

Experiment Number: 01

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

Theory:

This Java code is a simple Swing GUI (Graphical User Interface) application that displays two images in a JFrame (Java Frame)

Certainly! Here's a theoretical explanation of the code without delving into the specific Java syntax:

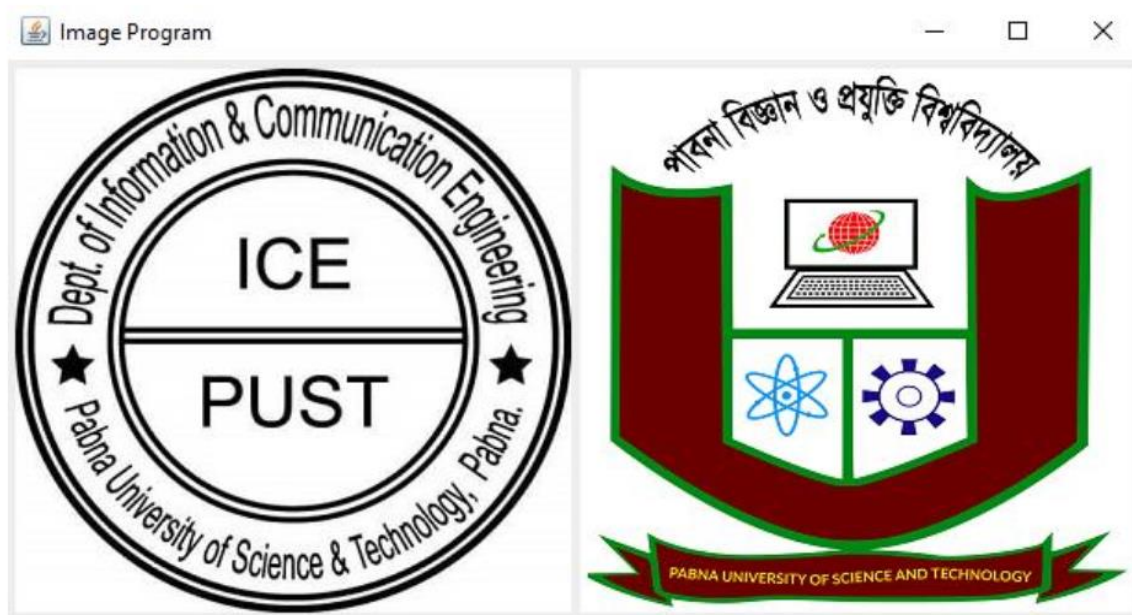
This Java program demonstrates the creation of a simple graphical user interface (GUI) application using Swing, which is a set of GUI components provided by Java. Let's break down the key components and concepts:

1. **Swing Framework:** Swing is a GUI widget toolkit for Java. It provides a comprehensive set of GUI components, such as buttons, labels, text fields, etc., to create interactive user interfaces.
2. **JFrame:** JFrame is a top-level container provided by Swing to represent a window in a GUI application. It can contain various GUI components and manages their layout and behavior.
3. **Layout Management:** The FlowLayout layout manager is used in this program. Layout managers in Swing are responsible for arranging the components within a container. FlowLayout arranges components in a left-to-right flow, wrapping to the next line when the current line is full.
4. **ImageIcon:** ImageIcon is a class in Java that represents an image icon. It can load images from files or URLs and is often used to display images in GUI applications.
5. **JLabel:** JLabel is a Swing component used to display text, images, or both. In this program, JLabel is used to display the images loaded using ImageIcon.
6. **getResource():** The getResource() method is used to load resources such as images from the classpath. It returns a URL object that represents the location of the specified resource.
7. **Constructor:** The constructor of the Image class is responsible for setting up the GUI components. It sets the layout manager, loads images using ImageIcon, creates JLabel instances with these images, and adds them to the frame.
8. **Main Method:** The main method is the entry point of the program. It creates an instance of the Image class, configures the frame's behavior (e.g., default close operation, visibility), adjusts the frame's size based on its contents, and sets the title of the frame.

