

DIT: OOP Assignment

Name: Brayant Kabugua Kinyua

RegNo: SCT121-0968/2022

Part A:

Instructions for part A: answer all the Questions in this section.

- i. **Using a well labeled diagram, explain the steps of creating a system using OOP principles.** [4 Marks]
- ii. **What is the Object Modeling Techniques (OMT).** [1 Marks]
in object-oriented programming (OOP) acts as a bridge between requirements analysis and system design. It offers a structured approach to translate real-world concepts into objects, classes, and relationships, ultimately forming a blueprint for object-oriented software development.
- iii. **Compare object-oriented analysis and design (OOAD) and object analysis and design (OOP).** [2 Marks]
OOAD: Encompasses the entire software development lifecycle, from gathering requirements and analyzing the problem domain to designing, implementing, and testing the system. It incorporates broader aspects like system architecture, data modeling, and project management.
OOP: Mainly focuses on the analysis and design phases, specifically defining objects, classes, and their relationships. It tends to be more technical and code-oriented, concentrating on implementing object-oriented features within programming languages.
- iv. **Discuss Main goals of UML.** [2 Marks]
UML provides a visual language to represent the structure and behavior of an object-oriented system. This includes diagrams like class diagrams, sequence diagrams, and state diagrams, which make complex relationships between objects and their interactions easily understandable.
- v. **DESCRIBE three advantages of using object oriented to develop an information system.** [3Marks]
Modularity and Reusability: systems are designed using modular components called objects. This modularity allows for the creation of reusable and interchangeable objects, promoting a more efficient and scalable development process.
Encapsulation and Information Hiding: supports encapsulation, where data and methods that operate on the data are encapsulated within a class. Information hiding ensures that the internal details of an object are hidden from the external world.
Flexibility and Extensibility: provides a flexible and extensible framework for developing information systems. New features and functionalities can be added by creating new classes or extending existing ones without modifying the existing code.
- vi. **Briefly explain the following terms as used in object-oriented programming. Write a sample java code to illustrate the implementation of each concept.** [12 Marks]

- a. **Constructor** - Initializes the object's state (sets values for attributes).

```
public class Person {  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

Person p1 = new Person("John", 30); // Triggers the constructor

- b. **Object**- An instance of a class, encapsulating data (attributes) and behavior (methods).

Person p1 = new Person("John", 30); // Creates an object of type Person

- c. **Destructor**

- d. **Polymorphism**- Ability of an object to take different forms and respond to the same message in different ways, depending on its actual type.

```
interface Shape {  
    double area();  
}  
  
class Circle implements Shape {  
    private double radius;  
  
    public Circle(double radius) {  
        this.radius = radius;  
    }  
  
    @Override  
    public double area() {  
        return Math.PI * radius * radius;  
    }  
}  
  
class Square implements Shape {  
    private double side;  
  
    public Square(double side) {  
        this.side = side;  
    }  
}
```

```

    @Override
    public double area() {
        return side * side;
    }
}

```

```

Shape shape1 = new Circle(5);
Shape shape2 = new Square(10);

```

```

double area1 = shape1.area(); // Different implementations based on object
type
double area2 = shape2.area();

```

- e. **Class-** Blueprint for creating objects, defining attributes and methods that objects of the class share.

```

public class Dog {
    // Attributes
    String breed;
    int age;

    // Constructor
    public Dog(String breed, int age) {
        this.breed = breed;
        this.age = age;
    }

    // Method
    public void bark() {
        System.out.println("Dog is barking!");
    }
}

```

- f. **Inheritance-** Allows classes to inherit attributes and methods from other classes (parent class), extending and specializing their functionality.

```

// Base class
class Vehicle {
    void start() {
        System.out.println("Vehicle starts");
    }
}

```

```

// Derived class inheriting from Vehicle
class Car extends Vehicle {
    void drive() {
        System.out.println("Car is driving");
    }
}

```

