# REEVES LITABALIA

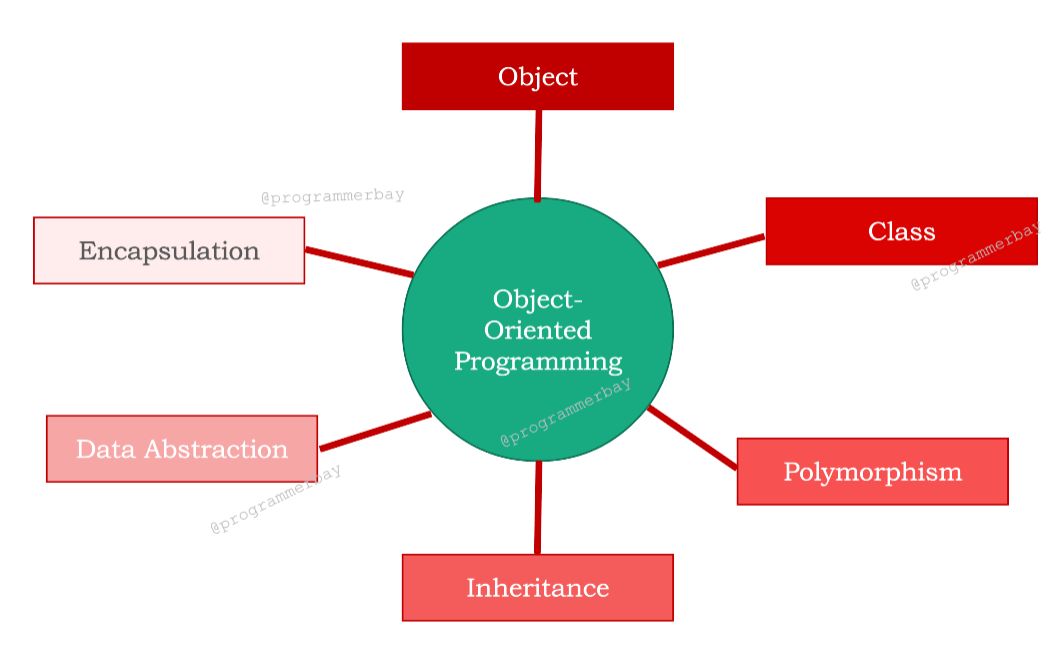
SCT 121-0463/2020

DIT: OOP

# **Part A**

I.

1. **Identify Objects:**
   * Identify the entities or objects that are relevant to the system. Objects represent real-world entities or concepts and encapsulate both data and behavior.
2. **Define Classes:**
   * Create classes for each identified object. A class is a blueprint or template for creating objects. It defines the properties (attributes) and behaviors (methods) that the objects will have.
3. **Encapsulation:**
   * Encapsulate the properties and behaviors within each class. Encapsulation involves bundling the data (attributes) and methods that operate on the data together within a class, and restricting access to certain components.
4. **Inheritance:**
   * Identify relationships between classes and establish inheritance hierarchies. Inheritance allows a class (subclass or derived class) to inherit properties and behaviors from another class (superclass or base class). It promotes code reuse and establishes an "is-a" relationship.
5. **Polymorphism:**
   * Implement polymorphism to allow objects of different classes to be treated as objects of a common base class. This involves using interfaces, abstract classes, or method overriding to provide a single interface for different types of objects.
6. **Instantiate Objects:**
   * Create instances or objects of the classes defined. Instantiation involves creating specific copies of a class, each with its own unique set of attributes and behaviors.
7. **Define Relationships:**
   * Establish relationships between objects. This involves specifying how objects interact with each other, such as through associations, aggregations, or compositions.
8. **Define Methods:**
   * Implement methods within each class to define the behaviors associated with the objects. Methods represent the actions that objects can perform.
9. **Refine and Test:**
   * Refine the design, if necessary, and thoroughly test the system. Testing ensures that the objects and their interactions work as intended and that the system behaves as expected.
10. **Maintenance and Updates:**
    * Ongoing maintenance and updates may be required to address changing requirements or to enhance the system. The modular nature of OOP allows for easier maintenance and extensibility.



