**PART A:**

**i. Creating a System Using OOP Principles:**

1. Identify Objects: Identify the objects relevant to the system.
2. Define Classes: Create classes for each identified object.
3. Define Attributes: Assign attributes (properties) to each class.
4. Define Behaviors: Add methods (functions) to implement behaviors.
5. Encapsulation: Encapsulate the attributes and methods within classes.
6. Inheritance: Establish relationships and hierarchies between classes.
7. Polymorphism: Allow objects of different classes to be treated uniformly.

**ii. Object Modeling Techniques (OMT):**

Object Modeling Techniques (OMT) is a method for object-oriented analysis and design that uses graphical notation to depict the objects, classes, and their relationships.

**iii. OOAD vs. OOP:**

Object-Oriented Analysis and Design (OOAD): It’s a software methodology that involves using Object oriented concept to design and implement software system.

Object Analysis and Design (OOP) is basically a computer programming design philosophy or methodology that organizes/ models software design around data, or objects rather than functions and logic.

**iv. Main Goals of UML:**

* Provide a standardized modeling language.
* Support visual representation of system structure and behavior.
* Facilitate communication and understanding among stakeholders.

**v. Advantages of Object-Oriented Development:**

* Modularity: Code is organized into manageable, independent modules.
* Reusability: Classes and objects can be reused in different parts of the system.
* Maintainability: Changes in one part of the system do not affect others.

**vi. Explanation of Terms in OOP:**

1. **Constructor:** Initializes an object when it is created.

public class Person {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

}

1. **Object:** An instance of a class.

Person person = new Person("John", 30);

1. **Polymorphism** is the ability of an object to take on many forms.

public class Animal {

public void makeSound() {

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

}

}

public class Dog extends Animal {

@Override

public void makeSound() {

System.out.println("The dog barks");

}

}

public 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

}

}

1. **Class:** A class is a blueprint for creating objects.

public class Person {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

}

**e) Destrutor is not used in Java**

**f. Inheritance**: Inheritance is the ability of a class to inherit properties and methods from its superclasspublic class Animal {

public void makeSound() {

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

}

}

public class Dog extends Animal {

public void bark() {

System.out.println("The dog barks");

}

}

**vii. Three Types of Associations in OOP:**

* Aggregation: Weaker relationship; objects can exist independently.
* Composition: Stronger relationship; one object owns another.
* Association: General relationship; one class is related to another.

**viii. Class Diagram:**

A **class diagram** is a visual representation of classes, relationships, and their associations in a system.

It is Used in UML to model the static view of a system.

**Steps to Draw:**

1. Identify classes.
2. Determine relationships and associations.
3. Add attributes and methods.
4. Draw connections between classes.

**vii. Inheritance, Friend Functions, Method Overloading, Late Binding, Abstract Class:**

#include <iostream>

#include <cmath>

***// Abstract class Shape***

class Shape {

public:

virtual double area() const = 0;

virtual double perimeter() const = 0;

};

***// Concrete class Circle inheriting from Shape***

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;

}

};

***// Concrete class Rectangle inheriting from Shape***

class Rectangle : public Shape {

private:

double length;

double 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);

}

};

***// Concrete class Triangle inheriting from Shape***

class Triangle : public Shape {

private:

double side1, side2, side3;

public:

Triangle(double s1, double s2, double s3) : side1(s1), side2(s2), side3(s3) {}

double area() const override {

double s = (side1 + side2 + side3) / 2;

return sqrt(s \* (s - side1) \* (s - side2) \* (s - side3));

}

double perimeter() const override {

return side1 + side2 + side3;

}

};

***// Concrete class Square inheriting from Rectangle***

class Square : public Rectangle {

public:

Square(double side) : Rectangle(side, side) {}

};

int main() {

*// Example usage of the classes*

Circle circle(5.0);

Rectangle rectangle(4.0, 6.0);

Triangle triangle(3.0, 4.0, 5.0);

Square square(2.0);

std::cout << "Circle Area: " << circle.area() << ", Perimeter: " << circle.perimeter() << std::endl;

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

std::cout << "Triangle Area: " << triangle.area() << ", Perimeter: " << triangle.perimeter() << std::endl;

std::cout << "Square Area: " << square.area() << ", Perimeter: " << square.perimeter() << std::endl;

return 0;

}

**viii. Differentiation in C++:**

**a. Function Overloading and Operator Overloading:**

**Function Overloading**: Multiple functions with the same name but different parameters or types.

**Operator Overloading:** Redefining operators for user-defined types. In the Complex class, + operator is overloaded.

#include <iostream>

***// Function Overloading***

void print(int num) {

std::cout << "Integer: " << num << std::endl;

}

void print(double num) {

std::cout << "Double: " << num << std::endl;

// Operator Overloading

class Complex {

public:

double real;

double imag;

Complex operator+(const Complex& other) {

Complex result;

result.real = real + other.real;

result.imag = imag + other.imag;

return result;

}

};

int main() {

int num1 = 5;

double num2 = 3.14;

***// Function Overloading***

print(num1);

print(num2);

***// Operator Overloading***

Complex c1, c2, result;

result = c1 + c2;

return 0;

}

**b. Pass by Value and Pass by Reference:**

**Pass by Value**: A copy of the variable is passed to the function, modifications inside the function do not affect the original variable.

