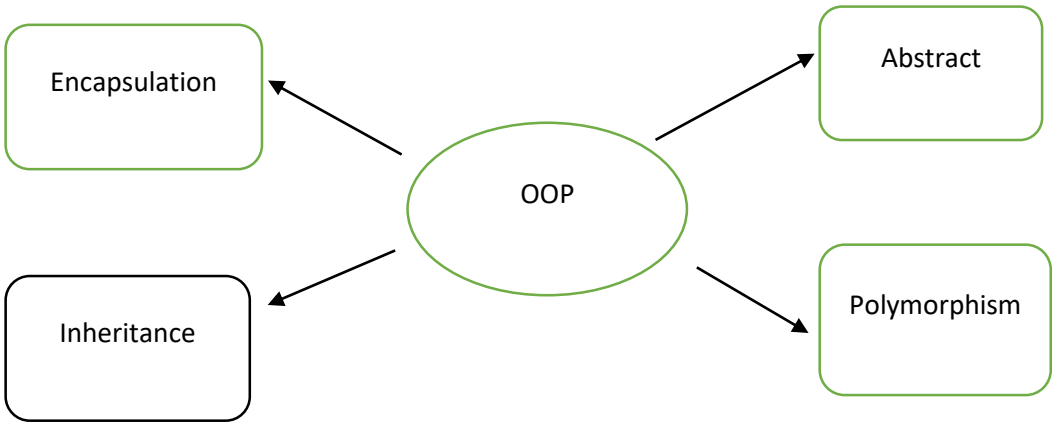
LUCY MWONGELI MUASA

REG NO: SCT121-0903/2022

DIT: OOP Assignment

Part A:

i. Using a well labeled diagram, explain the steps of creating a system using OOP principles.



Encapsulation: Hide the internal details of a class and expose only what is necessary. This is achieved through access modifiers (public, private, protected).

Inheritance: Use inheritance to model the "is-a" relationship between classes. Subclasses inherit attributes and behaviors from their superclass.

Polymorphism: Allow objects to take on multiple forms. This can be achieved through method overloading and overriding.

Abstraction: Abstract complex systems by simplifying them into manageable components. Focus on relevant details while hiding unnecessary complexity

ii. What is the Object Modeling Techniques

Is an object modelling approach for software modeling and designing

iii. Compare (OOAD) and (OOP).

OOAD involves analyzing and designing a system from an object-oriented perspective, considering both analysis and design phases.

OOP typically refers to the broader concept of programming using object-oriented principles without specifically emphasizing the analysis and design phases.

iv. Discuss main goals of UML

Provide users with a ready –to-use expressive visual modeling language so they can develop and exchange meaningful models.

Provide extensibility and specialization mechanism to extend the core concept.

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

Modularity: Object-oriented programming promotes modularity, which means breaking down a complex system into smaller, self-contained modules or objects

Reusability: One of the key advantages of object-oriented programming is the concept of inheritance and polymorphism. Inheritance allows objects to inherit properties and behaviors from other objects, promoting code reuse.

Flexibility and Extensibility: Object-oriented programming provides flexibility and extensibility in designing and implementing information systems. With inheritance and polymorphism, you can easily extend existing classes by adding new properties and behaviors.

vi. Briefly explain the following terms as used in OOP. Write a sample java code to illustrate the implementation of each concept.
    a. Constructor

In object-oriented programming, a constructor is a special method that is used to initialize an object when it is created. It is typically used to set the initial state of an object by assigning values to its instance variables

java

```
public class Car {
    private String brand;
    private String color;
    private int year;

    // Constructor with parameters
    public Car(String brand, String color, int year) {
        this.brand = brand;
```

```
        this.color = color;
        this.year = year;
    }

    // Getter methods

    public String getBrand() {
        return brand;
    }

    public String getColor() {
        return color;
    }

    public int getYear() {
        return year;
    }

    public static void main(String[] args) {
        // Creating an object of the Car class
        Car myCar = new Car("Toyota", "Red", 2022);

        // Accessing the object's properties using getter methods
        System.out.println("Brand: " + myCar.getBrand());
        System.out.println("Color: " + myCar.getColor());
        System.out.println("Year: " + myCar.getYear());
    }
}
```

b. object

In object-oriented programming, an object is an instance of a class. It represents a specific entity or thing that has state (data) and behavior (methods).

