NAMES: MANDERA THEOGENE

REGNO: SCT221-0897/2021

JAVA (OOP) ASSIGNMENT

i. **What is the Object Modeling Techniques (OMT).**

- The object-modeling technique is an object modeling approach for software modeling and designing

ii. **Compare object oriented analysis and design (OOAD) and object analysis and design(OOP).**

- Object-Oriented Analysis and Design (OOAD) is a broader concept that encompasses the entire process of analyzing and designing a system using object-oriented principles while Object-Oriented Programming (OOP) is a specific programming paradigm that is based on the principles of encapsulation, inheritance, and polymorphism.

iii. **Discuss Main goals of UML.**

- The main goal of UML is to define a standard way to visualize the way a system has been designed

- it helps engineers understand the system and identify potential issues.

iv. **DESCRIBE three advantages of using object oriented to develop an information system**.

- Inheritance and Code Reusability: Inheritance is a fundamental concept in OOP that allows a new class or subclass to inherit attributes and behaviors from an existing class.
- Encapsulation and Information Hiding**:** OOP supports the concept of encapsulation, where an object encapsulates its internal state and exposes only a well-defined interface.
- Modularity and Reusability**:** OOP promotes the modular design of software systems, breaking them down into smaller, manageable units called objects.

v. **briefly explain the following terms as used in object oriented programming. Write a sample java code to illustrate the implementation of the term.**

- **Constructor method**

a constructor is a special method that is automatically called when an object is instantiated or created.

eg: consider the code in the following example that shows the constructor method

```
 public class Person {
// Attributes or properties of the Person class
String name;
int age;
// Constructor method for initializing a Person object
public Person(String name, int age) {
   this.name = name;
   this.age = age;
}
// Display method to print information about the Person
public void displayInfo() {
   System.out.println("Name: " + name);
   System.out.println("Age: " + age);
}
public static void main(String[] args) {
   // Creating an instance of the Person class and calling the constructor
   Person person1 = new Person("John Doe", 25);
   // Accessing and displaying the attributes of the created Person object
   person1.displayInfo();
}
}
```

- **objec**t
- an object is an instance of a class.

eg: consider the code in the following example that shows the object method

```
public class Rectangle {
   // Attributes or properties of the Rectangle class
   double length;
   double width;

   // Constructor method for initializing a Rectangle object
   public Rectangle(double length, double width) {
      this.length = length;
```

```
                this.width = width;
        }

            // Method to calculate and return the area of the rectangle
    public double calculateArea() {
        return length * width;
    }

    public static void main(String[] args) {
        // Creating an instance of the Rectangle class (object)
        Rectangle myRectangle = new Rectangle(5.0, 8.0);

        // Accessing the attributes of the created Rectangle object
        System.out.println("Length: " + myRectangle.length);
        System.out.println("Width: " + myRectangle.width);

        // Calculating and displaying the area of the Rectangle
        double area = myRectangle.calculateArea();
        System.out.println("Area: " + area);
    }
}
```

eg: consider the code in the following example that shows the object method

- **interface**

an interface is a collection of abstract methods that define a contract for classes that implement the interface.

eg: consider the code in the following example that shows the interface method

```
interface Shape {
    // Abstract method for calculating the area
    double calculateArea();

    // Abstract method for displaying information about the shape
    void displayInfo();
}

// Implement the Shape interface in a class named Circle
class Circle implements Shape {
    double radius;

    public Circle(double radius) {
        this.radius = radius;
    }
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
    public void displayInfo() {
        System.out.println("Circle - Radius: " + radius);
    }
}
// Implement the Shape interface in a class named Rectangle
class Rectangle implements Shape {
    double length;
    double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
    public double calculateArea() {
        return length * width;
    }
    public void displayInfo() {
        System.out.println("Rectangle - Length: " + length + ", Width: " + width);
    }
}
public class InterfaceExample {
    public static void main(String[] args) {
        // Create instances of Circle and Rectangle
        Circle myCircle = new Circle(5.0);
        Rectangle myRectangle = new Rectangle(4.0, 6.0);
        // Use the interface methods to calculate area and display information
        myCircle.displayInfo();
        System.out.println("Area of Circle: " + myCircle.calculateArea());

        myRectangle.displayInfo();
        System.out.println("Area of Rectangle: " + myRectangle.calculateArea());
    }
}
```