* 1. Object Modeling Techniques (OMT) is a methodology for modeling and designing software systems based on the concept of objects. Objects, in the context of OMT, represent real-world entities and the interactions between them. The goal of OMT is to provide a systematic and structured approach to software development by capturing and representing the essential aspects of a system.
  2. **Object-Oriented Analysis and Design (OOAD):**

OOAD is a methodology that involves analyzing and designing a system by visualizing it as a group of interacting objects, each defined by its role, behavior, and state. This methodology is particularly associated with the early stages of the software development life cycle (SDLC) and focuses on understanding and modeling the problem domain.

**Emphasis:** The emphasis in OOAD is on understanding the system requirements, identifying objects in the problem domain, and defining their relationships and behaviors. Key techniques include use case modeling, class diagrams, sequence diagrams, and collaboration diagrams.

**Object-Oriented Programming (OOP):**

**Definition:** OOP is a programming paradigm that uses objects – instances of classes – to design and implement software. It is closely related to OOAD but is more concerned with the actual implementation of the system. Languages like Java, C++, and Python are examples of OOP languages.

**Emphasis:** In OOP, the emphasis is on the implementation phase of the SDLC. Objects are instances of classes, and they encapsulate data and behavior. Key principles include encapsulation, inheritance, and polymorphism.

**Comparison:**

* **Focus:**
  + OOAD focuses on the analysis and design phases of software development, aiming to create a conceptual model of the problem domain.
  + OOP focuses on the implementation phase, using classes and objects to structure and organize code.
* **Purpose:**
  + OOAD's purpose is to understand the problem domain, identify objects, and establish relationships between them.
  + OOP's purpose is to implement the identified objects in a programming language, using concepts like encapsulation, inheritance, and polymorphism.
* **Artifacts:**
  + OOAD produces artifacts such as use case diagrams, class diagrams, and collaboration diagrams.
  + OOP produces code artifacts, including classes, objects, and methods.
* **Lifecycle Stage:**
  + OOAD is typically associated with the early stages of the SDLC.
  + OOP is associated with the later stages, where coding and implementation take place.

In summary, OOAD and OOP are related concepts, with OOAD focusing on the analysis and design of a system, while OOP focuses on its implementation using programming languages and specific coding principles. These two aspects are interconnected and together contribute to a comprehensive approach to software development.

IV

Unified Modeling Language (UML) is a standardized modeling language in the field of software engineering that aims to provide a visual representation of a system for better understanding and communication among stakeholders. The main goals of UML are multifaceted and revolve around improving the software development process. Here are the key goals of UML:

**Visual Representation and Communication:**

Explanation: UML's primary goal is to provide a visual modeling language that facilitates communication among stakeholders. It uses graphical notations to represent different aspects of a system, such as its structure, behavior, and interactions. This visual representation helps in conveying complex information in a more understandable and accessible manner.

**Sources**: Rumbaugh, J., Jacobson, I., & Booch, G. (2005). "Unified Modeling Language Reference Manual, The (2nd Edition)." Addison-Wesley.

**Modeling Abstraction:**

UML allows developers to create abstract models that capture the essential aspects of a system without delving into unnecessary details. This abstraction is crucial for managing complexity and focusing on the key elements of a system's design, promoting a clearer understanding of the system.

**Sources:** Fowler, M., Scott, K. (1997). "UML Distilled: A Brief Guide to the Standard Object Modeling Language." Addison-Wesley.

**Standardization:**

UML provides a standardized notation and set of conventions for modeling software systems. This standardization ensures that developers, analysts, and other stakeholders use a common language when representing and discussing system designs, promoting consistency and clarity in communication.

**Sources:** Object Management Group (OMG). (2017). "Unified Modeling Language (UML) - Version 2.5." Retrieved from https://www.omg.org/spec/UML/2.5/.

**Flexibility and Extensibility:**

UML is designed to be a flexible and extensible modeling language. It accommodates various modeling needs and allows for the creation of custom modeling elements and profiles. This adaptability makes UML suitable for a wide range of applications and industries.

**Sources:** Ambler, S. W. (2004). "Introduction to UML 2 Activity Diagrams." Retrieved from http://www.agilemodeling.com/artifacts/activityDiagram.htm.

**Analysis and Design Support:**

UML supports both analysis and design phases of software development. It provides modeling constructs for capturing requirements (use cases, requirements diagrams) as well as design elements (class diagrams, sequence diagrams). This dual support helps bridge the gap between system requirements and the actual software design.