Source Code:

```
import java.awt.FlowLayout;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
public class Image extends JFrame { private ImageIcon image1; private JLabel
label1; private ImageIcon image2; private JLabel label2;
Image(){
setLayout(new FlowLayout());
image1 = new ImageIcon(getClass().getResource("ice.jpg")); label1 = new
JLabel(image1);
add(label1);
image2 = new ImageIcon(getClass().getResource("pust.png"));
label2 = new JLabel(image2);
add(label2);
}
public static void main(String args[]) {
Image gui = new Image();
gui.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); gui.setVisible(true);
gui.pack(); gui.setTitle("Image Program");
}
}
```

Output:



Experiment Number: 02

Experiment Name: Write a JAVA Program for generating Restaurant Bill

Theory:

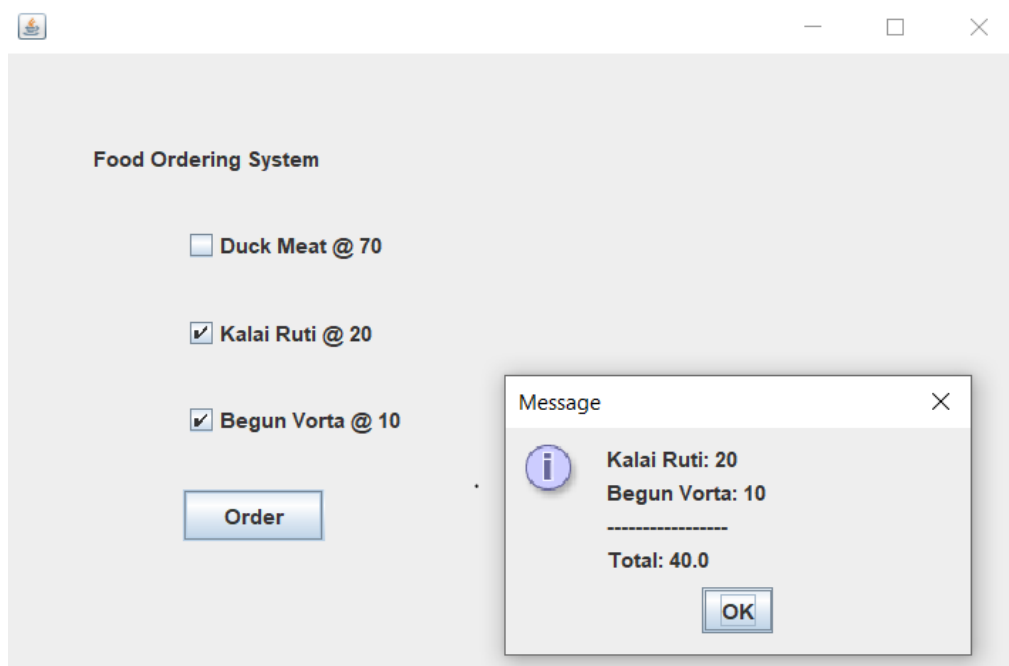
This Java code creates a basic graphical user interface (GUI) application for a food ordering system using Swing components. Let's break down its theoretical aspects:

1. **Swing Framework:** The code utilizes Swing, a GUI toolkit provided by Java, to create interactive GUI applications.
2. **JFrame:** JFrame is a class in Swing used to create a window for GUI applications. In this code, the BillGeneration class extends JFrame, indicating that it represents the main window of the application.
3. **Components:**
 - **JLabel** (l): A component for displaying text. It shows the title "Food Ordering System".
 - **JCheckBox** (cb1, cb2, cb3): Checkboxes for selecting food items. Users can choose from Pizza, Burger, and Tea.
 - **JButton** (b): A button labeled "Order" for placing the order.
4. **Layout Management:** The code uses absolute positioning (null layout) to place components at specific coordinates on the frame. This approach is simple but not recommended for complex GUIs.
5. **Event Handling:**
 - The class implements the ActionListener interface, which defines a method actionPerformed(ActionEvent e).
 - The "Order" button (b) is associated with an action listener (this), meaning the BillGeneration class itself listens for events from this button.
 - When the button is clicked, the actionPerformed method is triggered.
6. **Action Event Handling:**
 - In the actionPerformed method, the code checks which checkboxes are selected.
 - It calculates the total amount based on the selected items. Each selected item contributes to the total amount.
 - A message is constructed with the details of the selected items and the total amount.
 - A dialog box (JOptionPane.showMessageDialog) displays this message.
7. **Main Method:**
 - The main method creates an instance of the BillGeneration class, initializing the GUI application.
 - It doesn't explicitly handle threading issues, which might be necessary for more complex applications.