- **polymorphism**

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common base class or interface.

eg: consider the code in the following example that shows the polymorphism method

```
class Shape {
    // Method to calculate the area (to be overridden by subclasses)
```

```
  double calculateArea() {
    return 0.0;
  }
}

// Subclass Circle extends Shape
class Circle extends Shape {
  double radius;

  public Circle(double radius) {
    this.radius = radius;
  }
  // Override the calculateArea method for Circle
  double calculateArea() {
    return Math.PI * radius * radius;
  }
}
// Subclass Rectangle extends Shape
class Rectangle extends Shape {
  double length;
  double width;
  public Rectangle(double length, double width) {
    this.length = length;
    this.width = width;
  }
  // Override the calculateArea method for Rectangle
  double calculateArea() {
    return length * width;
  }
```

```
}

public class PolymorphismExample {
   // Method to calculate and display the area of any Shape
   public static void printArea(Shape shape) {
      System.out.println("Area: " + shape.calculateArea());
   }
   public static void main(String[] args) {
      // Create instances of Circle and Rectangle
      Circle myCircle = new Circle(5.0);
      Rectangle myRectangle = new Rectangle(4.0, 6.0);

      // Use polymorphism to calculate and display areas
      printArea(myCircle);
      printArea(myRectangle);
   }
}
```

- **class**
- a class is a blueprint or template for creating objects.

eg: consider the code in the following example that shows the class method

```
class Car {
   // Attributes or properties of the Car class
   String make;
   String model;
   int year;

   // Constructor method for initializing a Car object
   public Car(String make, String model, int year) {
```

```
this.make = make;
        this.model = model;
        this.year = year;
    }

    // Method to display information about the car
    public void displayInfo() {
        System.out.println("Make: " + make);
        System.out.println("Model: " + model);
        System.out.println("Year: " + year);
    }
}
public class ClassExample {
    public static void main(String[] args) {
        // Create an instance of the Car class
        Car myCar = new Car("Toyota", "Camry", 2022);

        // Accessing and displaying information about the Car object
        myCar.displayInfo();
    }
}
```

vi.EXPLAIN the three types of associations (relationships) between objects in object oriented.

- Association: is a generic term that represents a bi-directional relationship between two or more classes
- Aggregation: is a specific type of association where one class represents a whole and another class represents its part.
- Composition: is a stronger form of aggregation where the parts are strongly tied to the whole. It represents a "owns" relationship, indicating that the lifetime of the part is dependent on the lifetime of the whole.

Vii. What do you mean by class diagram? Where it is used and also discuss the steps to draw the class diagram with any one example.

- A class diagram is a type of UML (Unified Modeling Language) diagram that illustrates the structure and relationships of classes within a system.

Class diagrams are used to:

- Model System Structure: Class diagrams depict the static structure of a system by illustrating the classes and their relationships.
- Communication and Documentation: Class diagrams serve as a communication tool among stakeholders, including developers, designers, and clients.
- Code Generation: Class diagrams can be used to generate code, especially in environments that support Model-Driven Architecture (MDA) or code generation tools.

Steps to Draw a Class Diagram:

Lets Consider a simple hospital management system with classes such as

- Patient, Doctor, Appointment, and Department

1. Identify Classes:
    - Patient
    - Doctor
    - Appointment
    - Department
2. Define Class Attributes and Methods:
    - Patient class attributes: patientId, name, birthDate, contactNumber.
    - Doctor class attributes: doctorId, name, specialization, contactNumber.
    - Appointment class attributes: appointmentId, patient (reference to Patient), doctor (reference to Doctor), appointmentDate.
    - Department class attributes: departmentId, name, description.

3. Define Associations:

- Patient has an association with Appointment (one-to-many).
- Doctor has an association with Appointment (one-to-many).
- Department has an association with Doctor (one-to-many).

4. Specify Multiplicity:

- Patient has many appointments (0..*).
- Doctor has many appointments (0..*).
- Department has many doctors (0..*).