**Sources:** Booch, G., Rumbaugh, J., & Jacobson, I. (1998). "The Unified Modeling Language User Guide." Addison-Wesley.

In summary, the main goals of UML include providing a visual means of communication, supporting modeling abstraction, standardizing modeling notations, offering flexibility and extensibility, and aiding in both analysis and design phases of software development. These goals collectively contribute to enhancing the efficiency, clarity, and collaboration in the software development process.

V.

**Modularity and Reusability:**

OOP allows the decomposition of a system into modular, self-contained units called objects. Each object represents a real-world entity and encapsulates its data and behavior. This modularity enhances code organization and maintenance, making it easier to understand and modify. Additionally, objects can be reused in different parts of the system or even in other projects, promoting code reusability. This reusability not only saves development time but also improves the overall quality of the code.

**Source:** Grady Booch, in his book "Object-Oriented Analysis and Design with Applications," emphasizes the modularity and reusability benefits of OOP in system development.

**Encapsulation and Data Security:**

OOP supports encapsulation, where the internal details of an object are hidden from the outside world, and access is controlled through well-defined interfaces. This encapsulation enhances data security by preventing unauthorized access and modification of an object's internal state. It also promotes the concept of data hiding, allowing the system to protect sensitive information and reducing the likelihood of unintended side effects.

**Source:** Bertrand Meyer's book "Object-Oriented Software Construction" discusses the concept of encapsulation and its role in ensuring data security in information systems.

**Inheritance for Code Reuse and Extensibility:**

Explanation: Inheritance, a fundamental concept in OOP, allows the creation of new classes by inheriting attributes and behaviors from existing classes. This promotes code reuse and minimizes redundancy, as common functionalities can be defined in a base class and inherited by subclasses. In addition to code reuse, inheritance facilitates system extensibility, enabling developers to introduce new features or modify existing ones without affecting the entire codebase.

**Source**: "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides explains the role of inheritance in creating extensible and maintainable software through the use of design patterns.

In conclusion, the use of object-oriented programming in developing information systems offers advantages such as modularity and reusability, encapsulation for data security, and inheritance for code reuse and extensibility. These principles contribute to the creation of scalable, maintainable, and secure software systems.

VI.

a. **Constructor:**

A constructor is a special method in a class that is automatically called when an object of the class is created. It is used to initialize the object's state or perform any necessary setup operations. Constructors have the same name as the class and do not have a return type.

**Java Code:**

```java

public class MyClass {

// Constructor

public MyClass() {

System.out.println("Constructor called. Object created!");

}

public static void main(String[] args) {

// Creating an object of MyClass invokes the constructor

MyClass myObject = new MyClass();

}

}

```

b. **Object:**

An object is an instance of a class in object-oriented programming. It is a tangible entity that represents a real-world concept and encapsulates data and behavior. Objects are created from classes and can interact with each other through method calls.

**Java Code:**

```java

public class Car {

String model;

public static void main(String[] args) {

// Creating an object of the Car class

Car myCar = new Car();

MyCar. Model = "Toyota Camry";

System.out.println ("My car model is: " + myCar. Model);

}

}

```

c. **Destructor:**

In Java, there is no explicit destructor like in some other programming languages. Java uses automatic garbage collection to reclaim memory occupied by objects that are no longer referenced. The `finalize` method can be considered similar to a destructor, but it is not guaranteed to be called immediately when an object goes out of scope.

**Java Code:**

```java

public class MyClass {

@Override

protected void finalize() throws Throwable {

System.out.println("Destructor called. Object will be garbage collected.");

}

public static void main(String[] args) {

MyClass myObject = new MyClass();

myObject = null; // Making the object eligible for garbage collection

System.gc(); // Suggesting garbage collection

}

}

```

d. **Polymorphism:**

Polymorphism allows objects of different types to be treated as objects of a common type. In Java, this is achieved through method overloading and method overriding. Method overloading occurs when multiple methods have the same name but different parameter lists, while method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.