Source Code:

```
import javax.swing.*;
import java.awt.event.*;
public class BillGeneration extends JFrame implements ActionListener{
    JLabel l;
    JCheckBox cb1,cb2,cb3; JButton b; BillGeneration(){
    l=new JLabel("Food Ordering System"); l.setBounds(50,50,300,20);
    cb1=new JCheckBox("Pizza @ 100"); cb1.setBounds(100,100,150,20); cb2=new
    JCheckBox("Burger @ 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+=100; msg="Pizza: 100\n";}
    if(cb2.isSelected()){ amount+=30; msg+="Burger: 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 BillGeneration();
    }}
```

Output:



Experiment Number: 03

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

Theory:

This Java program creates a simple form GUI application using Swing components. Let's break down its theoretical aspects:

1. Swing Components:

- **JFrame:** Represents the main window of the application.
- **JPanel:** A container that holds other components.
- **JLabel:** Displays text.
- **JTextField:** Allows users to input text.
- **JButton:** A button that triggers an action when clicked.

2. Main Method:

- Creates an instance of JFrame named frame.
- Initializes a JPanel named panel and sets its layout to null (custom positioning of components).
- Sets the size of the frame to 400x300 pixels.
- Sets the default close operation to exit the application when the frame is closed.
- Adds the panel to the frame.

3. Labels and TextFields:

- Creates labels (label1, label2, label3) for "Name", "Roll", and "Department" respectively.
- Positions these labels on the panel using `setBounds`.
- Creates text fields (userText1, userText2, userText3) for user input.
- Positions these text fields on the panel using `setBounds`.
- Sets default text in the text fields.

4. Button:

- Creates a button named "Save".
- Positions the button on the panel using `setBounds`.
- Registers an instance of the form class (which implements `ActionListener`) as the listener for button clicks.
- Adds the button to the panel.

5. Action Event Handling:

- The `actionPerformed` method is invoked when the button is clicked.
- It sets the text of the success label to "Saved Successfully".

6. Visibility:

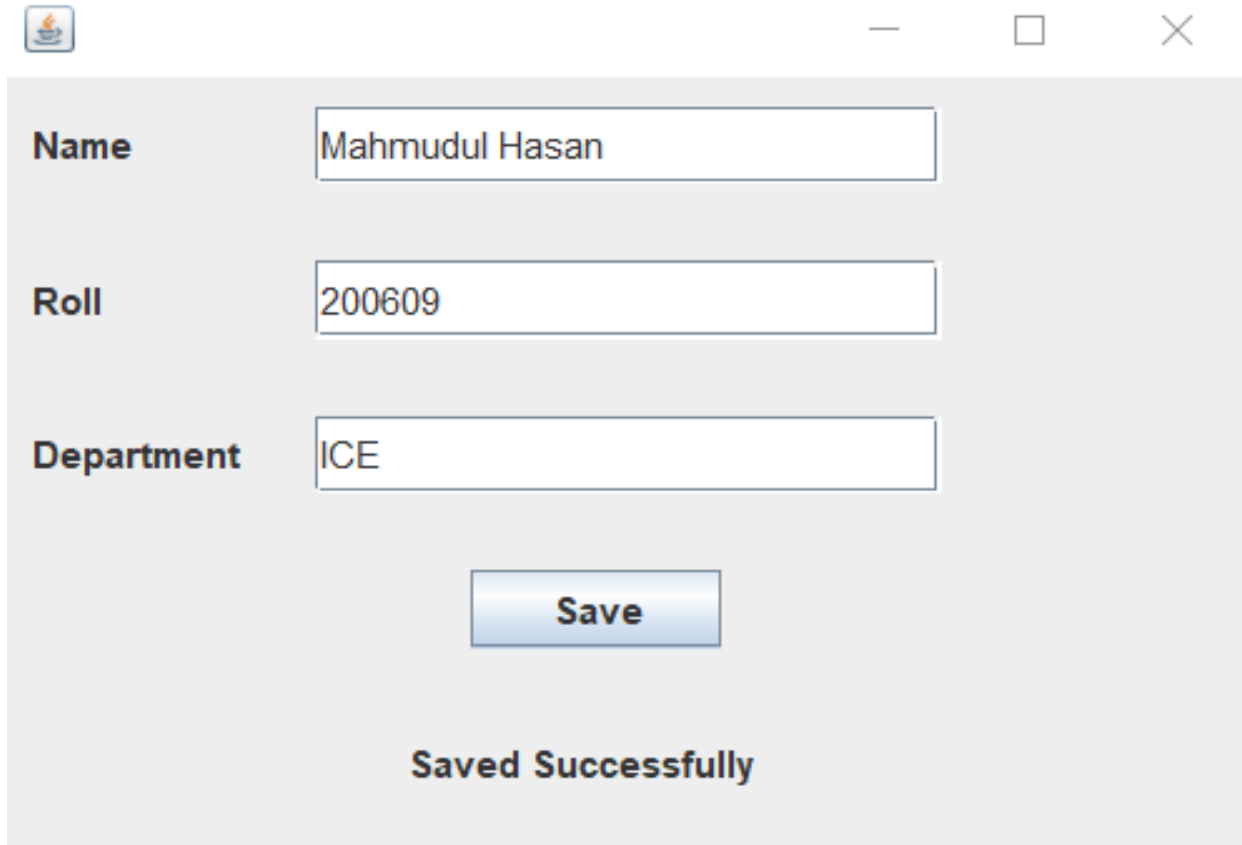
- Sets the visibility of the frame to `true`, making it visible to the user.

