**JAVA – LAB**

**ETCS-357**



**NAME:** SHIVAM SINGH

**ENROLLMENT NO:** 07313302720

**CLASS & SEC:** CSE-5A

**GROUP:** 2

**SUBMITTED TO: RASHMI TIWARI**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **INDEX** | | | | |
| **S.NO** | **AIM** | **DATE OF EXPERIMENT** | **DATE OF**  **SUBMISSION** | **REMARKS** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Experiment-1**

**Aim 🡺** Create a java program to implement stack and queue concept

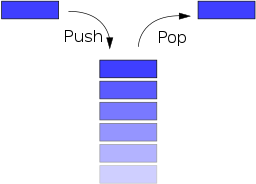
**Software Used 🡺** JetBrains IDE

**Theory 🡺**

**STACK 🡺** A stack is a linear data structure that follows the LIFO (Last–In, First–Out) principle. That means the objects can be inserted or removed only at one end of it, also called a top.

The stack supports the following operations:

* push inserts an item at the top of the stack (i.e., above its current top element).
* pop removes the object at the top of the stack and returns that object from the function. The stack size will be decremented by one



* isEmpty tests if the stack is empty or not.
* isFull tests if the stack is full or not.
* peek returns the object at the top of the stack without removing it from the stack or modifying the stack in any way.
* size returns the total number of elements present in the stack

**QUEUE 🡺** A queue is a linear data structure that follows the FIFO (First–In, First–Out) principle. That means the object inserted first will be the first one out, followed by the object inserted next.

The queue supports the following core operations:

1. Enqueue: Inserts an item at the rear of the queue.
2. Dequeue: Removes the object from the front of the queue and returns it, thereby decrementing queue size by one.
3. Peek: Returns the object at the front of the queue without removing it.
4. IsEmpty: Tests if the queue is empty or not.
5. Size: Returns the total number of elements present in the queue.

**Code 🡺**

**//**Stack Implementation

public class stack {

    private int[] arr;

    private int capacity;

    private int top;

    stack(int size) {

         arr = new int[size];

         capacity = size;

         top = -1;

    }

    public void push(int x) {

        if(isFull()) {

            System.out.println("Overflow\nExiting Program\n");

            System.exit(-1);

        }

        System.out.println("Inserting element "+ x);

        arr[++top] = x;

    }

    public int pop() {

        if(isEmpty()) {

            System.out.println("Underflow\nExiting Program\n");

            System.exit(-1);

        }

        System.out.println("Removing the element " + arr[top]);

        return arr[top--];

    }

    public int peek() {

        if(!isEmpty()) {

            return arr[top];

        }else {

            System.out.println("Stack is Empty\n");

            System.exit(-1);

        }

        return -1;

    }

    public int size() {

        return top+1;

    }

    public boolean isEmpty() {

        return top == -1;

    }

    public boolean isFull() {

        return top == capacity - 1;

    }

}

//Queue Implementation

public class queue {

    private int[] qarr;

    private int front;

    private int rear;

    private int capacity;

    private int count;

    queue(int size) {

        qarr = new int[size];

        front = 0;

        rear = -1;

        capacity = size;

        count = 0;

    }

    public int size() {

        return count;

    }

    public boolean isEmpty() {

        return count == 0;

    }

    public boolean isFull() {

        return count == capacity;

    }

    public void enqueue(int x) {

        if(isFull()) {

            System.out.println("Overflow\nExiting Program\n");

            System.exit(-1);

        }

        System.out.println("Inserting " + x);

        rear = (rear+1)%capacity;

        qarr[rear] = x;

        count++;

    }

    public int dequeue() {

        if(isEmpty()) {

            System.out.println("Underflow\nExiting Program\n");

            System.exit(-1);

        }

        System.out.println("Removing Element " + qarr[front]);

        int x = qarr[front];

        front = (front+1)%capacity;

        count--;

        return x;

    }

    public int qfront() {

        if(!isEmpty()){

            return qarr[front];

