

Experiment Number: 01

Name of the experiment: To write a JAVA Program to Display Image using JFrame.

Theory: A graphical user interface, or GUI is a form of user interface that allows users to interact with electronic devices through graphical icons and visual indicators. For the graphical user interface (GUI) we use the JAVA swing and awt packages. In these packages there are many components such as JFrame, JPanel, JButton, JTextField, ActionListener, ActionListener.

FlowLayout is suitable for arranging components in a simple, row-based layout with basic alignment and spacing options. It's often used in situations where we need a simple layout without much customization.

The **javax.swing.JFrame** class is a type of container which inherits the JFrame class. The JPanel is a simplest container class. It provides space in which an application can attach any other component. It inherits the JComponents class such as JButton.

ImageIcon provides a convenient way to work with images in Swing applications, allowing us to easily load, display, and customize graphical images to enhance the user interface.

Code:

```
import javax.swing.*;
import java.awt.FlowLayout;
import java.awt.Image;

public class ImageProgram {
    public ImageProgram() {
        JFrame frame = new JFrame("Image Program");
        frame.setLayout(new FlowLayout());

        ImageIcon image1 = resizeImage(new
        ImageIcon(getClass().getResource("boston_haze.jpg")), 200, 200);
        JLabel label1 = new JLabel(image1);
        frame.add(label1);

        ImageIcon image2 = resizeImage(new
        ImageIcon(getClass().getResource("DetectingEdgesSlide.png")), 200,
        200);
        JLabel label2 = new JLabel(image2);
        frame.add(label2);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
    }
}
```

```

        frame.setVisible(true);
    }

    private ImageIcon resizeImage(ImageIcon icon, int width, int height) {
        Image img = icon.getImage();
        Image resizedImg = img.getScaledInstance(width, height,
Image.SCALE_SMOOTH);
        return new ImageIcon(resizedImg);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new ImageProgram();
            }
        });
    }
}

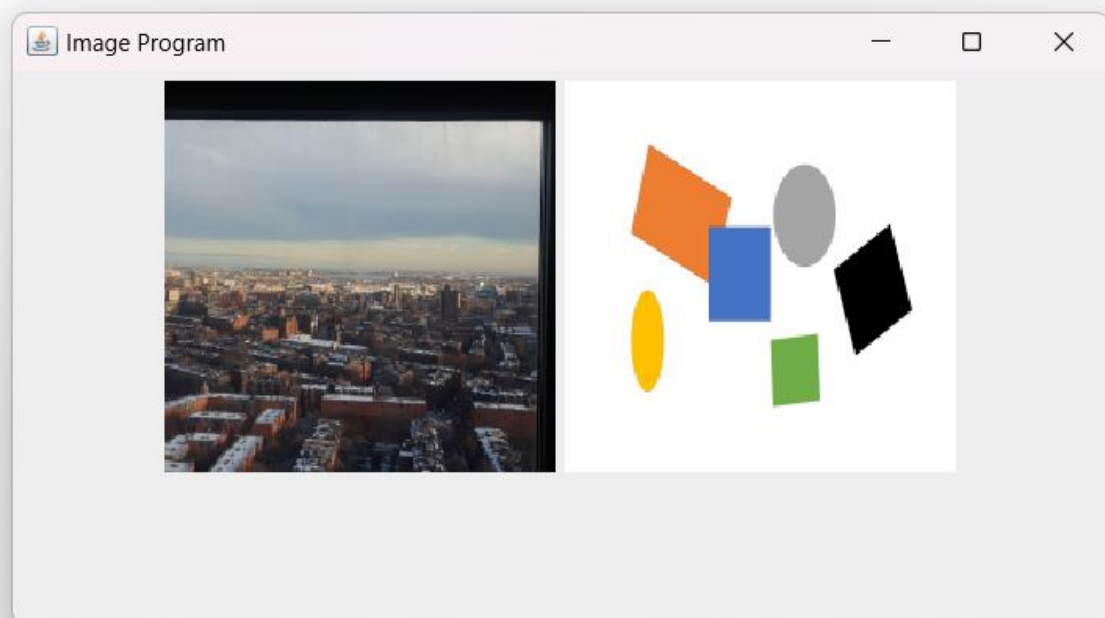
```

Input:

boston_haze.jpg

DetectingEdgesSlide.png

Output:



Experiment Number: 02

Name of the experiment: To write a JAVA Program for generating Restaurant Bill.