In summary, this code creates a simple form GUI application where users can enter their name, roll, and department. Upon clicking the "Save" button, it displays a message indicating that the information has been saved successfully.

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 form implements ActionListener {
    private static JLabel success;
    private static JFrame frame;
    private static JLabel label1,label2,label3;
    private static JPanel panel;
    private static JButton button;
    private static JTextField userText1, userText2, userText3;
    public static void main(String[] args) {
        frame = new JFrame();
        panel = new JPanel();
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(panel); panel.setLayout(null);
        //Setting all Three Labels
        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);
        //Creating all Textfields
        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");
        userText3.setBounds(100,110,200,25); panel.add(userText3);
        button = new JButton("Save"); button.setBounds(150, 160, 80, 25);
        button.addActionListener(new form()); 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:



A screenshot of a web application window. The window has a title bar with a small icon on the left and standard minimize, maximize, and close buttons on the right. The main content area is light gray and contains a form with three labeled input fields:

- Name**: A text input field containing the value "Mahmudul Hasan".
- Roll**: A text input field containing the value "200609".
- Department**: A text input field containing the value "ICE".

Below these fields is a blue button with the text "Save". At the bottom center of the form area, the text "Saved Successfully" is displayed in a bold, black font.

Experiment Number: 04

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

Theory:

Certainly! Let's break down the theoretical aspects of this Java calculator program:

1. Swing Components:

- **JFrame:** Represents the main window of the calculator.
- **JPanel:** A container that holds the buttons.
- **JButton:** Buttons for numbers (0-9), arithmetic operations (+, -, *, /, =), and clear (C).
- **JTextField:** Displays the input and output.

2. Constructor:

- Sets the title of the frame to "Calculator".
- Creates a panel (JPanel) with a grid layout to arrange the buttons.
- Initializes an array of buttons (b[]) for numbers (0-9).
- Adds buttons to the panel and registers action listeners for each button.
- Creates additional buttons for arithmetic operations and clear.
- Adds these buttons to the panel and registers action listeners for them.
- Creates a text field (JTextField) to display input and output.
- Adds the panel to the center and the text field to the top of the frame.
- Sets the visibility of the frame to true and sets its size.

3. Action Event Handling:

- Implements the ActionListener interface.
- Defines the actionPerformed method to handle button clicks.
- Determines the source of the action event (which button was clicked).
- If the clear button is clicked (b15), resets the input and output.
- If the equals button is clicked (b14), evaluates the expression and displays the result.
- If an arithmetic operation button is clicked, stores the first operand (n1) and the operation.
- If a number button is clicked, appends the corresponding digit to the input.

4. Evaluation Method:

- Evaluates the expression based on the stored operation and operands.
- Updates the result (r) accordingly.

5. Main Method:

- Creates an instance of the calculator class to start the application.

In summary, this program creates a simple calculator GUI application using Swing components. It allows users to perform basic arithmetic operations by clicking buttons and displays the result in a text field.

Source Code:

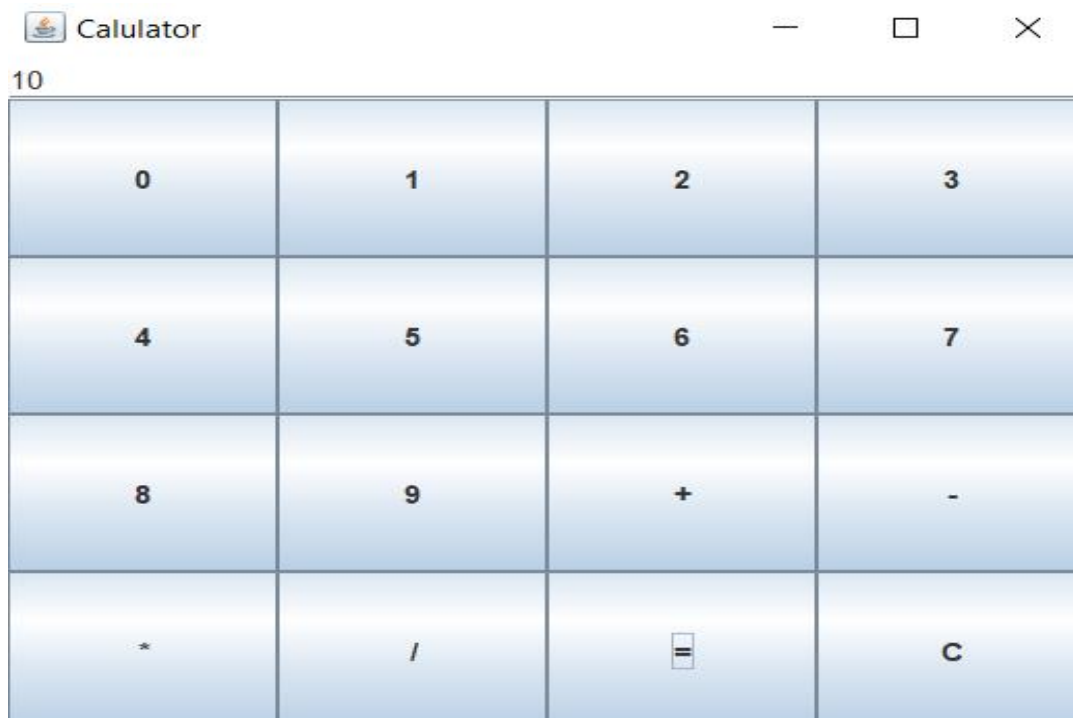
```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
public class calculator extends JFrame implements ActionListener
{
    JButton b10,b11,b12,b13,b14,b15;
    JButton b[]=new JButton[10];
    int i,r,n1,n2; JTextField res; char op;
    public calculator()
    {
        super("Calulator"); setLayout(new BorderLayout());
        JPanel p=new JPanel(); p.setLayout(new GridLayout(4,4)); for(int i=0;i<=9;i++)
        {
            b[i]=new JButton(i+""); p.add(b[i]); b[i].addActionListener(this);}
        b10=new JButton("+"); p.add(b10); b10.addActionListener(this);
        b11=new JButton("-"); p.add(b11); b11.addActionListener(this);
        b12=new JButton("*"); p.add(b12); b12.addActionListener(this);
        b13=new JButton("/"); p.add(b13); b13.addActionListener(this);
        b14=new JButton("="); p.add(b14); b14.addActionListener(this);
        b15=new JButton("C"); p.add(b15); b15.addActionListener(this);
        res=new JTextField(10); add(p,BorderLayout.CENTER);
        add(res,BorderLayout.NORTH); setVisible(true); setSize(200,200);
    }
    public void actionPerformed(ActionEvent ae)
    {
        JButton pb=(JButton)ae.getSource();
        if(pb==b15)
        { r=n1=n2=0;
          res.setText(""); }
        else if(pb==b14)
        { n2=Integer.parseInt(res.getText()); eval();
          res.setText(""+r); }
        else{
            boolean opf=false;
            if(pb==b10)
            { op='+';
              opf=true; }
            if(pb==b11) { op='-';opf=true;}
```

```

if(pb==b12) { op='*';opf=true;}
if(pb==b13) { op='/';opf=true;}
if(opf==false)
{ for(i=0;i<10;i++) {
if(pb==b[i]) {
String t=res.getText(); t+=i;
res.setText(t); }
} }
else{
n1=Integer.parseInt(res.getText()); res.setText("");
} }
} int eval()
{
switch(op) {
case '+':
case '-':          r=n1+n2;    break;
case '*':          r=n1-n2;    break;
case '/':
r=n1*n2; break;
r=n1/n2; break;
}
return 0;
}
public static void main(String arg[]){
new calculator();
} }

```

Output:



Experiment Number: 05

Experiment Name: Write a Java program to create thread using thread class.

Objective: The objective of this lab exercise is to understand how to create threads in Java using the **Thread** class.

Theory:

In Java, a thread can be created by extending the **Thread** class. The **Thread** class provides several constructors and methods to control the behaviour of threads. To create a new thread, we need to extend the **Thread** class and override its **run()** method, which defines the task the thread will execute.

