

**REPUBLIQUE DU CAMEROUN**

Paix- Travail- Patrie

**MINISTERE DE L’ENSEIGNEMENT**

**SUPERIEURE**

**L’UNIVERSITE DE BAMENDA**

**REPUBLIC OF CAMEROON**

Peace- Work- Fatherland

**MINISTRY OF HIGHER EDUCATION**

**THE UNIVERSITY OF BAMENDA**

**Faculty of SCIENCE**

**COURSE TITLE: OBJECT ORIENTED PROGRAMMING**

**COURSE CODE: CSCS4102**

**LEVEL: 400**

**ASSIGNEMNT**

Lecturer:

**VERDZEKOV EMILE TATINYUY**

**ACADEMIC YEAR 2024/2025**

**BASIC LEVEL**

1. Classes and Objects

Create a class called “Car” with properties like make, model, year, color. Implement a method “displayInfo()” that prints the details of the car.

**Definition:**

A class is a blueprint for creating objects. Objects are instances of classes and represent entities with attributes (fields) and behavior (methods).

**Solution:**

// Class representing a car with basic properties and a method  
public class Car {  
 String make, model, color; // Fields to store car details  
 int year;  
  
 // Constructor to initialize the car's properties  
 Car(String make, String model, int year, String color) {  
 this.make = make;  
 this.model = model;  
 this.year = year;  
 this.color = color;  
 }  
  
 // Method to display the car's information  
 void displayInfo() {  
 System.*out*.println("CAR DETAILS");  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*");  
 System.*out*.println("\nMake: " + make + "\nModel: " + model + "\nYear: " + year + "\nColor: " + color);  
 }  
}

public class Main {  
  
 public static void main(String[] args) {  
  
 Car car1 = new Car("Toyota", "Corolla", 2022, "White");  
 Car car2 = new Car("Honda", "Civic", 2021, "Black");  
  
 // Call the displayInfo method to print car details  
 car1.displayInfo();  
 car2.displayInfo();  
 }  
}

1. Encapsulation

Create a class “BankAccount” with private fields for accountNumber, balance, and methods to “deposit(double amount)” and “withdraw(double amount)”. Ensure that the balance cannot go below zero.

**Definition:**

Encapsulation is the concept of wrapping data (fields) and methods together in a single unit, restricting direct access to some of the object’s components.

**Solution:**

// Class representing a back account with encapsulated fields  
public class BankAccount {  
 private String accountNumber; // Private fields ensure restricted access  
 private double balance;  
  
 // Constructor to initialize the account  
 BankAccount(String accountNumber, double initialBalance) {  
 this.accountNumber = accountNumber;  
 this.balance = initialBalance;  
 }  
  
 // Public method to deposit money into the account  
 public void deposit(double amount) {  
 balance += amount;  
 System.*out*.println("Deposited: " + amount + "\nNew Balance: " + balance);  
 }  
  
 // Public method to withdraw money, ensuring the balance doesn't go below zero  
 public void withdraw(double amount) {  
 if (amount <= balance) {  
 balance -= amount;  
 System.*out*.println("Withdrawn: " + amount + "\nNew Balance: " + balance);  
 } else {  
 System.*out*.println("Insufficient balance. Current Balance: " + balance);  
 }  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 // Create a back account with an initial balance  
 BankAccount account = new BankAccount("12345678", 1000000);  
  
 // Perform deposit and withdrawal operations  
 account.deposit(500000);  
 account.withdraw(300000);  
 account.withdraw(1500000); // Will display an insufficient balance message  
 }  
}

1. Inheritance

Define a class “Animal” with a method “makeSound()”. Create a subclass “Dog” that overrides the makeSound() method to print “Bark”. Create another subclass “Cat” that prints “Meow”.

**Definition:**

Inheritance allows a class (subclass) to inherit the properties and methods of another class (superclass). This promotes code reuse.