Theory: A graphical user interface, or GUI is a form of user interface that allows users to interact with electronic devices through graphical icons and visual indicators. For the graphical user interface (GUI) we use the JAVA swing and awt packages. In these packages there are many components such as JFrame, JPanel, JButton, JTextField, ActionEven, ActionListener.

The **javax.swing.JFrame** class is a type of container which inherits the JFrame class. The JPanel is a simplest container class. It provides space in which an application can attach any other component. It inherits the JComponents class such as JButton. The JPanel is a simplest container class. It provides space in which an application can attach any other component. It inherits the JComponents class such as JButton. The object of a JTextField class is a text component that allows the editing of a single line text. It inherits JTextComponent class.

An ActionListener is an object listening for an action to occur, such as something being clicked. An ActionEvent is the event that occurred. So say you click on a JButton, an ActionEvent is fired containing the object that the event occurred to, and some other info.

Code:

```
import javax.swing.*;
import java.awt.event.*;
public class BillGeneration extends JFrame implements ActionListener {
    JLabel l, lPizza, lBurger, lTea;
    JTextField tfPizza, tfBurger, tfTea;
    JButton b;
    JTextArea ta;
    BillGeneration() {
        l = new JLabel("Food Ordering System");
        l.setBounds(50, 50, 300, 20);
        lPizza = new JLabel("Pizza @ 100");
        lPizza.setBounds(100, 100, 150, 20);
        tfPizza = new JTextField();
        tfPizza.setBounds(250, 100, 50, 20);
        lBurger = new JLabel("Burger @ 30");
        lBurger.setBounds(100, 150, 150, 20);
        tfBurger = new JTextField();
        tfBurger.setBounds(250, 150, 50, 20);
        lTea = new JLabel("Tea @ 10");
        lTea.setBounds(100, 200, 150, 20);
        tfTea = new JTextField();
        tfTea.setBounds(250, 200, 50, 20);
    }
}
```

```

        b = new JButton("Order");
        b.setBounds(100, 250, 80, 30);
        b.addActionListener(this);
        ta = new JTextArea();
        ta.setBounds(100, 300, 200, 100);
        add(l);
        add(lPizza);
        add(tfPizza);
        add(lBurger);
        add(tfBurger);
        add(lTea);
        add(tfTea);
        add(b);
        add(ta);

        setSize(400, 500);
        setLayout(null);
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
    public void actionPerformed(ActionEvent e) {
        float amount = 0;
        StringBuilder msg = new StringBuilder();
        int pizzaQuantity = Integer.parseInt(tfPizza.getText());
        amount += 100 * pizzaQuantity;
        msg.append("Pizza: ").append(pizzaQuantity).append(" x 100\n");
        int burgerQuantity = Integer.parseInt(tfBurger.getText());
        amount += 30 * burgerQuantity;
        msg.append("Burger: ").append(burgerQuantity).append(" x 30\n");
        int teaQuantity = Integer.parseInt(tfTea.getText());
        amount += 10 * teaQuantity;
        msg.append("Tea: ").append(teaQuantity).append(" x 10\n");
        msg.append("-----\n");
        ta.setText(msg + "Total: " + amount);
    }
    public static void main(String[] args) {
        new BillGeneration();
    }
}

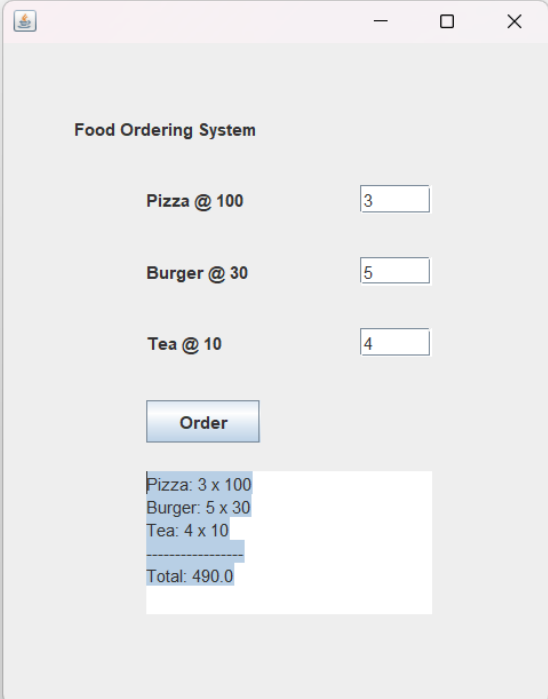
```