java

```
public class Person {
    private String name;
    private int age;

    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Method to display the person's information
    public void displayInfo() {
```

```
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }

    public static void main(String[] args) {
        // Creating an object of the Person class
        Person person1 = new Person("John", 25);

        // Calling the displayInfo method on the object
        person1.displayInfo();
    }
}
```

c. Destructor

In object-oriented programming, a destructor is a special method that is called when an object is about to be destroyed or deallocated. It is used to release any resources or perform any cleanup operations before the object is removed from memory.

java

```
public class MyClass {
    public MyClass() {
        System.out.println("Constructor called");
    }

    @Override
    protected void finalize() throws Throwable {
        System.out.println("Destructor called");
        super.finalize();
    }

    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj = null; // Set the reference to null to make the object eligible for garbage collection
        System.gc(); // Request the garbage collector to run
    }
}
```

d. polymorphism

In object-oriented programming, polymorphism refers to the ability of an object to take on many forms. It allows objects of different classes to be treated as objects of a common superclass.

java

```
class Animal {
    public void makeSound() {
        System.out.println("The animal makes a sound");
    }
}
```

```
class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("The dog barks");
    }
}

class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("The cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal1 = new Dog();
        Animal animal2 = new Cat();

        animal1.makeSound(); // Output: The dog barks
        animal2.makeSound(); // Output: The cat meows
    }
}
```

e. class

In object-oriented programming, a class is a blueprint or a template for creating objects. It defines the properties (attributes) and behaviors (methods) that an object of that class will have.

java

```
class Car {
    // Attributes
    String brand;
    String color;
    int year;

    // Constructor
    public Car(String brand, String color, int year) {
        this.brand = brand;
        this.color = color;
        this.year = year;
    }

    // Method
    public void startEngine() {
        System.out.println("The " + brand + " car's engine is starting...");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Creating an object of the Car class
        Car myCar = new Car("Toyota", "Red", 2022);

        // Accessing attributes
        System.out.println("Brand: " + myCar.brand);
        System.out.println("Color: " + myCar.color);
        System.out.println("Year: " + myCar.year);

        // Calling a method
        myCar.startEngine(); // Output: The Toyota car's engine is starting...
    }
}
```

f. Inheritance

Inheritance is a fundamental concept in object-oriented programming that allows a class (known as the child or subclass) to inherit properties and behaviors from another class (known as the parent or superclass).

java

```
// Parent class
class Animal {
    // Method
    public void eat() {
        System.out.println("The animal is eating.");
    }
}

// Child class inheriting from Animal
class Dog extends Animal {
    // Method overriding
    @Override
    public void eat() {
        System.out.println("The dog is eating.");
    }

    // Additional method
    public void bark() {
        System.out.println("The dog is barking.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object of the Dog class
        Dog myDog = new Dog();
```

```
        // Calling methods
        myDog.eat(); // Output: The dog is eating.
        myDog.bark(); // Output: The dog is barking.
    }
}
```

vii. ***EXPLAIN*** the three types of associations (relationships) between objects in object orient

Aggregation: Aggregation represents a "has-a" relationship between objects, where one object is composed of or contains other objects. It is a form of association where the objects have a whole-part relationship.

Composition: Composition is a stronger form of aggregation, where the lifetime of the contained objects is dependent on the lifetime of the container object. It represents a "owns-a" relationship, where the container object is responsible for the creation and destruction of the contained objects.

Inheritance: Inheritance represents an "is-a" relationship between objects, where one object inherits the properties and behaviors of another object. It is a mechanism that allows objects to acquire the characteristics of a parent object, leading to code reuse and hierarchy.