```

    }
}

public class Main {
    public static void main(String[] args) {
        // Inheritance: Car inherits from Vehicle
        Car myCar = new Car();
        myCar.start(); // Output: Vehicle starts
        myCar.drive(); // Output: Car is driving
    }
}

```

vi. **EXPLAIN** the three types of associations (relationships) between objects in object oriented. [6 Marks]

Association: This is the most basic type of relationship, where two objects simply "know" about each other but don't necessarily rely on each other's existence or functionality.

Composition: This is a stronger association where one object (whole) owns and manages the lifecycle of another object (part). The owned object (part) cannot exist independently and is destroyed when the whole object is destroyed.

Inheritance: This is a specialized relationship where a new class (child) inherits the attributes and methods of an existing class (parent). The child class can then extend or modify the inherited functionality while adding its own.

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. [6 Marks]

A class diagram is a visual representation of the relationships between classes in an object-oriented system. It uses standardized symbols and notations to show the classes, their attributes (data), operations (methods), and the relationships between them.

It is used in:

- ❖ **Analysis and design-** used during the analysis and design phases of software development to understand and solidify the structure and behavior of the system.
- ❖ **Documentation and communication-** used to document the system architecture and communicate technical details to non-technical stakeholders.
- ❖ **Code generation-** in some tools, class diagrams can be used to automatically generate code for specific programming languages.

Steps to Draw a Class Diagram:

1. Identify the Classes: Analyze the system and list all the key entities or concepts that represent real-world objects (e.g., Customer, Product, Order).

2. Define Attributes and Operations: For each class, identify its attributes (data) and operations (methods) that describe its state and behavior.

3. Establish Relationships: Determine the relationships between classes, such as:

Association: Two classes have a connection but don't own or depend on each other.

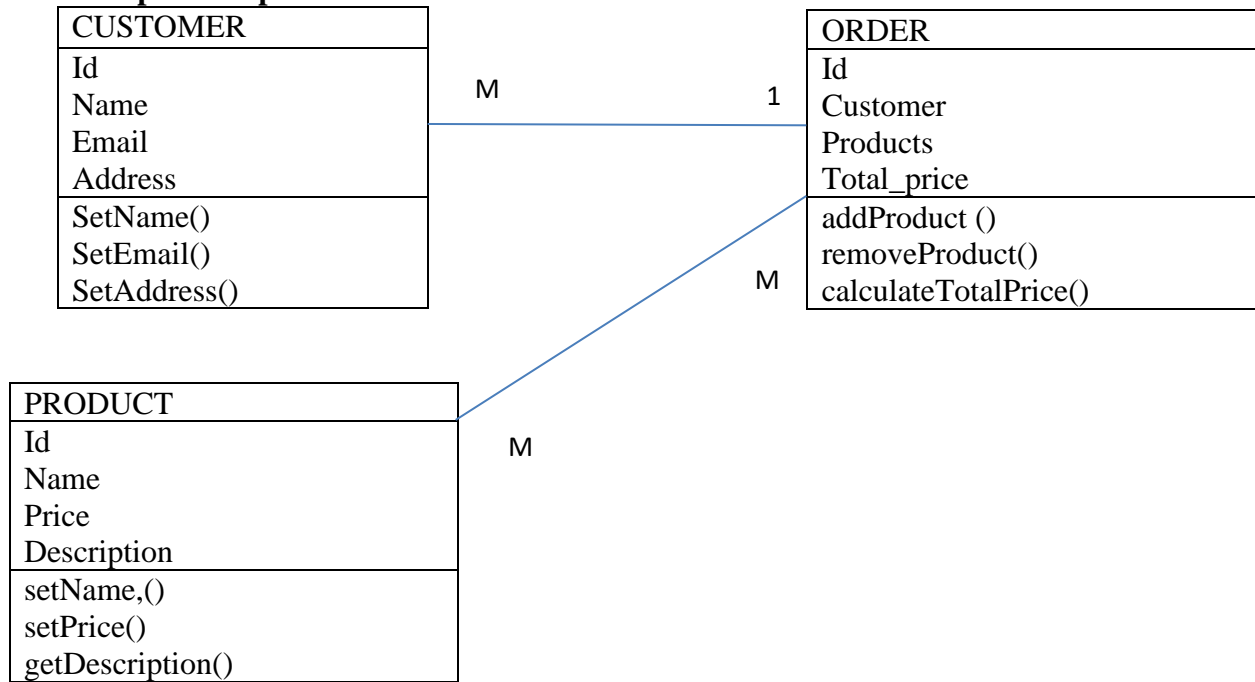
Composition: One class (whole) owns and manages the lifecycle of another (part).

Inheritance: A child class inherits attributes and methods from a parent class.

4.Choose Notation: Use a standard notation like UML (Unified Modeling Language) to represent classes, attributes, operations, and relationships visually.

5.Refine and Iterate: Review and refine the diagram, ensuring it accurately reflects the system's design and is easy to understand.

Example a simple online store:



- vii. **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)** [10 Marks]

Single inheritance: refers to a scenario where a class inherits properties and behaviors from only one superclass. The derived class forms a single chain of inheritance with its superclass.

Code:

```
class Animal {
    // properties and behaviors
};

class Dog : public Animal {
```

```
// additional properties and behaviors specific to Dog
};
```

Multiple inheritance: occurs when a class inherits properties and behaviors from more than one superclass. The derived class has multiple parent classes.

Code:

```
class Shape {
    // properties and behaviors related to shapes
};

class Color {
    // properties and behaviors related to colors
};

class ColoredShape : public Shape, public Color {
    // inherits from both Shape and Color
}
```

Hierarchical inheritance: involves a single superclass being inherited by multiple subclasses. This forms a tree-like structure where a base class serves as a common ancestor for several derived classes.

Code:

```
class Vehicle {
    // properties and behaviors common to all vehicles
};

class Car : public Vehicle {
    // properties and behaviors specific to cars
};

class Motorcycle : public Vehicle {
    // properties and behaviors specific to motorcycles
};
```

b. Friend functions

[5 Marks]

A friend function in C++ is defined as a function that can access private, protected, and public members of a class.

Code:

Friend function syntax

```
class className {
    ... ..
    friend returnType functionName(arguments);
    ... ..
}
```

}

Great learning team. (n.d.). Great Learning Blog: Free Resources What Matters to Shape Your Career!

<https://www.mygreatlearning.com/blog/author/greatlearning/>

c. Method overloading and method overriding

[10 Marks]

Overloaded area and perimeter functions in each shape class for different parameter combinations (e.g., area with radius or side lengths).

area and perimeter functions in child classes override implementations in Shape with specific calculations.

```
#include <iostream>
```

```
class Shape {
public:
    virtual double area() const {
        return 0.0;
    }

    virtual double perimeter() const {
        return 0.0;
    }
};
```

```
class Circle : public Shape {
private:
    double radius;

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

    double area() const override {
        return 3.14 * radius * radius;
    }

    double perimeter() const override {
        return 2 * 3.14 * radius;
    }
};
```

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

```

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

    double area() const override {
        return length * width;
    }

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

    // Method Overloading
    double perimeter(double multiplier) const {
        return multiplier * 2 * (length + width);
    }
};

int main() {
    Circle circle(5);
    std::cout << "Circle - Area: " << circle.area() << ", Perimeter: " <<
circle.perimeter() << std::endl;

    Rectangle rectangle(4, 6);
    std::cout << "Rectangle - Area: " << rectangle.area() << ", Perimeter: " <<
rectangle.perimeter() << std::endl;

    // Method Overloading
    std::cout << "Rectangle - Scaled Perimeter: " << rectangle.perimeter(1.5)
<< std::endl;

    return 0;
}

```

d. Late binding and early binding

[8 Marks]

Late Binding: Using virtual functions for area and perimeter allows dynamic binding at runtime. The actual implementation depends on the object type at runtime.

Early Binding: If no virtual functions are used, binding happens statically during compilation. This can be faster but less flexible.

```
#include <iostream>
```

```

class Shape {
public:
    virtual double area() const {
        return 0.0;
    }
}

```



```

        virtual double perimeter() const {
            return 0.0;
        }
    };

    class Circle : public Shape {
    private:
        double radius;

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

        double area() const override {
            return 3.14 * radius * radius;
        }

        double perimeter() const override {
            return 2 * 3.14 * radius;
        }
    };

    int main() {
        Circle circle(5);

        // Early Binding
        double area = circle.area();           // Early binding - resolved at compile-
time
        double perimeter = circle.perimeter(); // Early binding - resolved at compile-
time

        // Late Binding
        Shape* shape = &circle;
        double lateArea = shape->area();       // Late binding - resolved at runtime
        double latePerimeter = shape->perimeter(); // Late binding - resolved at
runtime

        std::cout << "Early Binding - Area: " << area << ", Perimeter: " <<
perimeter << std::endl;
        std::cout << "Late Binding - Area: " << lateArea << ", Perimeter: " <<
latePerimeter << std::endl;

        return 0;
    }

```

e. Abstract class and pure functions

[6 Marks]

Shape is an abstract class with pure virtual functions for area and perimeter. This forces child classes to define their own implementations, ensuring all shapes can calculate these values.

```
#include <iostream>
```

```
class Shape {
public:
    virtual double area() const = 0;    // Pure virtual function
    virtual double perimeter() const = 0; // Pure virtual function
};
```

```
class Circle : public Shape {
private:
    double radius;

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

    double area() const override {
        return 3.14 * radius * radius;
    }

    double perimeter() const override {
        return 2 * 3.14 * radius;
    }
};
```

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

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

    double area() const override {
        return length * width;
    }

    double perimeter() const override {
        return 2 * (length + width);
    }
};
```

```
int main() {
    Circle circle(5);
```

```
Rectangle rectangle(4, 6);
```

```
Shape* shapes[] = { &circle, &rectangle };
```

```
for (const auto& shape : shapes) {  
    std::cout
```

viii. Using a program written in C++, differentiate between the following. [6 Marks]

a. **Function overloading and operator overloading**

Function Overloading: Allows defining multiple functions with the same name but different parameter types or numbers.

