i)The following are the steps involved in creating a system using OOP principles:

Identify the objects that will be used in the system. These objects should represent real-world entities and should have well-defined states and behaviors.

Define the classes that will be used in the system. A class is a blueprint for creating objects. It defines the properties and methods that an object of that class will have.

Create the objects based on the classes that were defined in step 2. Each object should have its own unique state and behavior.

Define the relationships between the objects that were created in step 3. This can be done using inheritance, composition, or aggregation.

Define the behavior of the objects by implementing the methods that were defined in the classes.

Test the system to ensure that it works as expected.

ii) Object Modeling Technique (OMT) is a modeling approach for software modeling and designing.

iii) OOAD is a software engineering methodology that involves using object-oriented concepts to design and implement software systems. While Object-oriented programming (OOP) is a programming paradigm that uses objects to represent real-world entities

iv) The main goals of UML are:

Standardization-UML aims to provide a standardized way of visualizing the design of a system. It offers a common language and set of notations that can be understood by software developers, analysts, designers, and stakeholders involved in the software development process. This standardization helps in communication and understanding among team members.

Visualization-UML allows stakeholders to visualize different aspects of a system using various types of diagrams. These diagrams help in understanding the system's structure, behavior, interactions, and functionalities. By employing different types of diagrams (such as class diagrams, sequence diagrams, activity diagrams, etc.), UML enables a comprehensive view of the system from different perspectives.

v) Three advantages of using OOP to develop an information system:

Modularity- OOP allows developers to break down a large program into smaller, more manageable modules. Each module can be developed and tested independently, making it easier to troubleshoot problems when they arise. This modularity also makes it easier for teams of developers to work on different parts of the program simultaneously, without interfering with each other’s work.

Reusability-OOP promotes code reuse, which means that developers can reuse existing code rather than writing new code from scratch. This saves time and effort, and also reduces the likelihood of errors. In OOP, code is organized into classes, which can be used to create objects. These objects can be reused in different parts of the program, or even in different programs altogether.

Flexibility-OOP allows developers to create programs that are more flexible and adaptable to change. Because OOP programs are organized into classes and objects, it is easier to modify or extend the program without affecting other parts of the program. This makes it easier to add new features or functionality to the program as needed.

vi) A constructor is a special method that is called when an object is created. It is used to initialize the object’s state. In Java, a constructor has the same name as the class and does not have a return type. example of a constructor in Java:

Java code

public class Car {

private String model;

private int year;

public Car (String model, int year) {

this. model = model;

this. year = year;

}

}

An object is an instance of a class. It has state and behavior. In Java, you create an object using the new keyword followed by the name of the class. Here is an example of creating an object in Java:

Java code

Car myCar = newCar("Toyota", "Camry", 2022);

Destructor: In Java, there is no explicit destructor method. Instead, Java has a garbage collector that automatically frees up memory when an object is no longer being used.

class MyClass {

// Override finalize method

@Override

protected void finalize() throws Throwable {

try {

System .out. print ln("Object destroyed.");

} finally {

Super.finalize();

}

}

}

public class Garbage Collection Example {

public static void main(String[] {

MyClass obj = new MyClass();

obj = null

}

}

d)Polymorphism is the ability of an object to take on many forms. In Java, polymorphism is achieved through method overriding and method overloading. Method overriding is when a subclass provides a specific implementation of a method that is already provided by its parent class. Method overloading is when a class has two or more methods with the same name but different parameters.

Java

public class Animal {

public void make Sound() {

System. out. Print ln("The animal makes a sound");

}

}

public class Dog extends Animal {

public void make Sound() {

System. out. Print ln("The dog barks");

}

}

public class Cat extends Animal {

public void make Sound() {

System. Out .print ln("The cat meows");

}

}

public class Main {

public static void main(String[] ) {

Animal my Animal = newAnimall();

Animal my Dog = newDog();

Animal my Cat = newCat();

My Animal. Make Sound();

My Dog.make Sound();

My Cat.make Sound();

}

}

e. A class is a blueprint for creating objects. It defines a set of attributes and methods that the objects will have. In Java, you define a class using the class keyword.

