

**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**

**Name:**

**NDZISHEPNGONG GEORGE TARLA**

**Matriculation Number:**

**UBa22S0477**

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.

**Solution:**

public class Car {  
 String make, model, color;  
 int year;  
  
 Car(String make, String model, int year, String color) {  
 this.make = make;  
 this.model = model;  
 this.year = year;  
 this.color = color;  
 }  
  
 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 car = new Car("Tesla", "X", 2009, "Black");  
  
 car.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.

**Solution:**

public class BankAccount {  
 private String accountNumber;  
 private double balance;  
  
 BankAccount(String accountNumber, double initialBalance) {  
 this.accountNumber = accountNumber;  
 this.balance = initialBalance;  
 }  
  
 public void deposit(double amount) {  
 balance += amount;  
 System.*out*.println("Deposited: " + amount + "\nNew Balance: " + balance);  
 }  
  
 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 System.*out*.println("Withdrawn: " + amount + "\nNew Balance: " + balance);  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);  
 }  
 }  
}

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”.

**Solution:**

public class Animal {  
 void makeSound() {  
 System.*out*.println("Some generic sound");  
 }  
}

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

public class Cat extends Animal {  
 @Override  
 void makeSound() {  
 System.*out*.println("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.

**Solution:**

abstract class Shape {  
 abstract double calculateArea();  
  
 void printShapeArea(Shape shape) {  
 System.*out*.println("Area: " + shape.calculateArea());  
 }  
}

public class Circle extends Shape {  
 double radius;  
  
 Circle(double radius) {  
 this.radius = radius;  
 }  
  
 @Override  
 double calculateArea() {  
 return Math.*PI* \* radius \* radius;  
 }  
}

public class Rectangle extends Shape {  
 double length, width;  
  
 Rectangle(double length, double width) {  
 this.length = length;  
 this.width = width;  
 }  
  
 @Override  
 double calculateArea() {  
 return length \* width;  
 }  
}

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.

**Solution:**

abstract class Employee {  
 String name;  
  
 Employee(String name) {  
 this.name = name;  
 }  
  
 abstract double calculateSalary();  
}

public class FullTimeEmployee extends Employee {  
 double monthlySalary;  
  
 FullTimeEmployee(String name, double monthlySalary) {  
 super(name);  
 this.monthlySalary = monthlySalary;  
 }  
  
 @Override  
 double calculateSalary() {  
 return monthlySalary;  
 }  
}

public class PartTimeEmployee extends Employee {  
 double hourlyRate;  
 int hoursWorked;  
  
 PartTimeEmployee(String name, double hourlyRate, int hoursWorked) {  
 super(name);  
 this.hourlyRate = hourlyRate;  
 this.hoursWorked = hoursWorked;  
 }  
  
 @Override  
 double calculateSalary() {  
 return hourlyRate \* hoursWorked;  
 }  
}

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.

**Solution:**

public interface Playable {  
 void play();  
}

public class Guitar implements Playable {  
 @Override  
 public void play() {  
 System.*out*.println("Playing the Guitar");  
 }  
}

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

public class Main {  
 public static void main(String[] args) {  
  
 Guitar guitar = new Guitar();  
 Piano piano = new Piano();  
  
  
 guitar.play();  
 piano.play();  
 }  
}

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.

**Solution:**

public 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;  
  
public class Library {  
 private final ArrayList<Book> books = new ArrayList<>();  
  
 void addBook(Book book) {  
 books.add(book);  
 }  
  
 void removeBook(Book book) {  
 books.remove(book);  
 }  
  
 void displayBooks() {  
 for (Book book : books) {  
 System.*out*.println(book);  
 }  
 }  
}

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”.

**Solution:**

public class InsufficientFundsException extends Exception {  
 public InsufficientFundsException(String message) {  
 super(message);  
 }  
}

public class BankAccount {  
 private String accountNumber;  
 private double balance;  
  
 BankAccount(String accountNumber, double initialBalance) {  
 this.accountNumber = accountNumber;  
 this.balance = initialBalance;  
 }  
  
 public void withdraw(double amount) throws InsufficientFundsException {  
 if (amount > balance) {  
 throw new InsufficientFundsException("Insufficient funds. Available balance: " + balance);  
 }  
  
 balance -= amount;  
 System.*out*.println("Withdrawn: " + amount + "\nNew Balance: " + balance);  
 }  
}

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.

**Solution:**

public class Student {  
 private static int *studentCount* = 0;  
 private String name;  
   
 Student(String name) {  
 this.name = name;  
 *studentCount*++;  
 }  
   
 static int getStudentCount() {  
 return *studentCount*;  
 }  
}

1. Design Patterns

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

**Solution:**

public class Configuration {  
 private static Configuration *instance*;  
 private Configuration() {}  
   
 public static Configuration getInstance() {  
 if (*instance* == null) {  
 *instance* = new Configuration();  
 }  
 return *instance*;  
 }  
}

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:**

public class Product {  
 String name;  
 double price;  
  
 Product(String name, double price) {  
 this.name = name;  
 this.price = price;  
 }  
}

public class Customer {  
 String name;  
  
 Customer(String name) {  
 this.name = name;  
 }  
}

import java.util.ArrayList;  
  
public class Order {  
 private ArrayList<Product> products = new ArrayList<>();  
  
 void addProduct(Product product) {  
 products.add(product);  
 }  
  
 double calculateTotal() {  
 double total = 0;  
  
 for (Product product : products) {  
 total += product.price;  
 }  
 return total;  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 Product product1 = new Product("Rice", 39000);  
 Product product2 = new Product("Meat", 3000);  
 Product product3 = new Product("Fish", 1500);  
  
 Customer favouriteCustomer = new Customer("Georgee Flash");  
  
 Order order = new Order();  
  
 order.addProduct(product1);  
 order.addProduct(product2);  
 order.addProduct(product3);  
  
 System.*out*.println(order.calculateTotal()); // 43500.0  
 }  
}