Input:

Pizza: 3 x 100

Burger: 5 x 30

Tea: 4 x 10

Output:

The screenshot shows a Java Swing window titled "Food Ordering System". It contains three input fields for quantities: "Pizza @ 100" with value 3, "Burger @ 30" with value 5, and "Tea @ 10" with value 4. Below these is an "Order" button. A text area at the bottom displays the following text:

```
Pizza: 3 x 100
Burger: 5 x 30
Tea: 4 x 10
-----
Total: 490.0
```

Experiment Number: 03

Name of the experiment: To write a JAVA Program to Create a Student form in GUI.

Theory: A graphical user interface, or GUI is a form of user interface that allows users to interact with electronic devices through graphical icons and visual indicators. For the graphical user interface (GUI) we use the JAVA swing and awt packages. In these packages there are many components such as JFrame, JPanel, JButton, JTextField, ActionListener, and ActionEvent.

The **javax.swing.JFrame** class is a type of container which inherits the JFrame class. The JPanel is a simplest container class. It provides space in which an application can attach any other component. It inherits the JComponent class such as JButton. The JPanel is a simplest container class. It provides space in which an application can attach any other component. It inherits the JComponent class such as JButton. The object of a JTextField class is a text component that allows the editing of a single line text. It inherits JTextComponent class.

An ActionListener is an object listening for an action to occur, such as something being clicked. An ActionEvent is the event that occurred. So say you click on a JButton, an ActionEvent is fired containing the object that the event occurred to, and some other info.

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 FormI implements ActionListener {
    private static JLabel success;
    private static JFrame frame;
    private static JLabel label1, label2, label3, label4; // Added label4 for
blood group
    private static JPanel panel;
    private static JButton button;
    private static JTextField userText1, userText2, userText3, userText4;
    public static void main(String[] args) {
        frame = new JFrame();
        panel = new JPanel();
        frame.setSize(400, 350);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(panel);
```