Java code

public class Car {

private String model;

private int year;

public Car(String model, int year) {

this.model = model;

this.year = year;

}

public String getModel() {

return model;

}

public int getYear() {

return year;

}

}

f) Inheritance is the mechanism of basing an object or class upon another object or class, retaining similar implementation. In Java, inheritance is achieved using the extends keyword.

Java

public class Animal {

public void makeSound() {

System.out.println("The animal makes a sound");

}

}

public class Dog extends Animal {

public void bark() {

System .out .print in("The dog barks");

}

}

public class Main {

public static void main(String[] args) {

Dog my Dog = new Dog();

My Dog .make Sound();

my Dog. bark();

}

}

vi)three types of associations relationship between objects:

Association: It is a relationship between two objects where both objects have their own lifecycle and there is no ownership between the objects. For example, a teacher and a student. A teacher can teach multiple students, and a student can be taught by multiple teachers.

Aggregation: It is a specialized form of association where all objects have their own lifecycle, but there is ownership, and child objects can not belong to another parent object.

Composition: It is a specialized form of aggregation where child objects have no meaning or purpose without the parent object. If the parent object is destroyed, all child objects will also be destroyed. For example, a house and its rooms. A house can have multiple rooms, but a room cannot exist without a house.

vii)A class diagram is a type of static structure diagram in the Unified Modeling Language (UML) that describes the structure of a system by showing the system’s classes, their attributes, operations (or methods), and the relationships among objects . It is used to model the static view of a system, which means the classes, interfaces, and their relationships, and is one of the most widely used UML diagrams

Here are the steps to draw a class diagram with an example:

Identify the objects in the problem domain: In this step, we identify the objects that we want to represent in our class diagram. For example, let’s say we want to create a class diagram for a library management system. The objects in this domain could be books, authors, publishers, and members.

Create classes for each of the objects: In this step, we create a class for each of the objects identified in step 1. For example, we can create classes named Book, Author, Publisher, and Member.

Add attributes for those classes: In this step, we add attributes to each of the classes created in step 2. For example, the Book class could have attributes such as title, author, publisher, ISBN, and year of publication.

Add operations for those classes: In this step, we add operations to each of the classes created in step 2. For example, the Book class could have operations such as checkOut(), checkIn(), and reserve().

Add relationships between the classes: In this step, we add relationships between the classes created in step 2. For example, the Book class could have a relationship with the Author class, where each book is written by one or more authors. Similarly, the Book class could have a relationship with the Publisher class, where each book is published by one publisher.

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 concept

# #include <iostream>

#include <cmath>

using namespace std;

class Shape

{

public:

    double area();

    double perimeter();

};

class Circle : public Shape

{

private:

     double radius;

    public: Circle(double rad)

    {

        radius = rad;

    }

    double area()

    {

        return 3.14 \* radius \* radius;

    }

    double perimeter()

    {

        return 2 \* 3.14 \* radius;

    }

};

class Rectangle : public Shape

{

private:

    double length;

    double width;

public:

    Rectangle(double len, double wid)

    {

        length = len;

        width = wid;

    }

    double area()

    {

        return length \* width;

    }

    double perimeter()

    {

        return 2 \* (length \* width);

    }

};

class Triangle : public Shape

{

private:

    double base;

    double height;

    double hypotenuse;

public:

    Triangle(double b, double h, double hypo)

    {

        base = b;

        height = h;

        hypotenuse = hypo;

    }

    double area()

    {

        return 0.2 \* base \* height;

    }

    double perimeter()

    {

        return base + height + hypotenuse;

    }

};

int main()

{

    Circle circle(8);

    circle.area();

    circle.perimeter();

    Rectangle rectangle(7, 9);

    rectangle.area();

    rectangle.perimeter();