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

Class diagram is a visual representation of class objects in a model system, categorized by a class.

1. Identify the classes: Determine the main classes in your system.
2. Identify the attributes: For each class, identify the attributes or properties that define its characteristics.
3. Identify the methods: Determine the methods or behaviors that the classes can perform.
4. Define relationships: Identify the relationships between the classes.
5. Draw the class diagram: Use a visual modeling tool or drawing software to create the class diagram.

viii. Given that you are creating area and perimeter calculator using C++, to computer area and perimeter of various shaped like Circles, Rectangle, Triangle and Square, use well written code to explain and implement the calculator using the following OOP concepts.
    a. Inheritance (Single inheritance, Multiple inheritance and Hierarchical inheritance)

1. Single Inheritance:
In single inheritance, a derived class inherits properties and behaviors from a single base class. Let's create an example using a Circle class as the base class and a Cylinder class as the derived class:

```
class Circle {
protected:
    double radius;

public:
```

```
    Circle(double r) {
        radius = r;
    }

    double getArea() {
        return 3.14159 * radius * radius;
    }

    double getPerimeter() {
        return 2 * 3.14159 * radius;
    }
};

class Cylinder : public Circle {
private:
    double height;

public:
    Cylinder(double r, double h) : Circle(r) {
        height = h;
    }

    double getVolume() {
        return getArea() * height;
    }
};
```

2. Multiple Inheritance:
In multiple inheritance, a derived class can inherit properties and behaviors from multiple base classes.

```
class Rectangle {
protected:
    double length;
    double width;

public:
    Rectangle(double l, double w) {
        length = l;
        width = w;
    }

    double getArea() {
        return length * width;
    }

    double getPerimeter() {
```

```
        return 2 * (length + width);
    }
};

class Triangle {
protected:
    double base;
    double height;

public:
    Triangle(double b, double h) {
        base = b;
        height = h;
    }

    double getArea() {
        return 0.5 * base * height;
    }

    double getPerimeter() {
        return base + 2 * sqrt((base / 2) * (base / 2) + height * height);
    }
};

class Square : public Rectangle, public
```

b. Friend functions [5 Marks]

Friend functions are functions that are not a member of a class but have access to the private and protected members of that class

```
class Circle {
private:
    double radius;

public:
    Circle(double r) {
        radius = r;
    }

    friend double calculateArea(const Circle& c);
    friend double calculatePerimeter(const Circle& c);
};

class Rectangle {
private:
    double length;
```

```
    double width;

public:
    Rectangle(double l, double w) {
        length = l;
        width = w;
    }

    friend double calculateArea(const Rectangle& r);
    friend double calculatePerimeter(const Rectangle& r);
};

class Triangle {
private:
    double base;
    double height;

public:
    Triangle(double b, double h) {
        base = b;
        height = h;
    }

    friend double calculateArea(const Triangle& t);
    friend double calculatePerimeter(const Triangle& t);
};

class Square {
private:
    double side;

public:
    Square(double s) {
        side = s;
    }

    friend double calculateArea(const Square& s);
    friend double calculatePerimeter(const Square& s);
};

double calculateArea(const Circle& c) {
    return 3.14159 * c.radius * c.radius;
}

double calculatePerimeter(const Circle& c) {
    return 2 * 3.14159 * c.radius;
```

```
}

double calculateArea(const Rectangle& r) {
   return r.length * r.width;
}

double calculatePerimeter(const Rectangle& r) {
   return 2 * (r.length + r.width);
}

double calculateArea(const Triangle& t) {
   return 0.5 * t.base * t.height;
}

double calculatePerimeter(const Triangle& t) {
   return t.base + 2 * sqrt((t.base / 2) * (t.base / 2) + t.height * t.height);
}

double calculateArea(const Square& s) {
   return s.side * s.side;
}

double calculatePerimeter(const Square& s) {
   return 4 * s.side;
}
```

c. Method overloading and method overriding

Method overloading allows us to define multiple methods with the same name but different parameters.