```

        panel.setLayout(null);
        label1 = new JLabel("Name");
        label1.setBounds(10, 10, 80, 25);
        panel.add(label1);
        label2 = new JLabel("Roll");
        label2.setBounds(10, 60, 80, 25);
        panel.add(label2);
        label3 = new JLabel("Department");
        label3.setBounds(10, 110, 80, 25);
        panel.add(label3);
        label4 = new JLabel("Blood Group");
        label4.setBounds(10, 160, 80, 25);
        panel.add(label4);
        userText1 = new JTextField("Enter Your Name");
        userText1.setBounds(100, 10, 200, 25);
        panel.add(userText1);
        JTextField userText2 = new JTextField("Enter Your Roll");
        userText2.setBounds(100, 60, 200, 25);
        panel.add(userText2);
        JTextField userText3 = new JTextField("Enter Your Department");
        userText3.setBounds(100, 110, 200, 25);
        panel.add(userText3);
        userText4 = new JTextField("Enter Your Blood Group"); // New text field
for blood group
        userText4.setBounds(100, 160, 200, 25); // Adjusted position
        panel.add(userText4);
        button = new JButton("Save");
        button.setBounds(150, 210, 80, 25); // Adjusted position
        button.addActionListener(new FormI());
        panel.add(button);
        success = new JLabel("");
        success.setBounds(130, 260, 300, 25); // Adjusted position
        panel.add(success);
        frame.setVisible(true);
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        success.setText("Saved Successfully");
    }
}

```

Input:

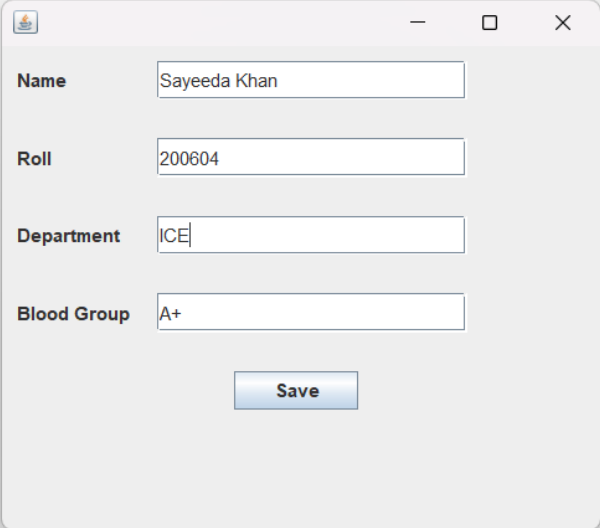
Sayeeda Khan

200604

ICE

A+

Output:



The image shows a screenshot of a Java Swing window titled "Form". The window has a standard Mac OS-style title bar with a red, yellow, and green button on the left, and minus, maximize, and close buttons on the right. Inside the window, there are four text input fields arranged vertically, each with a label to its left: "Name" (containing "Sayeeda Khan"), "Roll" (containing "200604"), "Department" (containing "ICE"), and "Blood Group" (containing "A+"). Below these fields is a single "Save" button. The window is set against a light gray background.

Experiment Number: 04

Name of the experiment: To write a JAVA Program to develop a simple calculator in GUI.

Theory: A graphical user interface, or GUI is a form of user interface that allows users to interact with electronic devices through graphical icons and visual indicators. For the graphical user interface (GUI) we use the JAVA swing and awt packages. In these packages there are many components such as JFrame, JPanel, JButton, JTextField, ActionListener, and ActionEvent.

The **javax.swing.JFrame** class is a type of container which inherits the JFrame class. The JPanel is a simplest container class. It provides space in which an application can attach any other component. It inherits the JComponent class such as JButton. The JPanel is a simplest container class. It provides space in which an application can attach any other component. It inherits the JComponent class such as JButton. The object of a JTextField class is a text component that allows the editing of a single line text. It inherits JTextComponent class.

An ActionListener is an object listening for an action to occur, such as something being clicked. An ActionEvent is the event that occurred. So say you click on a JButton, an ActionEvent is fired containing the object that the event occurred to, and some other info.

Code:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class calculator extends JFrame implements ActionListener {
    JButton[] numButtons;
    JButton addButton, subButton, mulButton, divButton, equalsButton,
    clearButton;
    JTextField displayField;
    int num1, num2;
    char operator;
    public calculator() {
        super("Calculator");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel contentPane = new JPanel(new BorderLayout());
        JPanel buttonPanel = new JPanel(new GridLayout(4, 4, 5, 5)); // Add some
        spacing between buttons
        displayField = new JTextField();
        displayField.setEditable(false);
        displayField.setHorizontalAlignment(SwingConstants.RIGHT);
        contentPane.add(displayField, BorderLayout.NORTH);
        numButtons = new JButton[10];
```

```

for (int i = 0; i < 10; i++) {
    numButtons[i] = new JButton(String.valueOf(i));
    numButtons[i].addActionListener(this);
    buttonPanel.add(numButtons[i]);
}
addButton = new JButton("+");
subButton = new JButton("-");
mulButton = new JButton("*");
divButton = new JButton("/");
equalsButton = new JButton("=");
clearButton = new JButton("C");
addButton.addActionListener(this);
subButton.addActionListener(this);
mulButton.addActionListener(this);
divButton.addActionListener(this);
equalsButton.addActionListener(this);
clearButton.addActionListener(this);
buttonPanel.add(addButton);
buttonPanel.add(subButton);
buttonPanel.add(mulButton);
buttonPanel.add(divButton);
buttonPanel.add(equalsButton);
buttonPanel.add(clearButton);
contentPane.add(buttonPanel, BorderLayout.CENTER);
setContentPane(contentPane);
pack(); // Adjusts the frame size to fit its contents
setLocationRelativeTo(null); // Centers the frame on the screen
setVisible(true);
}

public void actionPerformed(ActionEvent e) {
    String command = e.getActionCommand();
    if (command.equals("C")) {
        displayField.setText("");
        num1 = num2 = 0;
        operator = '\u0000';
    } else if (Character.isDigit(command.charAt(0))) {
        displayField.setText(displayField.getText() + command);
    } else if (command.equals("+") || command.equals("-") ||
command.equals("*") || command.equals("/")) {
        num1 = Integer.parseInt(displayField.getText());
        operator = command.charAt(0);
        displayField.setText("");
    } else if (command.equals("=")) {
        num2 = Integer.parseInt(displayField.getText());
        int result = calculate();
    }
}