**Java Code:**

```java

// Method Overloading

public class MathOperations {

public int add(int a, int b) {

return a + b;

}

public double add(double a, double b) {

return a + b;

}

}

// Method Overriding

public class Animal {

public void makeSound() {

System.out.println("Animal makes a sound");

}

}

Public class Dog extends Animal {

@Override

Public void makeSound () {

System.out.println ("Dog barks");

}

}

```

e. **Class:**

A class is a blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that objects of the class will have. Objects are instances of classes, and classes provide a way to structure and organize code in a modular and reusable manner.

**Java Code:**

```java

public class Rectangle {

Int length;

Int width;

Public int calculateArea () {

Return length \* width;

}

}

public class Main {

public static void main(String[] args) {

// creating an object of the Rectangle class

Rectangle myRectangle = new Rectangle ();

myRectangle.length = 5;

myRectangle.width = 3;

System.out.println ("Area of the rectangle: " + myRectangle.calculateArea ());

}

}

```

f. **Inheritance:**

Inheritance is a mechanism in object-oriented programming that allows a class (subclass or derived class) to inherit properties and behaviors from another class (superclass or base class). It promotes code reuse and establishes a relationship between classes, where the subclass can use and extend the functionality of the superclass.

**Java Code:**

```java

// Superclass

public class Animal {

public void eat() {

System.out.println("Animal eats");

}

}

// Subclass inheriting from Animal

public class Dog extends Animal {

public void bark() {

System.out.println("Dog barks");

}

}

// Using the inheritance

public class Main {

public static void main(String[] args) {

Dog myDog = new Dog();

myDog.eat(); // Inherited method from Animal

myDog.bark(); // Method specific to Dog

}

}

```

VI. In object-oriented programming (OOP), associations refer to the relationships between objects. These relationships are crucial for modeling the interactions and dependencies among different classes or objects in a system. There are three primary types of associations in OOP:

1. **Association:**

Association is a general relationship between two or more classes, where objects of one class are related to objects of another class. It represents a bi-directional connection and can be one-to-one, one-to-many, or many-to-many.

**Example:** Consider a scenario where a `Student` class is associated with a `Course` class. A student can enroll in multiple courses, and a course can have multiple students.

```java

class Student {

// attributes and methods

}

class Course {

// attributes and methods

}

// Association

class Enrollment {

Student student;

Course course;

}

```

**Explanation:** The `Enrollment` class represents the association between `Student` and `Course`. It holds references to both `Student` and `Course` objects, illustrating the relationship.

2. **Aggregation:**

**Definition:** Aggregation is a specialized form of association where a "whole" class has a relationship with a "part" class. It represents a weaker form of association, implying a looser coupling between objects.

**Example:** Consider a `Department` class composed of multiple `Employee` classes. Here, a department aggregates employees, and an employee can exist independently of a specific department.

```java

class Department {

List<Employee> employees;

// other attributes and methods

}

class Employee {

// attributes and methods

}

```

**Explanation:** The `Department` class aggregates `Employee` objects. The relationship is more of a "has-a" connection, and if the department is dissolved, the employees can still exist.

3. **Composition:**

**Definition:** Composition is a stronger form of aggregation, where the "part" class is an integral part of the "whole" class. The part class cannot exist independently of the whole.

**Example:** A `Car` class composed of `Engine`, `Wheel`, and `Chassis` classes. A car cannot exist without its essential components, forming a composition relationship.

```java

class Car {

Engine engine;

List<Wheel> wheels;

Chassis chassis;

// other attributes and methods

}

class Engine {

// attributes and methods

}

class Wheel {

// attributes and methods

}

class Chassis {

// attributes and methods

}

```

**Explanation:** In this case, the `Car` class is composed of essential components like `Engine`, `Wheel`, and `Chassis`. If the car is destroyed, these components are also typically disposed of, illustrating the stronger relationship.

In summary, associations, aggregations, and compositions represent different levels of relationships between objects in object-oriented programming, ranging from a general connection to a more specialized and tightly coupled relationship. These concepts help in creating more modular, maintainable, and flexible software designs.

VII.  
A class diagram is a type of diagram from the Unified Modeling Language (UML) that represents the structure and relationships of classes and their members in an object-oriented system. It provides a static view of a system, focusing on the classes and their associations, attributes, and methods. Class diagrams are widely used in software development to visualize the design and architecture of a system.