A Thread class has several methods and constructors which allow us to perform various operations on a thread. The Thread class extends the Object class. The Object class implements the Runnable interface.

Runnable Interface (run() method): The Runnable interface is required to be implemented by that class whose instances are intended to be executed by a thread. The runnable interface gives us the run() method to perform an action for the thread. The method is used for starting a thread that we have newly created. It starts a new thread with a new call stack. After executing the start() method, the thread changes the state from New to Runnable. It executes the run() method when the thread gets the correct time to execute it.

Here are some key points about threads in Java:

Creation: Threads can be created by extending the Thread class and overriding its run() method, or by implementing the Runnable interface and passing an instance of the Runnable to a Thread object.

Lifecycle: Threads have a lifecycle that includes states such as new, runnable, blocked, waiting, and terminated. Java provides methods to manage the lifecycle of threads, such as start() to begin execution, sleep() to pause execution for a specified time, and join() to wait for a thread to finish execution.

Concurrency: Java supports concurrent execution of threads, allowing multiple threads to run simultaneously. However, developers need to be cautious about thread safety and synchronization to avoid race conditions and data inconsistencies.

Synchronization: Java provides synchronization mechanisms such as synchronized blocks and methods, as well as explicit locks (java.util.concurrent.locks.Lock) to control access to shared resources and ensure thread safety.

Source Code:

<pre>class A extends Thread { public void run() { for (int i = 1; i <= 5; i++) { System.out.println("\nFrom Thread A: i = " + i); } System.out.println("Exit from A"); } } class B extends Thread { public void run() { for (int j = 1; j <= 5; j++) { System.out.println("\nFrom Thread B: j = " + j); } System.out.println("Exit from B"); } }</pre>	<pre>class C extends Thread { public void run() { for (int k = 1; k <= 5; k++) { System.out.println("\nFrom Thread C: k = " + k); } System.out.println("Exit from C"); } } public class ThreadProgram1_SKR { public static void main(String args[]) { // A Athread=new A(); // Athread.start(); new A().start(); new B().start(); new C().start(); } }</pre>
--	---

Output:

From Thread A: i = 1	From Thread C: k = 3
From Thread A: i = 2	From Thread B: j = 3
From Thread C: k = 1	From Thread A: i = 5
From Thread B: j = 1	From Thread C: k = 4
From Thread A: i = 3	From Thread B: j = 4
From Thread C: k = 2	Exit from A
From Thread B: j = 2	From Thread C: k = 5
From Thread A: i = 4	From Thread B: j = 5
	Exit from B
	Exit from C

Experiment Number: 06

Experiment Name: Write a Java program to call threads using **run()** method.

Objective: The objective of this lab exercise is to understand the basics of multithreading in Java by implementing threads using the **run()** method

Theory:

In Java, multithreading allows concurrent execution of multiple tasks within a program. Threads can be created by extending the **Thread** class or implementing the **Runnable** interface. The **Runnable** interface provides a way to represent a task or unit of work that can be executed by a thread. It has a single abstract method **run()** that must be implemented to define the behavior of the thread.

Implementing the Runnable Interface: The MyRunnable class implements the Runnable interface. This interface provides a way to define a task that can be executed by a thread.

The run() Method: Inside the MyRunnable class, the run() method is overridden. This method contains the code that will be executed when the thread runs. In this example, a simple loop is used to print messages to the console.

Creating an Object: In the main() method of the Main class, an object of the MyRunnable class is created using the default constructor. This object represents the task that will be executed by a thread.

Calling the run() Method: The run() method of the MyRunnable object is called directly using the object reference myRunnable.run(). Unlike using the start() method, calling run() directly executes the code in the current thread instead of creating a new thread. In this case, it executes the run() method in the main thread.

Source Code:

<pre>class A 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 A"); } } class B extends Thread { public void run(){ for(int j=1;j<=5;j++){ System.out.println("\t From Thread B : j= "+j); } System.out.println("Exit from B"); } }</pre>	<pre>class C extends Thread { public void run(){ for(int k=1;k<=5;k++){ System.out.println("\t From Thread C : k= "+k); } System.out.println("Exit from C"); } } public class threadprogram2_SKR { public static void main(String[] args) { new A().run(); new B().run(); new C().run(); } }</pre>
--	--

Output:

```
From Thread A : i= 1
    From Thread A : i= 2
    From Thread A : i= 3
    From Thread A : i= 4
    From Thread A : i= 5
Exit from A
    From Thread B : j= 1
    From Thread B : j= 2
    From Thread B : j= 3
    From Thread B : j= 4
    From Thread B : j= 5
Exit from B
    From Thread C : k= 1
    From Thread C : k= 2
    From Thread C : k= 3
    From Thread C : k= 4
    From Thread C : k= 5
Exit from C
```

Experiment Number: 07

Experiment Name: Write a Java program to illustrate yield() and sleep() method using thread.