**Pass by Reference:** The memory address of the variable is passed to the function, modifications inside the function directly affect the original variable.

#include <iostream>

***// Function to demonstrate pass by value***

void passByValue(int x) {

std::cout << "Inside passByValue function (Before Modification): " << x << std::endl;

x = 10;

std::cout << "Inside passByValue function (After Modification): " << x << std::endl;

}

***// Function to demonstrate pass by reference***

void passByReference(int &x) {

std::cout << "Inside passByReference function (Before Modification): " << x << std::endl;

x = 10;

std::cout << "Inside passByReference function (After Modification): " << x << std::endl;

}

int main() {

int num = 5;

***// Passing by value***

std::cout << "Main function (Before passByValue): " << num << std::endl;

passByValue(num);

std::cout << "Main function (After passByValue): " << num << std::endl;

***// Passing by reference***

std::cout << "Main function (Before passByReference): " << num << std::endl;

passByReference(num);

std::cout << "Main function (After passByReference): " << num << std::endl;

return 0;

}

**C). Parameters:** The Variables are declared in the function signature

**Arguments:** Actual values passed to the function when calling it

#include <iostream>

***// Function with parameters***

void exampleFunction(int param1, double param2) {

std::cout << "Parameter 1: " << param1 << std::endl;

std::cout << "Parameter 2: " << param2 << std::endl;

}

int main() {

int arg1 = 1;

double arg2 = 2.5;

// Calling the function with arguments

exampleFunction(arg1, arg2);

return 0;

}

**Part A -CONTINUATION**

public class CalculateG {

***// Variables***

double gravity = -9.81;

double fallingTime = 30;

double initialVelocity = 0.0;

double finalVelocity;

double initialPosition = 0.0;

double finalPosition;

***// Method to compute position and velocity***

public void calculatePositionAndVelocity() {

***// Add formulas for position and velocity***

finalPosition = 0.5 \* gravity \* Math.pow(fallingTime, 2) + initialVelocity \* fallingTime + initialPosition;

finalVelocity = gravity \* fallingTime + initialVelocity;

***// Output lines for position and velocity***

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

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

}

***// Method for multiplication***

public double multi(double a, double b) {

return a \* b;

}

***// Methods for powering to square and summation***

public double powerToSquare(double a) {

return Math.pow(a, 2);

}

public double summation(double a, double b) {

return a + b;

}

***// Method for printing out a result***

public void outline(String result) {

System.out.println(result);

}

public static void main(String[] args) {

***// Create an instance of the CalculateG class***

CalculateG calculator = new CalculateG();

***// Compute the position and velocity of an object***

calculator.calculatePositionAndVelocity();

***// Example usage of other methods***

double multiplicationResult = calculator.multi(2, 3);

System.out.println("Multiplication result: " + multiplicationResult);

double squareResult = calculator.powerToSquare(4);

System.out.println("Square result: " + squareResult);

double summationResult = calculator.summation(5, 7);

System.out.println("Summation result: " + summationResult);

calculator.outline("This is a result message.");

}

}

**PART B**

**QUESTION 1: C++ Method for Fibonacci Sequence**

#include <iostream>

int main() {

int term1 = 1, term2 = 2, nextTerm = 0, sum = 0;

while (term1 <= 4000000) {

if (term1 % 2 == 0) {

sum += term1;

}

nextTerm = term1 + term2;

term1 = term2;

term2 = nextTerm;

}

std::cout << "Sum of even-valued terms: " << sum << std::endl;

return 0;

}

**Question 2 - C++ Palindrome Check Program with GUI**

#include <iostream>

#include <string>

bool isPalindrome(int num) {

std::string original = std::to\_string(num);

std::string reversed(original.rbegin(), original.rend());

return original == reversed;

}

int main() {

int userNumber;

***// GUI: Input from user***

std::cout << "Enter a number: ";

std::cin >> userNumber;

if (isPalindrome(userNumber)) {

std::cout << userNumber << " is a palindrome." << std::endl;

} else {

std::cout << userNumber << " is not a palindrome." << std::endl;

}

***// GUI: Display result***

return 0;

}

**Question 3 - C++ Program for Array Operations**

#include <iostream>

#include <algorithm>

int main() {

const int size = 15;

int values[size];

***// a) Print values stored in the array***

std::cout << "Values in the array: ";

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

std::cout << values[i] << " ";

}

std::cout << std::endl;

***// b) Ask user to enter a number and check if it's present in the array***

int searchNumber;

std::cout << "Enter a number to search: ";

std::cin >> searchNumber;

auto result = std::find(std::begin(values), std::end(values), searchNumber);

if (result != std::end(values)) {

std::cout << "The number found at index " << std::distance(std::begin(values), result) << std::endl;

} else {

std::cout << "Number not found in this array" << std::endl;

}

***// c) Create another array, copy elements in reverse order, and print***

int reversedArray[size];

std::reverse\_copy(std::begin(values), std::end(values), std::begin(reversedArray));

std::cout << "Values in the reversed array: ";

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

std::cout << reversedArray[i] << " ";

}

std::cout << std::endl;

***// d) Get the sum and product of all elements***

int sum = 0, product = 1;

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

sum += values[i];

product \*= values[i];

}

std::cout << "Sum of array elements: " << sum << std::endl;

std::cout << "Product of array elements: " << product << std::endl;

return 0;

}