    Triangle triangle(4, 6, 10);

    triangle.area();

    triangle.perimeter();

    cout << "Area of Circle : " << circle.area() << endl;

     cout << "Perimeter of Circle : " << circle.perimeter() << endl;

     cout << "Area of Rectangle : " << rectangle.area() << endl;

      cout << "Perimeter of Rectangle : " << rectangle.perimeter() << endl;

       cout << "Area of Triangle : " << triangle.area() << endl;

        cout << "Perimeter of Triangle : " << triangle.area() << endl;

    return 0;

a)Inheritance (Single inheritance, Multiple inheritance and Hierarchical inheritance)

Inheritance The Circle, Rectangle, Triangle, and Square classes inherit from the Shape class.

b)Friend functions: None are used in this implementation.

c)Method overloading and overriding: The get Area() and get Perimeter() methods are overridden in each of the derived classes.

d)Late binding and early binding: Late binding is used in this implementation because the get Area() and get Perimeter() methods are virtual functions. Early binding is used for non-virtual functions.

e)Abstract class and pure functions: The Shape class is an abstract class because it contains pure virtual functions.

viii)

#include <iostream>

using namespace std;

// Function overloading

int add(int x, int y) {

return x + y;

}

double add(double x, double y) {

return x + y;

}

// Operator overloading

class Complex {

private:

double real, imag;

public:

Complex(double r = 0, double i = 0) {

real = r;

imag = i;

}

Complex operator + (Complex const &obj) {

Complex res;

res.real = real + obj.real;

res.imag = imag + obj.imag;

return res;

}

void print() {

cout << real << " + i" << imag << endl;

}

};

// Pass by value

void swap(int x, int y) {

int temp = x;

x = y;

y = temp;

}

// Pass by reference

void swap(int &x, int &y) {

int temp = x;

x = y;

y = temp;

}

int main() {

// Function overloading

cout << "Function overloading:" << endl;

cout << "add(2, 3) = " << add(2, 3) << endl;

cout << "add(2.5, 3.5) = " << add(2.5, 3.5) << endl;

// Operator overloading

cout << "\nOperator overloading:" << endl;

Complex c1(10, 5), c2(2, 4);

Complex c3 = c1 + c2;

c3.print();

// Pass by value

cout << "\nPass by value:" << endl;

int a = 2, b = 3;

swap(a, b);

cout << "a = " << a << ", b = " << b << endl;

// Pass by reference

cout << "\nPass by reference:" << endl;

int c = 2, d = 3;

swap(c, d);

cout << "c = " << c << ", d = " << d << endl;

// Parameters and arguments

cout << "\nParameters and arguments:" << endl;

int e = 2, f = 3;

cout << "add(" << e << ", " << f << ") = " << add(e, f) << endl;

return 0;

}

This program demonstrates the following concepts:

Function overloading is demonstrated by the add() function, which is defined twice with different parameter types. The first add() function takes two integers as arguments and returns their sum, while the second add() function takes two doubles as arguments and returns their sum.

Operator overloading is demonstrated by the Complex class, which overloads the + operator to add two complex numbers. The + operator is redefined to add the real and imaginary parts of two Complex objects.

Pass by value is demonstrated by the swap() function, which takes two integers as arguments. The first swap() function takes two integers as arguments and swaps their values using a temporary variable. However, since the arguments are passed by value, the original variables are not modified.

Pass by reference is demonstrated by the second swap() function, which takes two integer references as arguments. The second swap() function swaps the values of the two integers by modifying their references.