**Solution:**

// Base class representing a generic animal  
public class Animal {  
 void makeSound() {  
 System.*out*.println("Some generic sound");  
 }  
}

// Subclass representing a dog  
public class Dog extends Animal {  
 @Override  
 void makeSound() {  
 System.*out*.println("Bark");  
 }  
}

// Subclass representing a cat  
public class Cat extends Animal {  
 @Override  
 void makeSound() {  
 System.*out*.println("Meow");  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 // Demonstrating polymorphism through inheritance  
 Animal dog = new Dog(); // Dog is treated as an Animal  
 Animal cat = new Cat(); // Cat is treated as an Animal  
  
 // Call the overridden makeSound methods  
 dog.makeSound(); // Outputs: Bark  
 cat.makeSound(); // Outputs: Meow  
 }  
}

1. Polymorphism

Create a base class “Shape” with a method “calculateArea()”. Create subclasses “Circle” and “Rectangle” that implement the “calculateArea()” method. Demonstrate polymorphism by creating a method that takes a “Shape” object and prints the area.

**Definition:**

Polymorphism allows objects of different classes to be treated as objects of a common superclass. This is typically achieved through method overriding.

**Solution:**

// Abstract base class representing a generic shape  
abstract class Shape {  
 abstract double calculateArea(); // Abstract method to calculate area  
}

// Subclass representing a circle  
class Circle extends Shape {  
 double radius;  
  
 Circle(double radius) {  
 this.radius = radius;  
 }  
  
 @Override  
 double calculateArea() {  
 return Math.*PI* \* radius \* radius; // Area of a circle formula  
 }  
}

// Subclass representing a rectangle  
class Rectangle extends Shape {  
 double length, width;  
  
 Rectangle(double length, double width) {  
 this.length = length;  
 this.width = width;  
 }  
  
 @Override  
 double calculateArea() {  
 return length \* width; // Area of a rectangle formula  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 // Create instances of shapes  
 Shape circle = new Circle(5);  
 Shape rectancle = new Rectangle(4, 6);  
  
 // Demonstrate polymorphism by passing shapes to a method  
 *printShapeArea*(circle); // Outputs: Area: 78.53981633974483  
 *printShapeArea*((rectancle)); // Outputs: Area: 24.0  
 }  
  
 // Method that takes a Shape object and prints its area  
 public static void printShapeArea(Shape shape) {  
 System.*out*.println("Area: " + shape.calculateArea());  
 }  
}

INTERMEDIATE LEVEL

1. Abstract Classes

Create an abstract class Employee with an abstract method “calculateSalary()”. Implement subclasses “FullTimeEmployee” and “PartTimeEmployee” that provide specific implementations for the “calculateSalary()” method.

**Definition:**

An abstract class is a class that cannot be instantiated. It may contain abstract methods (methods without a body) that must be implemented by its subclasses, and it can also contain concrete methods.

**Solution:**

// Abstract class representing a generic employee  
abstract class Employee {  
 String name;  
 int id;  
  
 Employee(String name, int id) {  
 this.name = name;  
 this.id = id;  
 }  
  
 // Abstract method to calculate salary  
 abstract double calculateSalary();  
}

// Subclass for full-time employees  
class FullTimeEmployee extends Employee {  
 double monthlySalary;  
  
 FullTimeEmployee(String name, int id, double monthlySalary) {  
 super(name, id);  
 this.monthlySalary = monthlySalary;  
 }  
  
 @Override  
 double calculateSalary() {  
 return monthlySalary; // Full-time employees have a fixed monthly salary  
 }  
}

// Subclass for part-time employees  
class PartTimeEmployee extends Employee {  
 double hourlyRate;  
 int hoursWorked;  
  
 PartTimeEmployee(String name, int id, double hourlyRate, int hoursWorked) {  
 super(name, id);  
 this.hourlyRate = hourlyRate;  
 this.hoursWorked = hoursWorked;  
 }  
  
 @Override  
 double calculateSalary() {  
 return hourlyRate \* hoursWorked; // Salary = hourly rate \* hours worked  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 // Create instances of employees  
 Employee fullTime = new FullTimeEmployee("Alice", 1, 50000);  
 Employee partTime = new PartTimeEmployee("Bob", 2, 20, 1200);  
  
 // Display their salaries  
 System.*out*.println("Full-time Employee Salary: $" + fullTime.calculateSalary());  
 System.*out*.println("Part-time Employee Salary: $" + partTime.calculateSalary());  
  
 }  
}

1. Interfaces

Define an interface “Playable” with a method “play()”. Create two classes “Guitar” and “Piano” that implement the Playable interface. Demonstrate calling the play() method on both classes.

**Definition:**

An interface is a contract that specifies what methods a class must implement. Unlike abstract classes, interfaces do not contain any concrete methods. A class can implement multiple interfaces.