```

```

        displayField.setText(String.valueOf(result));
    }
}
private int calculate() {
    int result = 0;
    switch (operator) {
        case '+':
            result = num1 + num2;
            break;
        case '-':
            result = num1 - num2;
            break;
        case '*':
            result = num1 * num2;
            break;
        case '/':
            if (num2 != 0) {
                result = num1 / num2;
            } else {
                JOptionPane.showMessageDialog(this, "Cannot divide by zero",
"Error", JOptionPane.ERROR_MESSAGE);
            }
            break;
    }
    return result;
}
}
public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new calculator();
        }
    });
}
}
}

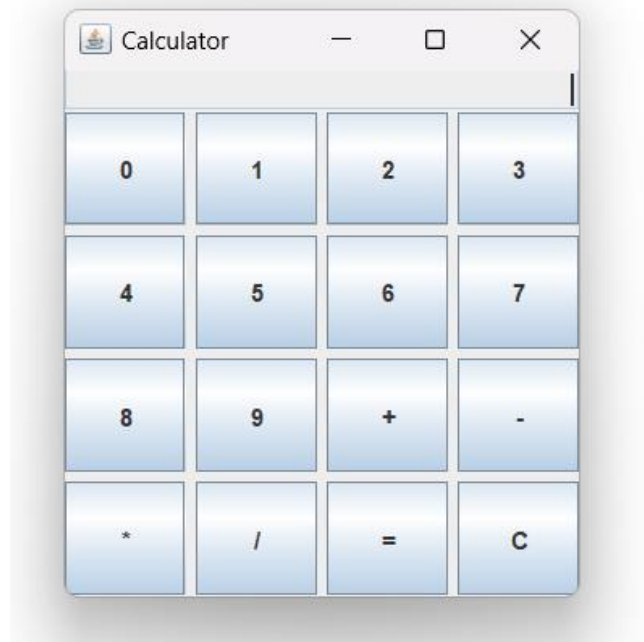
```

Input:

20+10

03+02

Output:



Experiment Number: 05

Name of the experiment: To write a JAVA Program to create java threads using thread class.

Objective: To know the concept how a thread is created using the thread class and how we can run the using the dedicated methods.

Theory: Multithreading is a programming and execution model that allows multiple threads to run concurrently within a single process. Here programming is divided in two or more subprograms which can be implemented in parallel. It supports the execution of multiple parts of a single program. Multithreading is highly efficient. A thread is the smallest unit in unthreading. It helps in developing efficient programs.

Threads are implemented in from object that contains a method called run (). The run () method is the heart and soul of any thread. It makes up the entire body of a thread and is the only method in which thread behaviour can be implemented. We can create a thread by extending the "Thread" class or implementing the runnable interface.

For Extending the thread class we have to create a new class that extends thread class. Overriding the run() method in our created extended class this run() method contains the code that the thread will execute. Instantiate an object of subclass. By calling the start() method on the object to start the execution of thread.

Code:

```
class thread1 extends Thread{
    public void run(){
        for( int i=1; i<=5; i++){
            System.out.println("\t From Thread A : i=" +i);
        }System.out.println("Exit From Thread 1");
    }
}
class thread2 extends Thread{
    public void run(){
        for( int j=1; j<=5; j++){
            System.out.println("\t From Thread A : j=" +j);
        }System.out.println("Exit From Thread 2");
    }
}
class thread3 extends Thread{
    public void run(){
        for( int k=1; k<=5; k++){
            System.out.println("\t From Thread A : k=" +k);
        }System.out.println("Exit From Thread 3");
    }
}
public class SKThread {
    public static void main(String[] args) {
```

```
        new thread1().start();
        new thread2().start();
        new thread3().start();
    }
}
```

Output:

```
From Second Thread : j=1
From Second Thread : j=2
From First Thread : i=1
From Third Thread : k=1
From Third Thread : k=2
From Second Thread : j=3
From First Thread : i=2
From Third Thread : k=3
From Second Thread : j=4
From First Thread : i=3
From First Thread : i=4
From Third Thread : k=4
From Second Thread : j=5
Exit From Thread 2
From First Thread : i=5
From Third Thread : k=5
Exit From Thread 3
Exit From Thread 1
```

Results and Discussion: We can see that is the code three thread has been created and they are running as the thread is assigned to a processor and as we are using the start() method that's why output is not sequential.