Parameters and arguments are demonstrated by the add() function, which takes two integer parameters and returns their sum. The add() function is called with two integer arguments, which are passed to the function as parameters.

public class Calculate {

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

    double falling time= 30;

    double initialVelocity = 0.0;

    double finalVelocity;

    double initialPosition = 0.0;

    double finalPosition;

    public double multiply(double a, double b) {

        return a \* b;

    }

    public double power(double base, int exponent) {

        return mathpow(base, exponent);

    }

    public double sum(double a, double b) {

        return a + b;

    }

    public void outline(String message) {

        System.out.println(message);

    }

    public void calculate() {

        // Calculate final position using the formula x(t) = 0.5 \* a \* t^2 + v\_i \* t + x\_i

        finalPosition = 0.5 \* gravity \* power(fallingTime, 2) + multiply(initialVelocity, fallingTime) + initialPosition;

        // Calculate final velocity using the formula v(t) = a \* t + v\_i

        finalVelocity = multiply(gravity, fallingTime) + initialVelocity;

        outline("The object's position after " + fallingTime + " seconds is " + finalPosition + " m.");

        outline("The object's velocity after " + fallingTime + " seconds is " + finalVelocity + " m/s.");

    }

    public static void main(String[] args) {

        CalculateG calculateG = new CalculateG();

        calculateG.calculate();

    }

}

This version of the CalculateG class includes methods for multiplication, powering, summation, and an outline method for printing results. The calculate method computes the position and velocity of the falling object using the provided formulas and then prints out the results using the outline method.

To execute this code in Eclipse:

Copy the code into a Java class file (named CalculateG.java, for example).

Open Eclipse and create a new Java project or use an existing one.

Add this class (CalculateG) to the source folder of your project.

Right-click on the file, then select Run As > Java Application.

6)public class CalculateG {

// Method for multiplication

public static int multiply(int a, int b) {

return a \* b;

}

// Method for powering to square

public static int square(int a) {

return a \* a;

}

// Method for summation

public static int sum(int[] numbers) {

int total = 0;

for (int num : numbers) {

total += num;

}

return total;

}

// Method to print out the result

public static void printResult(int result) {

System.out.println("The result is: " + result);

}

public static void main(String[] args) {

// Example usage of the methods

int multiplicationResult = multiply(5, 10);

printResult(multiplicationResult);

int squareResult = square(8);

printResult(squareResult);

int[] numbers = {1, 2, 3, 4, 5};

int sumResult = sum(numbers);

printResult(sumResult);

}

}

Part B

#include <iostream>

// Function to find the sum of even Fibonacci numbers up to a limit

long long sumEvenFibonacci(int limit) {

    long long sum = 0;

    int a = 1, b = 2, c = 0;

    while (b <= limit) {

        if (b % 2 == 0) {

            sum += b;

        }

        c = a + b;

        a = b;

        b = c;

    }

    return sum;

}

int main() {

    int limit = 4000000;

    long long result = sumEvenFibonacci(limit);

    std::cout << "The sum of even-valued terms in the Fibonacci sequence not exceeding 4 million is: " << result << std::endl;

    return 0;

}

QUESTION 15

#include <iostream>

using namespace std;

int main()

{

    const int length = 15;

    int arr[length];

    cout<<"Enter 15 integers" << endl;

    for (int i = 0; i < length; i++)

    {

        cin >> arr[i];

    }

    cout<<"Values in the array" << endl;

    for (int i = 0; i < length; i++)

    {

        cout << arr[i] << endl;

    }

    int number;

    cout<<"Enter the number to search" << endl;

    cin >> number;

    bool exists = false;

    int i;

    for (i = 0; i < length; i++)

    {

        if (arr[i] == number)

        {

            exists = true;

        }

    }

    if (exists)

    {

        cout<<"Number is located at index " << i << endl;

    }

    else

    {

        cout<<"Number is not in the array " << endl;

    }

    int reverse[length];

    for (int i = 0; i < length; i++)

    {

        reverse[i] = arr[length - 1 - i];

    }

    cout<<"Elements of the array in a reversed order" << endl;

    for (int i = 0; i < length; i++)

    {

        cout << reverse[i] << endl;

    }

    int sum = 0;

    long product = 1;

    for (int i = 0; i < length; i++)

    {

        sum += arr[i];

        product \*= arr[i];

    }

    cout<<"Sum of all elements " << sum << endl;

    cout<<"Product of the elements " << product << endl;

    return 0;

}

EDITH AOKO

SCT121-1012/2022