setText(t);
                }
            }
        }
        else{
            for(i=0;i<=9;i++){
                if(pb==btn[i]){
                    text +=i;
                    res.append(Integer.toString(i));
                }
            }
        }
    }
}

int eval(){
    switch(op)
    {

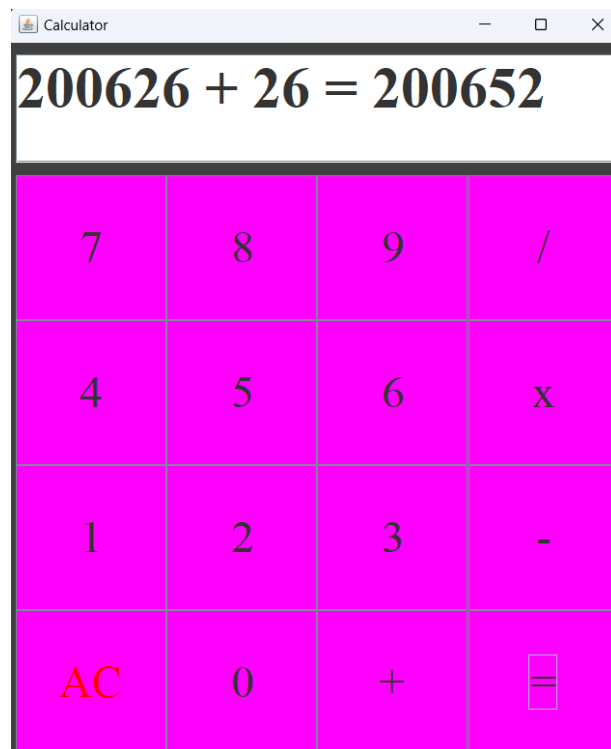
```

```

        case '+': r=n1+n2; break;
        case '-': r=n1-n2; break;
        case '*': r=n1*n2; break;
        case '/': r=n1/n2; break;
    }
    public static void main(String[] args) {
        test frm = new test();
        frm.setVisible(true);
    }
}

```

Output :



## Experiment No : 05

**Experiment Title:** Write a Java program to illustrate suspend, resume and stop operation of thread.

**Theory:** In Java, the **Thread** class provides methods to manage the lifecycle of a thread, including suspending, resuming, and stopping a thread. However, these methods (**suspend()**, **resume()**, and **stop()**) are deprecated due to their potential to cause deadlocks and other thread-related issues. It's recommended to use alternative mechanisms like **wait()**, **notify()**, and **interrupt()** for more controlled thread management.

**Program Structure:**

### Program Structure:

1. Create a class named **MyThread** implementing the **Runnable** interface.
2. Inside **MyThread**, define variables for thread object, suspend flag, and stop flag.
3. Implement the **run()** method where the thread performs its task.
4. Implement methods **suspendThread()**, **resumeThread()**, and **stopThread()** to control the thread's execution.
5. Create a class named **ThreadControlExperiment** which contains the **main()** method.
6. Inside **main()**, create an instance of **MyThread** and demonstrate the suspend, resume, and stop operations.

### Source Code:

```
class MyThread implements Runnable {
    private Thread thread;
    private volatile boolean suspended;
    private volatile boolean stopped;

    MyThread(String name) {
        thread = new Thread(this, name);
        suspended = false;
        stopped = false;
        thread.start();
    }
}
```

```

public void run() {
    System.out.println(Thread.currentThread().getName() + " starting.");
    try {
        for (int i = 1; i <= 10; i++) {
            System.out.print(i + " ");
            Thread.sleep(500);
            synchronized (this) {
                while (suspended) {
                    wait();
                }
                if (stopped) {
                    break;
                }
            }
        }
    } catch (InterruptedException e) {
        System.out.println(Thread.currentThread().getName() + "
    }
}

public class ThreadControlExperiment {
    public static void main(String[] args) {
        MyThread myThread = new MyThread("MyThread");

        try {
            Thread.sleep(2000); // Let the thread start
            myThread.suspendThread();
            System.out.println("Suspending MyThread...");
            Thread.sleep(2000);
            myThread.resumeThread();
            System.out.println("Resuming MyThread...");
            Thread.sleep(2000);
            myThread.suspendThread();
            System.out.println("Suspending MyThread again...");
            Thread.sleep(2000);
            myThread.resumeThread();
            System.out.println("Resuming MyThread again...");
            Thread.sleep(2000);
            myThread.stopThread();
            System.out.println("Stopping MyThread...");
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
    }
}

```

```
    }  
    try {  
        myThread.thread.join();  
    } catch (InterruptedException e) {  
        System.out.println("Main thread interrupted.");  
    }  
    System.out.println("Main thread exiting.");  
}  
}
```

#### Output:

```
Thread-0 started.  
Thread-0: 1  
Thread-0: 2  
Thread-0: 3  
Suspending the thread...  
Resuming the thread...  
Thread-0: 4  
Thread-0: 5  
Thread-0 exiting.  
Stopping the thread...
```

#### Experiment No: 06

**Experiment Title:** Write a Java program to create thread class.

#### Theory:

In Java, the **Thread** class provides methods to manage the lifecycle of a thread, including suspending, resuming, and stopping a thread. However, these methods (**suspend()**, **resume()**, and **stop()**) are deprecated due to their potential to cause deadlocks and other thread-related issues. It's recommended to use alternative mechanisms like **wait()**, **notify()**, and **interrupt()** for more controlled thread management.

**Program Structure:**

#### Program Structure:

1. Create a class named **MyThread** implementing the **Runnable** interface.
2. Inside **MyThread**, define variables for thread object, suspend flag, and stop flag.
3. Implement the **run()** method where the thread performs its task.

4. Implement methods **suspendThread()**, **resumeThread()**, and **stopThread()** to control the thread's execution.
5. Create a class named **ThreadControlExperiment** which contains the **main()** method.
6. Inside **main()**, create an instance of **MyThread** and demonstrate the suspend, resume, and stop operations.

#### Source Code:

```
class MyThread extends Thread {
    public void run() {
        System.out.println(Thread.currentThread().getName() + " is
running.");
        try {
            Thread.sleep((long) (Math.random() * 5000));
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName() + "
interrupted.");
        }
        System.out.println(Thread.currentThread().getName() + " has
finished.");
    }
}

public class BasicThreadCreationExperiment {
    public static void main(String[] args) {

        MyThread thread1 = new MyThread();
        MyThread thread2 = new MyThread();
        thread1.start();
        thread2.start();
    }
}
```

### Output:

```
Thread-0: 1
Thread-0: 2
Thread-0: 3
Thread-1: 1
Thread-1: 2
Thread-0: 4
Thread-0: 5
Thread-1: 3
Thread-1: 4
Thread-1: 5
```

### Experiment No: 07

**Experiment Title:** Write a Java program to illustrate `yield()`, `stop()` and `sleep()` method using thread.

### Theory:

In Java, the **`yield()`**, **`stop()`**, and **`sleep()`** methods are used for thread control.

1. **`yield()`**: The **`yield()`** method causes the currently executing thread to temporarily pause and allow other threads of the same priority to execute. However, there is no guarantee that the current thread will yield the processor. It depends on the JVM's implementation.
2. **`stop()`**: The **`stop()`** method is used to abruptly terminate the execution of a thread. However, this method is deprecated due to its unsafe nature. It may leave the object that the thread was working on in an inconsistent state, leading to potential problems.
3. **`sleep()`**: The **`sleep()`** method pauses the execution of the current thread for a specified amount of time. It allows other threads to execute while the current thread is sleeping. It throws **`InterruptedException`** if another thread interrupts the sleeping thread.

### Program Structure:

1. Create a custom thread class that extends the **`Thread`** class.
2. Override the **`run()`** method to define the thread's behavior, including the use of **`yield()`**, **`stop()`**, and **`sleep()`** methods.

3. Create an instance of the custom thread class and start it.
4. Demonstrate the behavior of the thread by observing the effects of **yield()**, **stop()**, and **sleep()** methods.

**Source Code:**

```
class A extends Thread {
    public void run() {
        for (int i = 1; i <= 10; i++) {
            if (i == 1) {
                Thread.yield();
            }
            System.out.println("\t From thread A: i = " + i);
        }
        System.out.println("Exit from Thread A.");
    }
}

class B extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("\t From thread B: i = " + i);
        }
        System.out.println("Exit from Thread B.");
    }
}

class C extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("\t From thread C: i = " + i);
            if (i == 1) {
                try {
                    sleep(1000); // Sleep for 1 second (1000 milliseconds)
                } catch (InterruptedException e) {
                    System.out.println(e);
                }
            }
        }
    }
}
```



```

        System.out.println("Exit from Thread C.");
    }
}

public class Thread_Methods {
    public static void main(String[] args) {
        A threadA = new A();
        B threadB = new B();
        C threadC = new C();

        System.out.println("Thread A Started :");
        threadA.start();

        System.out.println("Thread B Started :");
        threadB.start();

        System.out.println("Thread C Started :");
        threadC.start();

        System.out.println("End Of Main Thread");
    }
}

```

### Output:

```

Thread A Started :
Thread B Started :
Thread C Started :
End Of Main Thread
    From thread A: i = 1
    From thread A: i = 2
    From thread B: i = 1
    From thread C: i = 1
    From thread A: i = 3
    From thread B: i = 2
    From thread A: i = 4
    From thread B: i = 3
    From thread A: i = 5
    From thread B: i = 4
    From thread A: i = 6
    From thread B: i = 5
Exit from Thread B.
    From thread A: i = 7
    From thread A: i = 8
    From thread A: i = 9
    From thread A: i = 10
Exit from Thread A.
    From thread C: i = 2
    From thread C: i = 3
    From thread C: i = 4
    From thread C: i = 5
Exit from Thread C.

```

## Experiment No: 08

**Experiment Title:** Write a Java program to use priority of thread.

### Theory:

In Java, threads are scheduled for execution by the JVM based on their priority. Thread priority is an integer value that determines the importance or urgency of a thread's execution. The thread scheduler attempts to allocate CPU time to higher priority threads before lower priority threads.

Thread priorities are represented by integer values ranging from 1 to 10, where 1 is the lowest priority and 10 is the highest. Threads with higher priority values are more likely to be scheduled for execution over threads with lower priority values. By default, all threads inherit the priority of the parent thread, which is usually the main thread. However, you can explicitly set the priority of a thread using the **setPriority()** method.

### Program Structure:

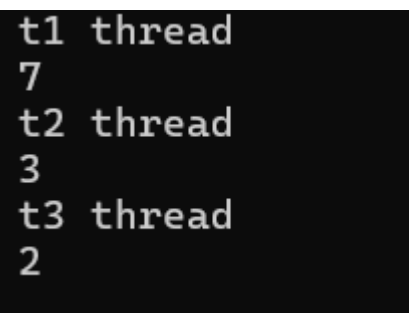
- **PriorityThread Class:**
  - Extends the **Thread** class.
  - Overrides the **run()** method to display the thread's name and priority.
- **Main Class:**
  - Contains the main method.
  - Creates instances of **PriorityThread** class with different priorities.
  - Starts the threads to demonstrate the effect of thread priority on scheduling.

### Source Code:

```
class A extends Thread {  
    public void run() {  
        System.out.println(Thread.currentThread().getName());  
        System.out.println(Thread.currentThread().getPriority());  
    }  
}
```

```
class newTheredPri {  
    public static void main(String[] args) {  
        A t1 = new A();  
        A t2 = new A();  
        A t3 = new A();  
  
        t1.setName("t1 thread");  
        t2.setName("t2 thread");  
        t3.setName("t3 thread");  
  
        t1.setPriority(7);  
        t2.setPriority(3);  
        t3.setPriority(2);  
  
        t1.start();  
        t2.start();  
        t3.start();  
    }  
}
```

**Output:**



```
t1 thread  
7  
t2 thread  
3  
t3 thread  
2
```

**Experiment No : 09**

**Experiment Title:** Write a client and server program in Java to establish a connection between them.

**Objective:**

The objective of this experiment is to understand the implementation of client-server communication in Java using sockets.

**Theory:**

In Java, client-server communication is facilitated through the use of sockets. Sockets provide the communication mechanism between two processes,

typically a client and a server, running on different machines or within the same machine.

- **Server:** The server program waits for client requests and responds to them accordingly.
- **Client:** The client program initiates communication with the server by sending requests and receiving responses.

#### **Program Structure:**

- **Server Program:**
  - Listens for incoming connections from clients.
  - Accepts client connections and handles client requests.
- **Client Program:**
  - Establishes a connection to the server.
  - Sends requests to the server and receives responses.

#### **Source Code:**

##### **Server:**

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

public class Server {
    public static void main(String[] args) {
        try {
            // Create a server socket bound to port 9999
            ServerSocket serverSocket = new ServerSocket(9999);
            System.out.println("Server started. Waiting for client connection...");

            // Accept client connection
            Socket clientSocket = serverSocket.accept();
            System.out.println("Client connected.");

            // Get input and output streams
            BufferedReader in = new BufferedReader(new
            InputStreamReader(clientSocket.getInputStream()));
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(),
            true);
```

```

        // Read client message
        String message = in.readLine();
        System.out.println("Received from client: " + message);

        // Send response to client
        out.println("Message received by server.");

        // Close streams and sockets
        in.close();
        out.close();
        clientSocket.close();
        serverSocket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

#### Client Sided Code:

```

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

public class Client {
    public static void main(String[] args) {
        try {
            // Create a socket to connect to the server running on localhost at
            port 9999
            Socket socket = new Socket("localhost", 9999);

            // Get input and output streams
            BufferedReader in = new BufferedReader(new
            InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

            // Send message to server
            out.println("Hello from client.");

            // Receive response from server
            String response = in.readLine();
            System.out.println("Response from server: " + response);
        }
    }
}

```

```
        // Close streams and socket
        in.close();
        out.close();
        socket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
```

### Output:

```
Server Started..
Client connected..
From Client: ICE
From Client: Hello
|
```

```
Client started..
Client Connected..
Enter message (type 'exit' to quit): ICE
From Server: ICE
Enter message (type 'exit' to quit): Hello
From Server: HELLO
Enter message (type 'exit' to quit):
```