Experiment Number: 06

Name of the experiment: To write a JAVA Program to call "run()" method directly from main.

Objective: To know the concept of run() method and how run() method is differ from start() method and while we should use the run() method and what kind of output we get due to the run() method.

Theory: Multithreading is a programming and execution model that allows multiple threads to run concurrently within a single process. Here programming is divided in two or more subprograms which can be implemented in parallel. It supports the execution of multiple parts of a single program. Multithreading is highly efficient. A thread is the smallest unit in unthreading. It helps in developing efficient programs.

Threads are implemented in from object that contains a method called run (). The run () method is the heart and soul of any thread. It makes up the entire body of a thread and is the only method in which thread behaviour can be implemented. We can create a thread by extending the "Thread" class or implementing the runnable interface.

For Extending the thread class we have to create a new class that extends thread class. Overriding the run() method in our created extended class this run() method contains the code that the thread will execute. Instantiate an object of subclass. By calling the start() method on the object to start the execution of thread.

While we use the start() method the execution of thread is random there is not any sequence that time. But if we replace the run() method with start() method there then we can se a sequential output. The threat created first is executed first or the thread called first is executed first. After executing one thread execution of another threads execution starts. As a result we can see a sequential output.

Code:

```
class thread1 extends Thread{
    public void run(){
        for( int i=1; i<=5; i++){
            System.out.println("\t From First Thread : i=" +i);
        }System.out.println("Exit From Thread 1");
    }
}class thread2 extends Thread{
    public void run(){
        for( int j=1; j<=5; j++){
            System.out.println("\t From Second Thread : j=" +j);
        }System.out.println("Exit From Thread 2");
    }
}class thread3 extends Thread{
    public void run(){
        for( int k=1; k<=5; k++){
```

```

        System.out.println("\t From  Third Thread : k=" +k);
    }System.out.println("Exit From Thread 3");
}
}public class SKThreadRun {
    public static void main(String[] args) {
        new thread1().run();
        new thread2().run();
        new thread3().run();
    }
}
}

```

Output:

From First Thread : i=1

From First Thread : i=2

From First Thread : i=3

From First Thread : i=4

From First Thread : i=5

Exit From Thread 1

From Second Thread : j=1

From Second Thread : j=2

From Second Thread : j=3

From Second Thread : j=4

From Second Thread : j=5

Exit From Thread 2

From Third Thread : k=1

From Third Thread : k=2

From Third Thread : k=3

From Third Thread : k=4

From Third Thread : k=5

Exit From Thread 3

Results and Discussion: We can see that in run method we get an output which is sequential. That means run() method produces a sequential output whereas start() method doesn't provide that.

Experiment Number: 07

Name of the experiment: To write a JAVA Program to use “yield ()”, “stop ()” and “sleep ()” method.

Objective: To know the general form of the three methods and why these methods are used and what are the benefits of the methods and where to use these methods.

Theory Multithreading is a programming and execution model that allows multiple threads to run concurrently within a single process. Here programming is divided in two or more subprograms which can be implemented in parallel. It supports the execution of multiple parts of a single program. Multithreading is highly efficient. A thread is the smallest unit in unthreading. It helps in developing efficient programs.

Threads are implemented in from object that contains a method called run (). The run () method is the heart and soul of any thread. It makes up the entire body of a thread and is the only method in which thread behaviour can be implemented. We can create a thread by extending the “Thread” class or implementing the runnable interface.

For Extending the thread class we have to create a new class that extends thread class. Overriding the run() method in our created extended class this run() method contains the code that the thread will execute. Instantiate an object of subclass. By calling the start() method on the object to start the execution of thread.

While we use the start() method the execution of thread is random there is not any sequence that time. But if we replace the run() method with start() method there then we can see a sequential output. The thread created first is executed first or the thread called first is executed first. After executing one thread execution of another threads execution starts. As a result we can see a sequential output.

Yield() method in java multithreading is an important one. When at some condition a thread finds yield() method it stops executing for some time and gives opportunity to the other thread to execute and after a while the thread temporarily stops execution that that starts executing.

Stop() method permanently stops the execution of that thread. It is close to suspend() method but the main difference between suspend () method and stop () method is suspend() method can start execution again when it got resume () method but when a thread got message of stop() method at any condition that permanently stops its execution and it can't execute again.