        }

        return -1;

    }

    public int qback() {

        if(!isEmpty()){

            return qarr[rear];

        }

        return -1;

    }

}

//Main Code

public class ExpOne {

    public static void main(String[] args) {

        System.out.println("Shivam Singh\n07313302720\nCSE-5A\n");

        //-----------------------Stack Implementation------------------------------------//

        System.out.println("\nStack Implementation\n");

        stack s1 = new stack(3);

        s1.push(1);

        s1.push(2);

        s1.pop();

        s1.pop();

        s1.push(3);

        System.out.println("The top element is " + s1.peek());

        System.out.println("The stack size is " + s1.size());

        //-----------------------Stack Implementation------------------------------------//

        //-----------------------Queue Implementation------------------------------------//

        System.out.println("\nQueue Implementation\n");

        queue q = new queue(5);

        q.enqueue(1);

        q.enqueue(2);

        q.enqueue(3);

        System.out.println("The front element is " + q.qfront());

        q.dequeue();

        System.out.println("The rear element is " + q.qback());

        System.out.println("The queue size is " + q.size());

        q.dequeue();

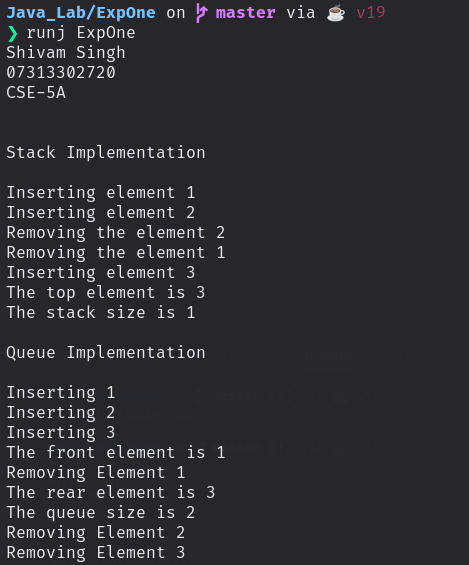
        q.dequeue();

        //-----------------------Queue Implementation------------------------------------//

    }

}

//Output



**Experiment-2**

**Aim 🡺 Write a program to implement Method Overloading**

**Software Used 🡺** JetBrains IDE

**Theory 🡺**

**Compile-time Polymorphism**

Compile-time polymorphism is also known as static polymorphism or early binding. Compile-time polymorphism is a polymorphism that is resolved during the compilation process. Overloading of methods is called through the reference variable of a class. Compile-time polymorphism is achieved by **method overloading**and **operator overloading.**

**1. Method overloading**

We can have one or more methods with the same name that are solely distinguishable by argument numbers, type, or order.

Method Overloading occurs when a class has many methods with the same name but different parameters. Two or more methods may have the same name if they have other numbers of parameters, different data types, or different numbers of parameters and different data types.

**Example:**

void sum() { ... }

void sum(int num1 ) { ... }

void sum(float num1) { ... }

void sum(int num1 , float num2 ) { ... }

**(a). Method overloading by changing the number of parameters**

 In this type, Method overloading is done by overloading methods in the function call with a varied number of parameters

**Example:**

show( char a )

show( char a ,char b )

#### (b). Method overloading by changing Datatype of parameter

In this type, Method overloading is done by overloading methods in the function call with different types of parameters

**Example:**

show( float a float b)

show( int a, int b )

#### (c). By changing the sequence of parameters

In this type, overloading is dependent on the sequence of the parameters

**Example:**

show( int a, float b )

show( float a, int b )

Here in this example, The parameters int and float are used in the first declaration. The parameters are int and float in the second declaration, but their order in the parameter list is different.