**Solution:**

// Interface representing something that can be played  
public interface Playable {  
 void play(); // Method to be implemented by any playable object  
}

// Class for a Guitar  
class Guitar implements Playable {  
 @Override  
 public void play() {  
 System.*out*.println("Strumming the guitar...");  
 }  
}

// Class for a Piano  
class Piano implements Playable {  
 @Override  
 public void play() {  
 System.*out*.println("Playing the Piano...");  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 // Create instances of instruments  
 Playable guitar = new Guitar();  
 Playable piano = new Piano();  
  
 // Call the play method for both  
 guitar.play(); // Outputs: Strumming the guitar...  
 piano.play(); // Outputs: Playing the piano...  
 }  
}

1. Composition

Create a class Library that contains a list of “Book” objects. Implement methods to add a book, remove a book, and display all books in the library.

**Definition:**

Composition is a design principle in which a class is made up of other objects (has-a relationship). This allows for modular and reusable code.

**Solution:**

// Class representing a Book  
class Book {  
 String title, author;  
  
 Book(String title, String author) {  
 this.author = author;  
 this.title = title;  
 }  
  
 @Override  
 public String toString() {  
 return "\"" + title + "\" by " + author;  
 }  
}

import java.util.ArrayList;  
  
// Class representing a Library that contains a list of books  
class Library {  
 private final ArrayList<Book> books = new ArrayList<>(); // List to store books  
  
 // Method to add a book to the library  
 public void addBook(Book book) {  
 books.add(book);  
 System.*out*.println(book + " added to the library.");  
 }  
  
 // Method to remove a book from the library  
 public void removeBook(Book book) {  
 if (books.remove(book)) {  
 System.*out*.println(book + " removed from the library.");  
 } else {  
 System.*out*.println(book + " is not in the library.");  
 }  
 }  
  
 // Method to display all books in the library  
 void displayBooks() {  
 System.*out*.println("Library Books:");  
 for (Book book : books) {  
 System.*out*.println("- " + book);  
 }  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 // Create a library  
 Library library = new Library();  
  
 // Create books  
 Book book1 = new Book("1984", "George Orwell");  
 Book book2 = new Book("To Kill a Mockingbird", "Harper Lee");  
 Book book3 = new Book("The Great Gatsby", "F. Scott Fritzgerald");  
  
 // Add books to the library  
 library.addBook(book1);  
 library.addBook(book2);  
 library.addBook(book3);  
  
 // Display all books  
 library.displayBooks();  
  
 // Remove a book  
 library.removeBook(book2);  
  
 // Display remaining books  
 library.displayBooks();  
 }  
}

ADVANCED LEVEL

1. Exception Handling

Modify the “BankAccount” class to throw an exception when attempting to withdraw an amount greater than the current balance. Create a custom exception class “InsufficientFundsException”.

**Definition:**

Exception handling is the process of responding to runtime errors (exceptions) in a program. This ensures that the program can handle errors gracefully without crashing.

**Solution:**

// Custom exception for insufficient funds  
class InsufficientFundsException extends Exception {  
 public InsufficientFundsException(String message) {  
 super(message); // Pass the error message to the Exception superclass  
 }  
}

// Bank account class with exception handling  
class BankAccount {  
 private String accountNumber;  
 private double balance;  
  
 public BankAccount(String accountNumber, double initialBalance) {  
 this.accountNumber = accountNumber;  
 this.balance = initialBalance;  
 }  
  
 // Deposit money into the account  
 public void deposit(double amount) {  
 balance += amount;  
 System.*out*.println("Deposited: $" + amount + ", New Balance: $" + balance);  
 }  
  
 // Withdraw money with exception handling  
 public void withdraw(double amount) throws InsufficientFundsException {  
 if (amount > balance) {  
 throw new InsufficientFundsException("Insufficient funds. Current balance: $" + balance);  
 }  
  
 balance -= amount;  
 System.*out*.println("Withdrawn: $" + amount + ", New Balance: $" + balance);  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 BankAccount account = new BankAccount("987654321", 100000);  
  
 try {  
 account.deposit(50000);  
 account.withdraw(30000); // Successful withdrawal  
 account.withdraw(150000); // Will throw InsufficientFundsException  
 } catch (InsufficientFundsException e) {  
 System.*out*.println("Error: " + e.getMessage());  
 }  
 }  
}

1. Static vs Instance Members

Create a class “Student” with a static variable “studentCount” that keeps track of how many Student objects have been created. Implement a constructor that increments this count each time a new student is created.