**Usage of Class Diagrams:** Class diagrams are used for several purposes in software development:

**System Design:** Class diagrams help in designing the structure of a system by representing classes, their attributes, and relationships.

**Communication:** They serve as a communication tool among developers, stakeholders, and team members by providing a clear and visual representation of the system's architecture.

**Code Generation:** Class diagrams are often used as a basis for generating code in object-oriented programming languages.

**Documentation:** They act as documentation for the system, helping developers and other stakeholders understand the relationships and interactions between different components.

**Steps to Draw a Class Diagram:**

Drawing a class diagram involves several steps. Let's go through these steps with an example of a simple library management system:

**Step 1: Identify Classes and Objects:** Identify the main classes and objects in the system. In our library management system example, classes could include Book, Author, Member, and Librarian.

**Step 2: Identify Attributes:** Determine the attributes (properties) of each class. For the Book class, attributes might include ISBN, title, author, and available copies.

**Step 3: Identify Methods:** Define the methods or behaviors associated with each class. For the Book class, methods might include checkOut(), returnBook(), and getAvailableCopies().

**Step 4: Define Relationships:** Establish relationships between classes. For example, a Member class may have a relationship with the Book class, indicating that a member can borrow multiple books.

**Step 5: Multiplicity and Role:** Specify the multiplicity (how many instances are involved) and roles (the name given to the role a class plays in a relationship). For instance, a Member can have a "borrows" relationship with the Book class, where the multiplicity is 1..\* (indicating a member can borrow multiple books).

**Step 6: Draw the Diagram:** Using a diagramming tool or software, create the class diagram by representing classes, attributes, methods, relationships, multiplicity, and roles. Use appropriate symbols and notation from UML.

VII.

Creating an area and perimeter calculator in C++ involves incorporating various object-oriented programming (OOP) concepts such as inheritance, friend functions, method overloading, method overriding, late binding, early binding, abstract class, and pure functions. Below is a comprehensive example illustrating these concepts:

```cpp

#include <iostream>

#include <cmath>

// Abstract class Shape

class Shape {

public:

virtual double area() const = 0; // Pure virtual function

virtual double perimeter() const = 0; // Pure virtual function

virtual ~Shape() {}

};

// Derived class Circle

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;

}

};

// Derived class Rectangle

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);

}

};

// Derived class Triangle

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;

}

};

// Derived class Square (Hierarchical Inheritance)

class Square : public Rectangle {

public:

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

};

// Friend Function for printing details

void printDetails(const Shape& shape) {

std::cout << "Area: " << shape.area() << std::endl;

std::cout << "Perimeter: " << shape.perimeter() << std::endl;

}

int main() {

// Single Inheritance Example

Circle circle(5);

std::cout << "Circle Details:\n";

printDetails(circle);

// Multiple Inheritance Example

Rectangle rectangle(4, 6);

std::cout << "\nRectangle Details:\n";

printDetails(rectangle);

// Hierarchical Inheritance Example

Square square(4);

std::cout << "\nSquare Details:\n";

printDetails(square);

// Friend Function Example

Triangle triangle(3, 4, 5);

std::cout << "\nTriangle Details using Friend Function:\n";

printDetails(triangle);

return 0;

}

```

a. **Inheritance:**

- Single Inheritance: Circle, Rectangle, and Triangle inherit from the base class `Shape`.

- Multiple Inheritance: Not used in this example.

- Hierarchical Inheritance: Square inherits from Rectangle.

b. **Friend Functions:**

- The `printDetails` function is a friend function that can access the private members of the `Shape` class to print details.

c. **Method Overloading and Method Overriding:**

- Method Overloading: The `area` and `perimeter` functions are overloaded in each derived class.

- Method Overriding: The `area` and `perimeter` functions in each derived class override the virtual functions in the base class.

d. **Late Binding and Early Binding:**

- Late Binding: Achieved through the use of virtual functions. The actual function to be called is determined at runtime.

- Early Binding: The binding of functions happens at compile-time in non-virtual scenarios.

e. **Abstract Class and Pure Functions:**

`Shape` is an abstract class with pure virtual functions `area` and `perimeter`. Instances of abstract classes cannot be created.

VIII.