5. Add Class Relationships:

- Draw lines between classes to represent associations.
- Add arrows to indicate the direction of associations.
- Label the associations with multiplicity.
- Add a relationship between Doctor and Department to represent the department each doctor belongs to.

6. Refine and Review:

- Add additional details such as visibility modifiers (public, private) and methods if needed.

7. Document the Diagram:

- Add a title, summary, and any additional notes that clarify the purpose and elements of the diagram.

8. Finalize and Share:

- Finalize the diagram, ensuring it is clear and easy to understand.

- **Create a new class called CalculateG.**
  Copy and paste the following initial version of the code. Note variables declaration and the types.
- SOLUTION: modified program

```
public class CalculateG {

  public static double gravity = -9.81; // Earth's gravity in m/s^2

  public static double fallingTime = 30;

  public static double initialVelocity = 0.0;

  public static double finalVelocity;

  public static double initialPosition = 0.0;

  public static double finalPosition;


  public static double multi(double a, double b) {

    return a * b;

  }


  public static double square(double a) {

    return a * a;

  }


  public static double sum(double a, double b) {

    return a + b;

  }


  public static void main(String[] args) {
        // Calculate position and velocity
        finalPosition = 0.5 * multi(gravity, square(fallingTime)) + multi(initialVelocity, fallingTime) +
initialPosition;
        finalVelocity = multi(gravity, fallingTime) + initialVelocity;


        // Output position
        System.out.println("The object's position after " + fallingTime + " seconds is " + finalPosition + "
m.");


        // Output velocity
        System.out.println("The object's velocity after " + fallingTime + " seconds is " + finalVelocity + "
m/s.");
    }
}
```

6**. Create methods for multiplication, powering to square, summation and printing out a result in CalculateG class.**

```
public class CalculateG {
    public static double gravity = -9.81; // Earth's gravity in m/s^2
    public static double fallingTime = 30;
    public static double initialVelocity = 0.0;
    public static double finalVelocity;
    public static double initialPosition = 0.0;
    public static double finalPosition;


    public static double multiply(double a, double b) {
        return a * b;
    }


    public static double square(double a) {
        return a * a;
    }

    public static double sum(double a, double b) {
        return a + b;
    }

    public static void printResult(double result) {
        System.out.println("The result is: " + result);
    }

    public static void calculatePositionAndVelocity() {
        // Calculate position and velocity
        finalPosition = 0.5 * multiply(gravity, square(fallingTime)) + multiply(initialVelocity, fallingTime) +
initialPosition;
        finalVelocity = multiply(gravity, fallingTime) + initialVelocity;

        // Output position
        System.out.println("The object's position after " + fallingTime + " seconds is " + finalPosition + "
m.");

        // Output velocity
        System.out.println("The object's velocity after " + fallingTime + " seconds is " + finalVelocity + "
m/s.");

        // Use the printResult method
        printResult(finalPosition + finalVelocity);
    }

    public static void main(String[] args) {
        // Call the method to calculate position and velocity

        calculatePositionAndVelocity();

    }

}
```

Part B

1. Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:
1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

By considering the terms in the Fibonacci sequence whose values do not exceed four million, write a Java method to find the sum of all the even- valued terms.

```
public class FibonacciSum {

    public static void main(String[] args) {

        int limit = 4000000;

        long sum = calculateEvenFibonacciSum(limit);


        System.out.println("The sum of even-valued terms in the Fibonacci sequence not exceeding " + limit
+ " is: " + sum);

    }


    public static long calculateEvenFibonacciSum(int limit) {

        long sum = 0;

        int fib1 = 1;

        int fib2 = 2;


        while (fib2 <= limit) {

            if (fib2 % 2 == 0) {

                sum += fib2;

            }
            int nextFib = fib1 + fib2;

            fib1 = fib2;

            fib2 = nextFib;

        }


        return sum;

    }

}
```

- 2.A palindrome number is a number that remain the same when read from behind or front (a number that is equal to reverse of number) for example, 353 is palindrome because reverse of 353 is 353 (you see the number remains the same). But a number like 591 is not palindrome because reverse of 591 is 195 which is not equal to 591. Write Java program to check if a number entered by the user is palindrome or not. You should provide the user with a GUI interface to enterface.