**Code 🡺**

class ExpTwo {

    public int sum(int a, int b) {

        System.out.println("Method with two arguments\n");

        return a+b;

    }

    public int sum(int a, int b, int c) {

        System.out.println("Method with three arguments\n");

        return a+b+c;

    }

    public float sum(float a, float b, float c, float d) {

        System.out.println("Method with four arguments\n");

        return a+b+c+d;

    }

    public static void main(String[] args) {

        System.out.println("Shivam Singh\n07313302720\nCSE-5A\n");

        ExpTwo obj1 = new ExpTwo();

        System.out.println(obj1.sum(5,2));

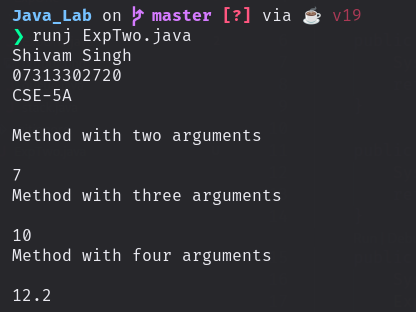
        System.out.println(obj1.sum(5,2, 3));

        System.out.println(obj1.sum(5.1f,2.3f, 3.3f, 1.5f));

    }

}

//**Output**

****

**EXPERIMENT-4**

**AIM 🡺** Develop an analog clock using applet.

**SOFTWARE USED 🡺** JetBrains IDE

**THEORY 🡺**

# Java Applet

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

### Advantage of Applet

There are many advantages of applet. They are as follows:

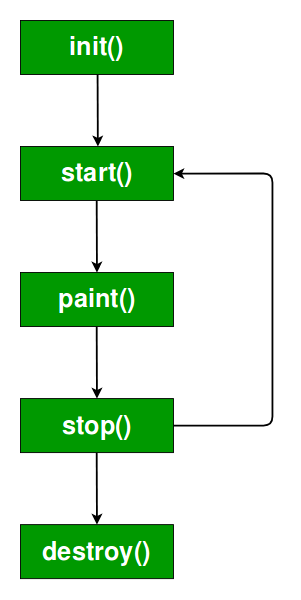
* It works at client side so less response time.
* Secured
* It can be executed by browsers running under many plateform, including Linux, Windows, MacOs etc.

### Drawback of Applet

* Plugin is required at client browser to execute applet.

Lifecycle of Java Applet

1. Applet is initialized.
2. Applet is started.
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.



**CODE 🡺**

// Java program to illustrate

// analog clock using Applets

import java.applet.Applet;

import java.awt.\*;

import java.util.\*;

public class analogClock extends Applet {

    @Override

    public void init()

    {

        // Applet window size & color

        this.setSize(new Dimension(800, 400));

        setBackground(new Color(50, 50, 50));

        new Thread() {

            @Override

            public void run()

            {

                while (true) {

                    repaint();

                    delayAnimation();

                }

            }

        }.start();

    }

    // Animating the applet

    private void delayAnimation()

    {

        try {

            // Animation delay is 1000 milliseconds

            Thread.sleep(1000);

        }

        catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

    // Paint the applet

    @Override

    public void paint(Graphics g)

    {

        // Get the system time

        Calendar time = Calendar.getInstance();

        int hour = time.get(Calendar.HOUR\_OF\_DAY);

        int minute = time.get(Calendar.MINUTE);

        int second = time.get(Calendar.SECOND);

        // 12 hour format

        if (hour > 12) {

            hour -= 12;

        }

        //Personal Info

        g.setColor(Color.white);

        g.drawString("Shivam Singh",200,200);

        g.drawString("CSE-5A",200,210);

        g.drawString("07313302720",200,220);

        // Draw clock body center at (400, 200)

        g.setColor(Color.white);

        g.fillOval(300, 100, 200, 200);

        // Labeling

        g.setColor(Color.black);

        g.drawString("12", 390, 120);

        g.drawString("9", 310, 200);

        g.drawString("6", 400, 290);

        g.drawString("3", 480, 200);

        // Declaring variables to be used

        double angle;

        int x, y;

        // Second hand's angle in Radian

        angle = Math.toRadians((15 - second) \* 6);