Certainly! Let's delve into each of the topics:

a. Function overloading and operator overloading:

1. **Function Overloading:**

Function overloading is a feature in C++ that allows multiple functions with the same name but different parameters or data types to be defined within the same scope.

```cpp

int add(int a, int b) {

return a + b;

}

double add(double a, double b) {

return a + b;

}

```

In the example above, there are two functions named `add`, one for integers and another for doubles. The compiler differentiates between them based on the types and/or number of parameters.

2. **Operator Overloading:**

Operator overloading allows you to redefine the behavior of operators for user-defined data types. It enables you to use operators such as `+`, `-`, `\*` with objects of your own classes.

```cpp

class Complex {

public:

int real, imag;

Complex operator+(const Complex &obj) {

Complex result;

result.real = real + obj.real;

result.imag = imag + obj.imag;

return result;

}

};

```

Here, the `+` operator is overloaded for the `Complex` class, enabling the addition of two objects of type `Complex` using the `+` operator.

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

1. **Pass by Value:**

Pass by value involves passing the actual value of a variable to a function. Changes made to the parameter within the function do not affect the original variable.

```cpp

void increment(int num) {

num++;

}

```

- If you call `increment(x)` where `x` is a variable, `x` remains unchanged outside the function.

2. **Pass by Reference:**

Pass by reference involves passing the memory address (reference) of a variable to a function. Changes made to the parameter within the function affect the original variable.

```cpp

void incrementByReference(int &num) {

num++;

}

```

If you call `incrementByReference(x)`, changes to `num` inside the function directly affect the original variable `x`.

### c. Parameters and Arguments:

1. **Parameters:**

Parameters are variables declared in the function prototype. They act as placeholders for the actual values that will be passed into the function when it is called.

Example:

```cpp

void printSum(int a, int b) {

cout << "Sum: " << a + b << endl;

}

```

Explanation: Here, `a` and `b` are parameters, representing the values that will be provided when calling `printSum`.

2. **Arguments:**

Arguments are the actual values passed into a function when it is called. They match the parameters in order and type.

-Example:

```cpp

int main() {

printSum(3, 5);

return 0;

}

```

Explanation: In this example, `3` and `5` are arguments provided to the `printSum` function, corresponding to its parameters `a` and `b`.

In C++, a function call with arguments provides values for the parameters, and the distinction between function and operator overloading as well as pass by value and pass by reference is crucial for effective programming.

**Create a new class called *CalculateG.***

Below is the modified code for the **CalculateG** class, including the methods for multiplication, powering to square, summation, and printing out a result:

public class CalculateG {

// Earth's gravity in m/s^2

static double gravity = -9.81;

static double fallingTime = 30;

static double initialVelocity = 0.0;

static double finalVelocity;

static double initialPosition = 0.0;

static double finalPosition;

// Method for multiplication

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

return a \* b;

}

// Method for powering to square

public static double square(double a) {

return a \* a;

}

// Method for summation

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

return a + b;

}

// Method for printing out a result

public static void outline(double result) {

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

}

public static void main(String[] args) {

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

finalPosition = sum(sum(multiply(0.5, multiply(gravity, square(fallingTime))), multiply(initialVelocity, fallingTime)), initialPosition);

outline(finalPosition);

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

finalVelocity = sum(multiply(gravity, fallingTime), initialVelocity);

// Add output line for velocity

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

}

}

This code includes the methods for multiplication (multiply), powering to square (square), summation (sum), and printing out a result (outline). The main method calculates the position and velocity of an object using these methods and prints out the results.

# **Part B:**

1.

#include <iostream>

int calculateFibonacciSum(int limit) {

int sum = 0;

int fibPrev = 1;

int fibCurrent = 2;

while (fibCurrent <= limit) {

if (fibCurrent % 2 == 0) {

sum += fibCurrent;

}

int temp = fibCurrent;

fibCurrent = fibPrev + fibCurrent;

fibPrev = temp;

}

return sum;

}

int main() {

int limit = 4000000;

int result = calculateFibonacciSum(limit);

std::cout << "Sum of even-valued terms in the Fibonacci sequence up to " << limit << ": " << result << std::endl;

return 0;

}

In this program, the **calculateFibonacciSum** function takes a limit as an argument and calculates the sum of all even-valued terms in the Fibonacci sequence up to that limit. The **main** function sets the limit to four million and prints the result.