Method overriding, on the other hand, is used when we have a base class and a derived class.

```
class Shape {
public:
   virtual double calculateArea() = 0;
   virtual double calculatePerimeter() = 0;
};

class Circle : public Shape {
private:
   double radius;

public:
   Circle(double r) {
      radius = r;
   }
```

```
    double calculateArea() override {
        return 3.14159 * radius * radius;
    }

    double calculatePerimeter() override {
        return 2 * 3.14159 * radius;
    }
};

class Rectangle : public Shape {
private:
    double length;
    double width;

public:
    Rectangle(double l, double w) {
        length = l;
        width = w;
    }

    double calculateArea() override {
        return length * width;
    }

    double calculatePerimeter() override {
        return 2 * (length + width);
    }
};

class Triangle : public Shape {
private:
    double base;
    double height;

public:
    Triangle(double b, double h) {
        base = b;
        height = h;
    }

    double calculateArea() override {
        return 0.5 * base * height;
    }

    double calculatePerimeter() override {
        return base + 2 * sqrt((base / 2) * (base / 2) + height * height);
```

```
    }
};

class Square : public Shape {
private:
    double side;

public:
    Square(double s) {
        side = s;
    }

    double calculateArea() override {
        return side * side;
    }

    double calculatePerimeter() override {
        return 4 * side;
    }
};
```

d. Late binding and early binding

Late binding and early binding are related to polymorphism in C++. Early binding, also known as static binding, happens at compile time. It means that the function call is resolved at compile time based on the type of the pointer or reference

```
public:
    virtual double calculateArea() = 0;
    virtual double calculatePerimeter() = 0;
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r) {
        radius = r;
    }

    double calculateArea() override {
        return 3.14159 * radius * radius;
    }

    double calculatePerimeter() override {
        return 2 * 3.14159 * radius;
```

```
    }
};

class Rectangle : public Shape {
private:
    double length;
    double width;

public:
    Rectangle(double l, double w) {
        length = l;
        width = w;
    }

    double calculateArea() override {
        return length * width;
    }

    double calculatePerimeter() override {
        return 2 * (length + width);
    }
};

class Triangle : public Shape {
private:
    double base;
    double height;

public:
    Triangle(double b, double h) {
        base = b;
        height = h;
    }

    double calculateArea() override {
        return 0.5 * base * height;
    }

    double calculatePerimeter() override {
        return base + 2 * sqrt((base / 2) * (base / 2) + height * height);
    }
};

class Square : public Shape {
private:
    double side;
```

```
public:
   Square(double s) {
      side = s;
   }

   double calculateArea() override {
      return side * side;
   }

   double calculatePerimeter() override {
```

e. Abstract class and pure functions [6 Marks]

In C++, an abstract class is a class that cannot be instantiated. It serves as a base class for other classes and provides a common interface for its derived classes.

```
#include <iostream>
class Shape {
public:
   virtual double calculateArea() = 0;
   virtual double calculatePerimeter() = 0;
};

class Circle : public Shape {
private:
   double radius;

public:
   Circle(double r) {
      radius = r;
   }

   double calculateArea() override {
      return 3.14159 * radius * radius;
   }
double calculatePerimeter() override {
      return 2 * 3.14159 * radius;
   }
};

class Rectangle : public Shape {
private:
   double length;
   double width;
```

```
public:
    Rectangle(double l, double w) {
        length = l;
        width = w;
    }

    double calculateArea() override {
        return length * width;
    }

    double calculatePerimeter() override {
        return 2 * (length + width);
    }
};

class Triangle : public Shape {
private:
    double base;
    double height;

public:
    Triangle(double b, double h) {
        base = b;
        height = h;
    }

    double calculateArea() override {
        return 0.5 * base * height;
    }

    double calculatePerimeter() override {
        // Implement the perimeter calculation for a triangle
    }
};

class Square : public Shape {
private:
    double side;

public:
    Square(double s) {
        side = s;
    }

  double calculateArea() override {
```

```
        return side * side;
    }

    double calculatePerimeter() override {
        return 4 * side;
    }
};

int main() {
    // Create instances of different shapes
    Circle circle(5.0);
    Rectangle rectangle(4.0, 6.0);
    Triangle triangle(3.0, 4.0);
    Square square(5.0);

    // Calculate and display the area and perimeter of each shape
    std::cout << "Circle: Area = " << circle.calculateArea() << ", Per
```