Objective: The objective of this lab exercise is to comprehend and demonstrate the use of thread control methods in Java, specifically yield(), stop(), and sleep().

Theory:

yield() Method: The yield() method is a static method of the Thread class in Java. It causes the currently executing thread to pause and allow other threads of the same priority to execute. If no other threads of the same priority are waiting, the thread continues its execution. yield() is often used to improve the efficiency of the thread scheduler.

stop() Method: The stop() method of the Thread class is used to forcefully terminate a thread. It immediately stops the execution of the thread, potentially leaving the program in an inconsistent state. Due to its unsafe nature, the stop() method is deprecated and should be avoided in favor of other termination mechanisms.

sleep() Method: The sleep() method of the Thread class is used to pause the execution of the current thread for a specified amount of time. It takes the duration of sleep in milliseconds as an argument. sleep() is commonly used to introduce delays in program execution or to simulate time-consuming tasks.

Source Code:

```
public class ThreadProgram3_SKR {  
    public static void main(String[] args)  
    {  
        A threadA = new A();  
        B threadB = new B();  
        C threadC = new C();  
        // Start all threads  
        threadA.start();  
        threadB.start();  
        threadC.start();  
        // Demonstrate yield()  
        System.out.println("Main thread  
executing...");  
    }  
}
```

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


<pre> Thread.yield(); System.out.println("Main thread resumed..."); // Demonstrate sleep() try { System.out.println("Main thread sleeping for 2 seconds..."); Thread.sleep(2000); System.out.println("Main thread woke up after sleeping."); for (int j = 1; j <= 5; j++) { System.out.println("From Main Thread: j = " + j); } } catch (InterruptedException e) { System.out.println("Main thread interrupted while sleeping."); } } </pre>	<pre> System.out.println("From Thread B: j = " + j); } System.out.println("Exit from B"); } } class C extends Thread { public void run() { for (int k = 1; k <= 5; k++) { System.out.println("From Thread C: k = " + k); } System.out.println("Exit from C"); } } </pre>
---	---

Output:

<pre> Main thread executing... Main thread resumed... Main thread sleeping for 2 seconds... From Thread B: j = 1 From Thread B: j = 2 From Thread B: j = 3 From Thread A: i = 1 From Thread C: k = 1 From Thread B: j = 4 From Thread A: i = 2 From Thread C: k = 2 From Thread C: k = 3 From Thread C: k = 4 From Thread B: j = 5 </pre>	<pre> From Thread A: i = 3 From Thread C: k = 5 Exit from C Exit from B From Thread A: i = 4 From Thread A: i = 5 Exit from A Main thread woke up after sleeping. From Main Thread: j = 1 From Main Thread: j = 2 From Main Thread: j = 3 From Main Thread: j = 4 From Main Thread: j = 5 </pre>
---	--

Experiment Number: 08

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

Theory:

In Java, thread priority is an integer value that determines the scheduling priority of a thread. Each thread is assigned a priority that helps the operating system decide how much CPU time should be allocated to each thread relative to other threads. Thread priorities are specified as integer values ranging from 1 (lowest priority) to 10 (highest priority). These priorities are constant values defined in the Thread class:

Thread.MIN_PRIORITY has a value of 1.

Thread.NORM_PRIORITY has a value of 5.

Thread.MAX_PRIORITY has a value of 10.

Here are some key points regarding thread priority in Java:

Scheduling: The thread scheduler in the JVM decides which thread to execute based on its priority. However, thread priority is only a suggestion to the thread scheduler, and it's not guaranteed that a thread with a higher priority will always run before a thread with a lower priority.

Preemptive Multitasking: In preemptive multitasking systems (like most modern operating systems), the thread scheduler can interrupt a lower-priority thread that is currently running to give CPU time to a higher-priority thread. This mechanism allows higher-priority threads to be executed more frequently than lower-priority threads.

Platform Dependence: Thread priorities might not have the same effect on different platforms or JVM implementations. Some platforms may ignore thread priorities altogether, or the granularity of thread priorities may be limited.