- **Solution: IMPLEMENTATION CODE**

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

        public class PalindromeCheckerGUI extends JFrame {

      private JTextField inputField;

          private JLabel resultLabel;


    public PalindromeCheckerGUI() {

        setTitle("Palindrome Checker");

        setSize(300, 150);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setLocationRelativeTo(null);


        createUI();
        setVisible(true);
    }

    private void createUI() {
        JPanel panel = new JPanel();
        panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));

        inputField = new JTextField(10);
        JButton checkButton = new JButton("Check Palindrome");
        resultLabel = new JLabel();
        checkButton.addActionListener(new ActionListener()
            public void actionPerformed(ActionEvent e) {
                checkPalindrome();
            }
        });
        panel.add(new JLabel("Enter a number:"));
        panel.add(inputField);
        panel.add(checkButton);
        panel.add(resultLabel);
        add(panel);
    }
    private void checkPalindrome() {
        String inputText = inputField.getText().trim();
        if (isPalindrome(inputText)) {
            resultLabel.setText("Palindrome: Yes");
        } else {
            resultLabel.setText("Palindrome: No");
        }
    }
```

```
  private boolean isPalindrome(String str) {
    String reversed = new StringBuilder(str).reverse().toString();
    return str.equals(reversed);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new PalindromeCheckerGUI();
      }
    });
  }
}
```

- **Question three:**

Write a Java program that takes 15 values of type integer as inputs from user, store the values in an array.

Print the values stored in the array on screen.

Ask user to enter a number, check if that number (entered by user) is present in array or not. If it is present print, "the number found at index (index of the number) " and the text "number not found in this array"

Create another array, copy all the elements from the existing array to the new array but in reverse order. Now print the elements of the new array on the screen

Get the sum and product of all elements of your array. Print product and the sum each on its own line.

```
import java.util.Scanner;


public class ArrayOperations {
  public static void main(String[] args) {
    // Part a: Taking 15 values as inputs and storing them in an array
    int[] array = new int[15];
```

```
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter 15 integers:");
        for (int i = 0; i < 15; i++) {
            System.out.print("Enter value #" + (i + 1) + ": ");
            array[i] = scanner.nextInt();
        }
        // Part b: Printing values stored in the array
        System.out.println("\nValues stored in the array:");
        printArray(array);

        // Part c: Checking if a number is present in the array
        System.out.print("\nEnter a number to check: ");
        int numberToCheck = scanner.nextInt();
        int index = findIndex(array, numberToCheck);
        if (index != -1) {
            System.out.println("The number found at index " + index);
        } else {
            System.out.println("Number not found in this array");
        }
        // Part d: Creating a new array with elements in reverse order and printing it
        int[] reversedArray = reverseArray(array);
        System.out.println("\nElements of the new array in reverse order:");
        printArray(reversedArray);
        // Part e: Calculating and printing the sum and product of array elements
        int sum = calculateSum(array);
        int product = calculateProduct(array);
        System.out.println("\nSum of array elements: " + sum);
        System.out.println("Product of array elements: " + product);
    }
```

```
// Method to print the elements of an array
private static void printArray(int[] arr) {
    for (int value : arr) {
        System.out.print(value + " ");
    }
    System.out.println();
}
// Method to find the index of a number in an array
private static int findIndex(int[] arr, int number) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == number) {
            return i;
        }
    }
    return -1; // Return -1 if the number is not found
}
// Method to create a new array with elements in reverse order
private static int[] reverseArray(int[] arr) {
    int[] reversedArray = new int[arr.length];
    for (int i = 0; i < arr.length; i++) {
        reversedArray[i] = arr[arr.length - 1 - i];
    }
    return reversedArray;
}
// Method to calculate the sum of array elements
private static int calculateSum(int[] arr) {
    int sum = 0;
    for (int value : arr) {
            sum += value;
        }
        return sum;
    }
    // Method to calculate the product of array elements
    private static int calculateProduct(int[] arr) {
        int product = 1;
        for (int value : arr) {
            product *= value;
        }
        return product;
    }
}
```