**Definition:**

Static members belong to the class, not an instance. They are shared across all instances of the class, while instance members are unique to each other.

**Solution:**

// Class representing a Student  
class Student {  
 static int *studentCount* = 0; // Static variable to track the number of students  
 String name;  
  
 // Constructor that increments the student count for every new student  
 public Student(String name) {  
 this.name = name;  
 *studentCount*++;  
 }  
  
 // Static method to get the total student count  
 public static int getStudentCount() {  
 return *studentCount*;  
 }  
}

public class Main {  
  
 public static void main(String[] args) {  
 // Create student objects  
 Student s1 = new Student("Alice");  
 Student s2 = new Student("Bob");  
 Student s3 = new Student("Charlie");  
   
 // Display the total number of students using a static method  
 System.*out*.println("Total Students: " + Student.*getStudentCount*());  
 }  
}

1. Singleton Patterns

Implement the Singleton pattern in a class “Configuration” that provides application configuration settings. Ensure that only one instance of Configuration can exist.

**Definition:**

The Singleton pattern ensures that a class has only one instance throughout the application and provides a global point of access to that instance.

**Solution:**

// Singleton class for Configuration  
class Configuration {  
 private static Configuration *instance*; // Static instance of the class  
 private String setting;  
  
 // Private constructor prevents external instantiation  
 private Configuration() {  
 setting = "Default Configuration";  
 }  
  
 // Public method to provide access to the single instance  
 public static Configuration getInstance() {  
 if (*instance* == null) {  
 *instance* = new Configuration();  
 }  
 return *instance*;  
 }  
  
 // Getter and Setter for the configuration setting  
 public String getSetting() {  
 return setting;  
 }  
  
 public void setSetting(String setting) {  
 this.setting = setting;  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 // Get the single instance of Configuration  
 Configuration config1 = Configuration.*getInstance*();  
 System.*out*.println("Config Setting: " + config1.getSetting());  
  
 // Change the setting using one instance  
 config1.setSetting("Updated Configuration");  
  
 // Access the same instance and verify the change  
 Configuration config2 = Configuration.*getInstance*();  
 System.*out*.println("Config Setting: " + config2.getSetting());  
 }  
}

BONUS QUESTION

1. Real-world Scenario

Design a simple e-commerce system with classes like Product, Customer, and Order. Implement relationships between these classes and demonstrate how they interact with each other. Include methods to add products to an order and calculate the total cost.

**Solution:**

// Class representing a product  
class Product {  
 String name;  
 double price;  
  
 public Product(String name, double price) {  
 this.name = name;  
 this.price = price;  
 }  
  
 @Override  
 public String toString() {  
 return name + " (XAF" + price + ")";  
 }  
}

// Class representing a customer  
class Customer {  
 String name;  
  
 public Customer(String name) {  
 this.name = name;  
 }  
}

import java.util.ArrayList;  
  
// Class representing an order  
class Order {  
 Customer customer;  
 ArrayList<Product> products = new ArrayList<>(); // List of products in the order  
  
 public Order(Customer customer) {  
 this.customer = customer;  
 }  
  
 // Add a product to the order  
 public void addProduct(Product product) {  
 products.add(product);  
 System.*out*.println(product.name + " added to the order.");  
 }  
  
 // Calculate the total cost of the order  
 public double calculateTotal() {  
 double total = 0;  
  
 for (Product product : products) {  
 total += product.price;  
 }  
 return total;  
 }  
  
 // Display the order details  
 public void displayOrder() {  
 System.*out*.println("Order for " + customer.name + ":");  
  
 for (Product product : products) {  
 System.*out*.println("- " + product);  
 }  
  
 System.*out*.println("Total Cost: XAF" + calculateTotal());  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 // Create products  
 Product product1 = new Product("Rice", 39000);  
 Product product2 = new Product("Meat", 3000);  
 Product product3 = new Product("Fish", 1500);  
  
 // Create a customer  
 Customer favouriteCustomer = new Customer("Georgee Flash");  
  
 // Create an order and add products  
 Order order = new Order(favouriteCustomer);  
  
 order.addProduct(product1);  
 order.addProduct(product2);  
 order.addProduct(product3);  
  
 // Display the order details  
 order.displayOrder();  
 }  
}