ix. Using a program written in C++, differentiate between the following.

a. Function overloading and operator overloading

Function overloading refers to defining multiple functions with the same name but different parameter lists. This allows you to perform similar operations on different types of data or with different numbers of arguments.

```
int add(int a, int b) {
    return a + b;
}

float add(float a, float b) {
    return a + b;
}
```

Operator overloading, on the other hand, allows you to redefine the behavior of operators such as +, -, *, /, etc., for user-defined types. This means you can use operators to perform custom operations on objects of your own classes.

```
class MyString {
```

```
    string str;
public:
    MyString(string s) : str(s) {}

    MyString operator+(const MyString& other) {
        return MyString(str + other.str);
    }
};

int main() {
    MyString s1("Hello");
    MyString s2(" World");
    MyString s3 = s1 + s2; // Concatenates the two strings using the overloaded + operator
}
```

b. Pass by value and pass by reference

When you pass an argument by value, a copy of the value is made and passed to the function. Any modifications made to the parameter inside the function will not affect the original value outside of the function.

```
void increment(int num) {
    num++; // Increment the local copy of num
    cout << "Inside the function: " << num << endl;
}

int main() {
    int num = 5;
    increment(num); // Pass by value
    cout << "Outside the function: " << num << endl;
    return 0;
}
```

Output:
Inside the function: 6
Outside the function: 5

On the other hand, when you pass an argument by reference, you pass a reference to the original value, allowing you to modify the value directly

```
void increment(int& num) {
    num++; // Increment the original value of num
    cout << "Inside the function: " << num << endl;
}
int main() {
    int num = 5;
    increment(num); // Pass by reference
    cout << "Outside the function: " << num << endl;
    return 0;
```

```
}
```
Output:
Inside the function: 6
Outside the function: 6

c. Parameters and arguments

A parameter is a variable declared in the function's declaration or definition. It acts as a placeholder for the value that will be passed to the function when it is called. Parameters are used to define the type and name of the values that a function expects to receive.

```
void addNumbers(int a, int b) {
   int sum = a + b;
   cout << "The sum is: " << sum << endl;
}

int main() {
   int num1 = 5;
   int num2 = 3;
   addNumbers(num1, num2); // Calling the function with arguments
   return 0;
}
```

An argument, on the other hand, is the actual value that is passed to a function when it is called. It is the value that is assigned to the corresponding parameter inside the function.

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

```
#include <iostream>

#include <cmath>class CalculateG {

public:

   double multiply(double a, double b) {

      return a * b;

   }
```

```
   double powerSquare(double num) {
      return pow(num, 2);
   }

   double summation(double a, double b) {
      return a + b;
   }
void outline(double result) {
 std::cout << "The result is: " << result << std::endl;
   }

   int main() {
      double gravity = -9.81; // Earth's gravity in m/s^2
      double fallingTime = 30;
      double initialVelocity = 0.0;
      double finalVelocity;
      double initialPosition = 0.0;
      double finalPosition;
    // Calculate the final position and velocity
      finalPosition = 0.5 * gravity * pow(fallingTime, 2) + initialVelocity * fallingTime + initialPosition;
      finalVelocity = gravity * fallingTime + initialVelocity;
```

```
        // Output the position and velocity

        std::cout << "The object's position after " << fallingTime << " seconds is " << finalPosition << " m." << std::endl;

        std::cout << "The object's velocity after " << fallingTime << " seconds is " << finalVelocity << " m/s." << std::endl;



        return 0;

    }

};



int main() {

    CalculateG calculator;

    calculator.main();



    return 0;

}
```