Sleep() method is another method that used to sleep a thread for a time period where the time is measured in milliseconds. After the time period the thread will execute automatically. It is necessary to use try-catch block for error handling in the sleep() method because a sleeping thread cannot deal with resume() method because a sleeping thread cannot deal with new instruction. Same goes for suspend() method as well.

Code:

```
class thread1 extends Thread{
    public void run(){
        for( int i=1; i<=5; i++){
            if(i==2)
                thread1.yield();
            System.out.println("\t From First Thread : i=" +i);
        }System.out.println("Exit From Thread 1");
    }
}
class thread2 extends Thread{
    @SuppressWarnings("removal")
    public void run(){
        for( int j=1; j<=5; j++){
            if(j==3){
                stop();
            }
            System.out.println("\t From Second Thread : j=" +j);
        }System.out.println("Exit From Thread 2");
    }
}
class thread3 extends Thread{
    public void run(){
        for( int k=1; k<=5; k++){
            if(k==1)
                try {
                    sleep(1000);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            System.out.println("\t From Third Thread : k=" +k);
        }System.out.println("Exit From Thread 3");
    }
}
public class SKThreadThreeMethod {
    public static void main(String[] args) {
        thread1 Thread_01 = new thread1();
        thread2 Thread_02 = new thread2();
        thread3 Thread_03 = new thread3();
        System.out.println("Execution Started");
        System.out.println("Start The First Thread");
        Thread_01.start();
        System.out.println("Start The Second Thread");
        Thread_02.start();
        System.out.println("Start The Third Thread");
        Thread_03.start();
    }
}
```

Output:

Execution Started

Start The First Thread

Start The Second Thread

Start The Third Thread

End of Main Thread

From First Thread : i=1

From First Thread : i=2

From Second Thread : j=1

From First Thread : i=3

From Second Thread : j=2

From First Thread : i=4

From First Thread : i=5

Exception in thread "Thread-1" Exit From Thread 1

java.lang.UnsupportedOperationException

at java.base/java.lang.Thread.stop(Thread.java:1667)

at thread2.run(SKThreadThreeMethod.java:16)

From Third Thread : k=1

From Third Thread : k=2

From Third Thread : k=3

From Third Thread : k=4

From Third Thread : k=5

Exit From Thread 3

Results and Discussion: We can see that when condition hits the `yield()` method gives opportunity to other threads to execute. On the other hand `stop()` method doesn't stop a thread permanently from executing it stops the particular thread for a particular condition. And finally, the `sleep()` method used to sleep a thread but there is a chance of error in the method because a sleeping thread cannot receive commands that's why a try catch block is used for error handling.

Experiment Number: 08

Name of the experiment: To write a Java program to use priority of thread.

Objective: To know the basic concept about thread priority and how it works and also know about the Minimum, Normal and Maximum Priority.

Theory: Multithreading is a programming and execution model that allows multiple threads to run concurrently within a single process. Here programming is divided in two or more subprograms which can be implemented in parallel. It supports the execution of multiple parts of a single program. Multithreading is highly efficient. A thread is the smallest unit in unthreading. It helps in developing efficient programs.

Thread priorities in Java allow developers to specify the relative importance of thread's execution. In java, each thread assigned a priority, which affects the order in which it is scheduled for running. The thread of the same priority is given equal treatment by the Java scheduler. Java permit us to set the priority of a thread using the set priority method as

```
ThreadName.setPriority( int number);
```

The int number is an integer value to which the thread priority is set. The Thread class defines several priority constants

```
MIN_PRIORITY=1
```

```
NORM_PRIORITY=5
```

```
MAX_PRIORITY=10
```

By assigning priorities to thread we can ensure that they are given the attention they deserve.