**Question Two**

Creating a GUI interface in C++ usually involves using a library or framework, and one popular choice is Qt. Below is a simple example of a C++ program using the Qt framework to create a basic GUI for checking if a number is a palindrome. Make sure you have Qt installed on your system before compiling and running the program.

#include <QtWidgets>

class PalindromeChecker : public QWidget {

Q\_OBJECT

public:

PalindromeChecker(QWidget \*parent = nullptr) : QWidget(parent) {

setupUi();

}

private slots:

void checkPalindrome() {

QString inputStr = inputLineEdit->text();

bool isPalindrome = isPalindromeNumber(inputStr.toInt());

if (isPalindrome) {

resultLabel->setText("Palindrome: Yes");

} else {

resultLabel->setText("Palindrome: No");

}

}

private:

void setupUi() {

QVBoxLayout \*layout = new QVBoxLayout(this);

QLabel \*titleLabel = new QLabel("Palindrome Checker", this);

titleLabel->setFont(QFont("Arial", 14, QFont::Bold));

titleLabel->setAlignment(Qt::AlignCenter);

QLabel \*inputLabel = new QLabel("Enter a number:", this);

inputLineEdit = new QLineEdit(this);

resultLabel = new QLabel("Palindrome: ", this);

resultLabel->setAlignment(Qt::AlignCenter);

QPushButton \*checkButton = new QPushButton("Check Palindrome", this);

connect(checkButton, &QPushButton::clicked, this, &PalindromeChecker::checkPalindrome);

layout->addWidget(titleLabel);

layout->addWidget(inputLabel);

layout->addWidget(inputLineEdit);

layout->addWidget(checkButton);

layout->addWidget(resultLabel);

setLayout(layout);

}

bool isPalindromeNumber(int number) {

int originalNumber = number;

int reversedNumber = 0;

while (number > 0) {

int digit = number % 10;

reversedNumber = reversedNumber \* 10 + digit;

number /= 10;

}

return originalNumber == reversedNumber;

}

QLineEdit \*inputLineEdit;

QLabel \*resultLabel;

};

int main(int argc, char \*argv[]) {

QApplication app(argc, argv);

PalindromeChecker palindromeChecker;

palindromeChecker.setGeometry(100, 100, 400, 200);

palindromeChecker.setWindowTitle("Palindrome Checker");

palindromeChecker.show();

return app.exec();

}

#include "main.moc"

**Question three**

#include <iostream>

const int ARRAY\_SIZE = 15;

int main() {

// Part a: Take 15 values as input and store them in an array

int inputArray[ARRAY\_SIZE];

std::cout << "Enter 15 integer values:" << std::endl;

for (int i = 0; i < ARRAY\_SIZE; ++i) {

std::cout << "Enter value " << (i + 1) << ": ";

std::cin >> inputArray[i];

}

// Part b: Print the values stored in the array and check if a specific number is present

std::cout << "\nValues stored in the array:" << std::endl;

for (int i = 0; i < ARRAY\_SIZE; ++i) {

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

}

int searchNumber;

std::cout << "\nEnter a number to search in the array: ";

std::cin >> searchNumber;

bool found = false;

int index = -1;

for (int i = 0; i < ARRAY\_SIZE; ++i) {

if (inputArray[i] == searchNumber) {

found = true;

index = i;

break;

}

}

if (found) {

std::cout << "The number found at index " << index << std::endl;

} else {

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

}

// Part c: Create another array in reverse order and print its elements

int reverseArray[ARRAY\_SIZE];

for (int i = 0; i < ARRAY\_SIZE; ++i) {

reverseArray[i] = inputArray[ARRAY\_SIZE - 1 - i];

}

std::cout << "\nValues in the new array (in reverse order):" << std::endl;

for (int i = 0; i < ARRAY\_SIZE; ++i) {

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

}

// Part d: Get the sum and product of all elements in the array

int sum = 0;

int product = 1;

for (int i = 0; i < ARRAY\_SIZE; ++i) {

sum += inputArray[i];

product \*= inputArray[i];

}

std::cout << "\n\nSum of array elements: " << sum << std::endl;

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

return 0;

}

This program takes 15 integer inputs from the user, stores them in an array, prints the array, searches for a specific number, creates a new array in reverse order, and calculates the sum and product of the array elements.