Code:

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

Operator Overloading: Allows defining custom behavior for built-in operators (e.g., +, -, *) for specific data types.

Code:

```
class ComplexNumber {  
    double real, imag;  
public:  
    ComplexNumber(double real, double imag) : real(real), imag(imag) {}  
    ComplexNumber operator+(const ComplexNumber& other) {  
        return ComplexNumber(real + other.real, imag + other.imag);  
    }  
};
```

b. **Pass by value and pass by reference**

Pass by Value: A copy of the argument value is passed to the function, and changes made within the function do not affect the original variable.

Code:

```
void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp; // Swapping copies doesn't affect original values  
}
```

```
int main() {  
    int x = 5, y = 10;  
    swap(x, y); // x and y remain 5 and 10  
}
```

Pass by Reference: A reference to the original variable is passed to the function, allowing direct modification of the original value.

Code:

```
void swap(int& a, int& b) {  
    int temp = a;
```

```

    a = b;
    b = temp; // Swapping references changes original values
}

```

```

int main() {
    int x = 5, y = 10;
    swap(x, y); // Now, x will be 10 and y will be 5
}

```

c. Parameters and arguments

Parameters: Define the type and name of variables used in a function definition.

Code:

```

int add(int a, int b) { // "a" and "b" are parameters
    return a + b;
}

```

Arguments: Are the actual values passed to the function when it is called.

Code:

```

int main() {
    int sum = add(5, 10); // 5 and 10 are arguments
}

```

NOTE: To score high marks, you are required to explain each question in detail. Do good research and cite all the sources of your information. **DO NOT CITE WIKIPEDIA.**

Create a new class called *CalculateG*.

Copy and paste the following initial version of the code. Note variables declaration and the types.

```

class CalculateG {
int main() {

    (datatype) gravity = -9.81; // Earth's gravity in m/s^2 (datatype) fallingTime = 30;

    (datatype) initialVelocity = 0.0; (datatype) finalVelocity = ;

    (datatype) initialPosition = 0.0; (datatype) finalPosition = ;

    // Add the formulas for position and velocity
    Cout<<"The object's position after " << fallingTime << " seconds is "
    + finalPosition + << m."<<endl;
    // Add output line for velocity (similar to position)

} }

```

Modify the example program to compute the position and velocity of an object after falling for 30 seconds, outputting the position in meters. The formula in Math notation is:

$$x(t)=0.5*at^2+v_it+x_i \quad v(t)=at+v_i$$

Run the completed code in Eclipse (Run → Run As → Java Application).

5. Extend *datatype* class with the following code:

```
public class CalculateG {

    public double multi(.....){ // method for multiplication

    }

    // add 2 more methods for powering to square and summation (similar to multiplication)

    public void outline(.....){
        // method for printing out a result

    }
    int main() {

        // compute the position and velocity of an object with defined methods and print out
        the
        result

    } }
```

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

Part B:

Instructions for part B: Do question 1 and any other one question from this section.

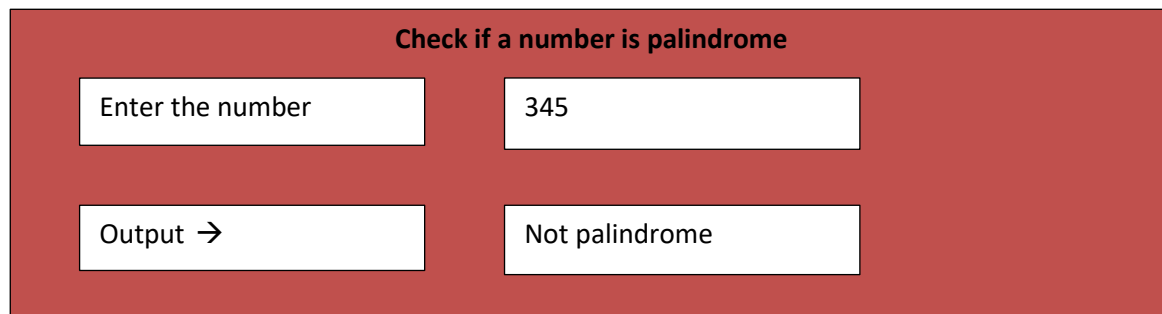
1. Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

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.

Question Two: [15 marks]

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 C++ program to check if a number entered by the user is palindrome or not. You should provide the user with a GUI interface to enter the number and display the results on the same interface.

The interface:



Check if a number is palindrome	
Enter the number	345
Output →	Not palindrome

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.

- 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.