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 C++ method to find the sum of all the even- valued terms.

```
#include <iostream>

using namespace std;
```

```
int main() {
    int firstTerm = 1;
    int secondTerm = 2;
    int sum = 2; // Initialize the sum with the second term (which is even)

    while (secondTerm <= 4000000) {
        int nextTerm = firstTerm + secondTerm;
        if (nextTerm % 2 == 0) {
            sum += nextTerm;
        }
        firstTerm = secondTerm;
        secondTerm = nextTerm;
    }

    cout << "The sum of all even-valued terms in the Fibonacci sequence up to four million is: " << sum << endl;

    return 0;
}
```

Question three: [15 marks]

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

a) Print the values stored in the array on screen.

```
#include <iostream>
using namespace std;

int main() {
   const int SIZE = 15;
   int values[SIZE];

   // Taking input from the user
   cout << "Please enter " << SIZE << " integer values:" << endl;
   for (int i = 0; i < SIZE; i++) {
      cout << "Value " << i + 1 << ": ";
      cin >> values[i];
   }

   // Printing the values stored in the array
   cout << "The values stored in the array are:" << endl;
   for (int i = 0; i < SIZE; i++) {
      cout << values[i] << " ";
   }
   cout << endl;

   return 0;
}
```

b) 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”

```
#include <iostream>

using namespace std;


int main() {

   const int SIZE = 15;

   int values[SIZE];


   // Taking input from the user

   cout << "Please enter " << SIZE << " integer values:" << endl;

   for (int i = 0; i < SIZE; i++) {

      cout << "Value " << i + 1 << ": ";

      cin >> values[i];

   }


   // Asking the user to enter a number

   int number;

   cout << "Enter a number to search in the array: ";

   cin >> number;


   // Checking if the number is present in the array

   bool found = false;
```

```
    int index;
    for (int i = 0; i < SIZE; i++) {
        if (values[i] == number) {
            found = true;
            index = i;
            break;
        }
    }

    // Printing the result
    if (found) {
        cout << "The number " << number << " found at index " << index << "." << endl;
    } else {
        cout << "The number " << number << " not found in this array." << endl;
    }

    return 0;
}
```

c)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

```
#include <iostream>
using namespace std;
```

```
int main() {
    const int SIZE = 15;
    int values[SIZE];

    // Taking input from the user
    cout << "Please enter " << SIZE << " integer values:" << endl;
    for (int i = 0; i < SIZE; i++) {
        cout << "Value " << i + 1 << ": ";
        cin >> values[i];
    }

    // Creating a new array and copying elements in reverse order
    int reverseArray[SIZE];
    for (int i = 0; i < SIZE; i++) {
        reverseArray[i] = values[SIZE - 1 - i];
    }

    // Printing the elements of the new array
    cout << "Elements of the new array in reverse order:" << endl;
    for (int i = 0; i < SIZE; i++) {
        cout << reverseArray[i] << " ";
    }
    cout << endl;
```

```
    return 0;
}
```

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

```
#include <iostream>
using namespace std;

int main() {
    const int SIZE = 15;
    int values[SIZE];

    // Taking input from the user
    cout << "Please enter " << SIZE << " integer values:" << endl;
    for (int i = 0; i < SIZE; i++) {
        cout << "Value " << i + 1 << ": ";
        cin >> values[i];
    }

    // Calculating the sum and product of the elements
    int sum = 0;
    int product = 1;
    for (int i = 0; i < SIZE; i++) {
        sum += values[i];
        product *= values[i];
    }
```

```
    // Printing the sum and product
    cout << "Sum of all elements: " << sum << endl;
    cout << "Product of all elements: " << product << endl;

    return 0;
}
```