Default Priority: If you create a thread without explicitly setting its priority, it inherits the priority of the thread that created it. Typically, threads inherit the priority of the main thread, which is Thread.NORM_PRIORITY.

Setting Priority: You can set the priority of a thread using the setPriority() method of the Thread class. This method accepts an integer argument representing the desired priority.

Source Code:

<pre>public class ThreadProgram4_SKR { public static void main(String[] args) { // Create three threads A threadA = new A(); B threadB = new B(); C threadC = new C(); // Set priorities for the threads threadA.setPriority(Thread.MIN_PRIORITY); // Priority 1 threadB.setPriority(Thread.NORM_PRIORITY); // Priority 5 threadC.setPriority(Thread.MAX_PRIORITY); // Priority 10 // Start the threads threadA.start(); threadB.start(); threadC.start(); } static class A extends Thread { public void run() { for (int i = 1; i <= 5; i++) { System.out.println("From Thread A: i = " + i); } System.out.println("Exit from A"); } } }</pre>	<pre>static class B extends Thread { public void run() { for (int j = 1; j <= 5; j++) { System.out.println("From Thread B: j = " + j); } System.out.println("Exit from B"); } } static class C extends Thread { public void run() { for (int k = 1; k <= 5; k++) { System.out.println("From Thread C: k = " + k); } System.out.println("Exit from C"); } } // The scheduler may not always honor thread priorities, // especially in situations where there are many threads competing for CPU time.</pre>
--	--

Output:

From Thread C: k = 1	From Thread B: j = 3
From Thread C: k = 2	From Thread B: j = 4
From Thread C: k = 3	From Thread A: i = 3
From Thread B: j = 1	Exit from C
From Thread A: i = 1	From Thread B: j = 5
From Thread C: k = 4	From Thread A: i = 4
From Thread B: j = 2	Exit from B
From Thread A: i = 2	From Thread A: i = 5
From Thread C: k = 5	Exit from A

Experiment Number: 09

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

Theory:

In Java, a client-server program is a network-based application where one program, called the client, sends requests to another program, called the server, over a network. The server responds to these requests by providing some service or data.

Here's a simple explanation of how client-server programs work in Java:

Server Program: The server program listens for incoming connections from clients and provides some service or data in response to client requests. It typically runs continuously, waiting for incoming requests. Server programs are typically multithreaded to handle multiple client connections simultaneously.

Client Program: The client program initiates a connection to the server, sends requests for services or data, and waits for responses from the server. Clients can be standalone applications or parts of larger systems.

The server listens on port 4999 for incoming connections using `ServerSocket`.

When a client connects, the server creates a new `ClientHandler` thread to handle communication with that client.

The client connects to the server using `Socket`, sends a message ("Hello, Server!"), and waits for a response.

The server receives the message, prints it to the console, and sends a response back to the client.

The client receives the response and prints it to the console.

This is a basic example demonstrating the communication between a single client and server. In practice, you would typically extend this to

Source Code:

Client	Server
<pre>import java.io.BufferedReader; import java.io.IOException; import java.io.InputStreamReader; import java.io.PrintWriter; import java.net.Socket; public class ClientModel { public static void main(String[] args) { try { Socket soc= new Socket("localhost",4999); String str="hello guys"; PrintWriter output=new PrintWriter(soc.getOutputStream()); output.println(str); output.flush(); BufferedReader in= new BufferedReader(new InputStreamReader(soc.getInputStream())); String strinput=in.readLine(); System.out.println("Server Sent :"+strinput); } catch (IOException e) { // TODO Auto- generated catch block e.printStackTrace(); } } }</pre>	<pre>import java.io.BufferedReader; import java.io.IOException; import java.io.InputStreamReader; import java.io.PrintWriter; import java.net.ServerSocket; import java.net.Socket; public class ServerModel { public static void main(String[] args) { // TODO Auto-generated method stub try { System.out.println("Waiting for the clients..."); ServerSocket ss= new ServerSocket(4999); Socket soc= ss.accept(); System.out.println("Client Connected..."); BufferedReader in= new BufferedReader(new InputStreamReader(soc.getInputStream())); String str=in.readLine(); System.out.println("Client Sent :"+str); PrintWriter out= new PrintWriter(soc.getOutputStream(),true); out.println("I got your message"); out.flush(); } catch (IOException e) { // TODO Auto-generated catch block e.printStackTrace(); } } }</pre>

Output:

Server:

Waiting for the clients....

Client connected...

Client Sent : hello guys

Client :

Server sent: I got Your messa