        // Position of the second hand

        // with length 100 unit

        x = (int)(Math.cos(angle) \* 100);

        y = (int)(Math.sin(angle) \* 100);

        // Red color second hand

        g.setColor(Color.red);

        g.drawLine(400, 200, 400 + x, 200 - y);

        // Minute hand's angle in Radian

        angle = Math.toRadians((15 - minute) \* 6);

        // Position of the minute hand

        // with length 80 unit

        x = (int)(Math.cos(angle) \* 80);

        y = (int)(Math.sin(angle) \* 80);

        // blue color Minute hand

        g.setColor(Color.blue);

        g.drawLine(400, 200, 400 + x, 200 - y);

        // Hour hand's angle in Radian

        angle = Math.toRadians((15 - (hour \* 5)) \* 6);

        // Position of the hour hand

        // with length 50 unit

        x = (int)(Math.cos(angle) \* 50);

        y = (int)(Math.sin(angle) \* 50);

        // Black color hour hand

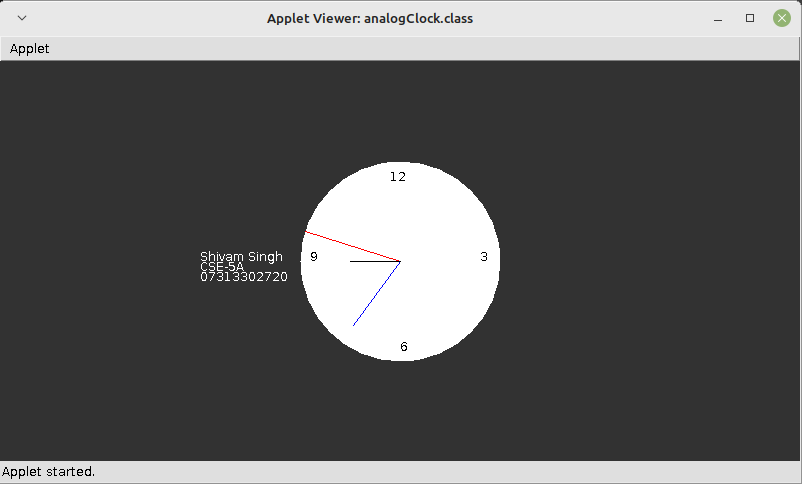
        g.setColor(Color.black);

        g.drawLine(400, 200, 400 + x, 200 - y);

    }

}

**//OUTPUT**



**EXPERIMENT-4**

**AIM 🡺** Write a program to access super class methods and instance variables without super keyword in java

**SOFTWARE USED 🡺** JetBrains IDE

**THEORY 🡺**

**SUPER CLASS :**The parent class from which many subclasses can be created. All the subclasses have all the attributes and properties that have parent class.

**INHERITANCE** is the mechanism of object-oriented language by which any class can (child class) inherit other class all the properties and behaviour of the parent class.

**PARENT CLASS:** FRUIT

**CHILD CLASS:** APPLE

* **There are two methods to call the instance variables and methods of the superclass (parent class) in the child class.**

1. **First Method:**super keyword is one of the reserved words in java. Super refers to an object of the parent class.

* **Uses:**
* We can invoke the overridden method of the parent class with the help of the super keyword.
* super() is used for executing the constructor of the parent class and should be used in the first line in the derived class constructor.

1. **Second Method:**Without using the keyword super keyword after inheritance all the methods and instance variables of the parent class is inherited by the child class. So we can direct them in the child class.

**Super Class:** parent class

**Main Class:** Derived class

**Code 🡺**

class Main extends SuperClass {

    void hello() {

        System.out.println("This is the main class");

    }

    public static void main(String[] args) {

        System.out.println("Shivam Singh\n07313302720\nCSE-5A\n");

        //calling the constructor

        Main ob = new Main();

        //calling the inherited name class

        ob.name();

    }

}

class SuperClass {

    SuperClass() {

        System.out.println("This is the constructor");

    }

    void name() {

        System.out.println("hello world");

    }

}

**//Output**