Code:

```
class MyThread extends Thread {
    public MyThread(String name) {
        super(name);
    }
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(getName() + " is running with priority: " +
getPriority() + ", Count: " + i);
            try {
                Thread.sleep(1000); // Simulate some work
            } catch (InterruptedException e) {
                System.out.println(getName() + " interrupted.");
            }
        }
    }
}
```

```

public class ThreadPriority {
    public static void main(String[] args) {
        MyThread t1 = new MyThread("Thread 1");
        MyThread t2 = new MyThread("Thread 2");
        MyThread t3 = new MyThread("Thread 3");
        t1.setPriority(Thread.MIN_PRIORITY); // Set lowest priority
        t2.setPriority(Thread.NORM_PRIORITY); // Set normal priority
        t3.setPriority(Thread.MAX_PRIORITY); // Set highest priority
        t1.start();
        t2.start();
        t3.start();
    }
}

```

Output:

```

Thread 2 is running with priority: 5, Count: 1
Thread 3 is running with priority: 10, Count: 1
Thread 1 is running with priority: 1, Count: 1
Thread 3 is running with priority: 10, Count: 2
Thread 2 is running with priority: 5, Count: 2
Thread 1 is running with priority: 1, Count: 2
Thread 2 is running with priority: 5, Count: 3
Thread 3 is running with priority: 10, Count: 3
Thread 1 is running with priority: 1, Count: 3
Thread 2 is running with priority: 5, Count: 4
Thread 1 is running with priority: 1, Count: 4
Thread 3 is running with priority: 10, Count: 4
Thread 2 is running with priority: 5, Count: 5
Thread 3 is running with priority: 10, Count: 5
Thread 1 is running with priority: 1, Count: 5

```

Results and Discussion: We can see that we have total three threads and each thread has given a priority and in the output we can notice that particular thread is running with the assigned priority.

Experiment Number: 09

Name of the experiment: To write a client server program in Java to establish a connection between them.

Objective: To establish connection between client side and server side through which client can communicate with the server.

Theory: Client-server programming in Java involves building applications where one program, the client, requests services from another program, the server, over a network. Leveraging Java's built-in support for socket programming through the `java.net` package, developers create sockets for communication between client and server endpoints. Servers, established using the `ServerSocket` class, bind to specific ports and listen for incoming connections via the `accept()` method. Upon connection, servers handle client requests, often asynchronously to support multiple clients concurrently, by reading input streams and sending responses through output streams. Conversely, clients utilize the `Socket` class to connect to server IP addresses and ports, enabling data exchange with servers. Implementation necessitates adherence to communication protocols such as HTTP or custom protocols, along with robust error handling mechanisms to manage exceptions and ensure application stability. Security considerations, including encryption and secure communication protocols, play a vital role in protecting data integrity and confidentiality. With Java's rich networking APIs and libraries, developers can create diverse client-server applications, spanning from basic chat servers to intricate distributed systems, with flexibility and scalability.

Code:

Server Side

```
import java.io.*;
import java.net.*;
public class Server {
    public static void main(String[] args) {
        final int PORT = 12345;
        try {
            ServerSocket serverSocket = new ServerSocket(PORT);
            System.out.println("Server started. Waiting for client...");
            Socket clientSocket = serverSocket.accept();
            System.out.println("Client connected: " + clientSocket);
            BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
            PrintWriter out = new
PrintWriter(clientSocket.getOutputStream(), true);
            String messageFromClient;
            while ((messageFromClient = in.readLine()) != null) {
                System.out.println("Client: " + messageFromClient);
                out.println("Message received: " + messageFromClient);
            }
        }
    }
}
```

```

        clientSocket.close();
        serverSocket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

Client Side

```

import java.io.*;
import java.net.*;
public class Client {
    public static void main(String[] args) {
        final String SERVER_ADDRESS = "localhost";
        final int PORT = 12345;
        try {
            Socket socket = new Socket(SERVER_ADDRESS, PORT);
            System.out.println("Connected to server: " + socket);
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(),
true);

            BufferedReader userInput = new BufferedReader(new
InputStreamReader(System.in));
            String messageToServer;
            while (true) {
                System.out.print("Enter message to send to server: ");
                messageToServer = userInput.readLine();
                if ("exit".equalsIgnoreCase(messageToServer)) {
                    break;
                }
                out.println(messageToServer);
                System.out.println("Server: " + in.readLine());
            }
            }socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Output:

Server started. Waiting for client...

Connected to server: Socket[addr=localhost/127.0.0.1,port=12345,localport=10872]

Enter message to send to server: HI

Server: Message received: HI

Enter message to send to server: Hello

Server: Message received: Hello

Enter message to send to server: Hi

Server: Message received: Hi

Enter message to send to server:

Results and Discussion: We can see that the code has able to make connection between the server and the client through which the server and client can communicate with each other. That means the experiment was successful.