Name: **Enrolment No:**	

UPES

End Semester Examination, May 2025

Course: Object Oriented Programming **Semester: IV**
Program: BTech_CS_H_CSE_All/BCA **Time : 03 hrs.**
Course Code: CSEG2020 **Max. Marks: 100**

Instructions:

SECTION A
(5Qx4M=20Marks)

S. No.		**Marks**	**CO**

Q1	How can Encapsulation be achieved in Java? Discuss with example.
	Encapsulation is the process of wrapping data (variables) and methods into a single unit called class. It helps in data hiding and improves modularity. How to achieve Encapsulation:
	a) Declare class variables as private.
	b) Provide public getter and setter methods to access and update the value of private variables.
	Example:
	`class Student {`
	`    private String name;`
	`    private int age;`
	
	`    // Getter method`
	`    public String getName() {`
	`        return name;`
	`    }`
	
	`    // Setter method`
	`    public void setName(String name) {`
	`        this.name = name;`
	`    }`
	
	`    // Getter and Setter for age`
	`    public int getAge() {`
	`        return age;`
		4	**CO1**

```
    }

    public void setAge(int age) {
        this.age = age;
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student();
        s.setName("John");
        s.setAge(20);
        System.out.println("Name: " + s.getName());
        System.out.println("Age: " + s.getAge());
    }
}
```

Q2

What is method overloading? Write a small example to illustrate it.

Ans: Method Overloading means defining multiple methods with the same name, but different parameter lists within a class.

```
class Calculator {
    // Method with 2 int parameters
    int add(int a, int b) {
        return a + b;
    }

    // Method with 3 int parameters
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method with 2 double parameters
    double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Sum1: " + calc.add(5, 10));
```

4

CO2

```
        System.out.println("Sum2: " + calc.add(2, 4, 6));
        System.out.println("Sum3: " + calc.add(3.5, 4.5));
    }
}
```

Q3

Write a Java program to find the sum of elements of a one-dimensional array.

```
public class ArraySum {
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40, 50};
        int sum = 0;

        for (int i = 0; i < numbers.length; i++) {
            sum += numbers[i];
        }

        System.out.println("Sum of array elements: " + sum);
    }
}
```

4

CO2

Q4

Define the importance of “this” keyword in Java. Show its 2 uses with examples.
Importance of this keyword:

a) Refers to the current class object.
b) Resolves naming conflicts between instance variables and parameters.
c) Used to invoke current class methods and constructors.

```
class Student {
    String name;
    int age;

    Student() {
        this("Unknown", 18); // Calls parameterized constructor
    }

    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    void show() {
        System.out.println(name + " is " + age + " years old.");
```

4

CO3

```
    }
}
```

Q5

Write a Java program to create and execute a thread by implementing Runnable interface.

```
class MyThread implements Runnable {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Running thread: " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        Thread thread = new Thread(t1);  // Creating thread using Runnable
        thread.start();  // Starting thread
    }
}
```

4 **CO4**

SECTION B
(4Qx10M= 40 Marks)

Q6

Create a Java package named mypack containing a class Calculator with add() and subtract() methods. Write a program to use this package.

```
// File: mypack/Calculator.java
package mypack;

public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public int subtract(int a, int b) {
        return a - b;
```

10 **CO3**

```
    }
}

// File: Main.java
import mypack.Calculator;

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Addition: " + calc.add(10, 5));
        System.out.println("Subtraction: " + calc.subtract(10, 5));
    }
}
```

Q7

Discuss the difference between checked & unchecked exceptions in Java. Write a program that shows the difference between “throw” and “throws”.

Checked Exceptions:

Checked At: These exceptions are verified at compile time, meaning the Java compiler requires the programmer to either handle them using a try-catch block or declare them in the method signature using the throws keyword.

Handling Requirement: Handling is mandatory; failure to address a checked exception results in a compilation error, ensuring potential issues are accounted for during development.

Examples: Common examples include IOException, which occurs during file operations, and SQLException, which arises during database interactions.

Inheritance: Checked exceptions inherit directly from the Exception class but not from RuntimeException, distinguishing them from unchecked exceptions.

Purpose: They are designed for recoverable conditions, such as missing files or database connection failures, allowing the program to handle and potentially recover from these issues gracefully.

Unchecked Exceptions:

Checked At: These exceptions are checked at runtime, meaning the compiler does not enforce any handling or declaration, allowing the program to compile even if they are unaddressed.

10

CO2

Handling Requirement: Handling is optional; programmers can choose to handle them with try-catch blocks, but the program will run without explicit handling, often leading to runtime errors if uncaught.

Examples: Typical examples include ArithmeticException, such as division by zero, and NullPointerException, which occurs when accessing a null object reference.

Inheritance: Unchecked exceptions inherit from RuntimeException, a subclass of Exception, which differentiates them from checked exceptions.

Purpose: They typically indicate programming errors or unexpected conditions, such as logical mistakes or invalid data, which should ideally be fixed during development to prevent runtime failures.

```
public class ExceptionExample {

    // throws keyword: declaring a checked exception
    static void checkAge(int age) throws Exception {
        if (age < 18) {
            throw new Exception("Age must be 18 or above"); // throw keyword: throwing
an exception manually
        } else {
            System.out.println("Eligible to vote.");
        }
    }

    public static void main(String[] args) {
        try {
            checkAge(16);
        } catch (Exception e) {
            System.out.println("Exception caught: " + e.getMessage());
        }

        // Example of unchecked exception
        int a = 10, b = 0;
        try {
            int c = a / b;
        } catch (ArithmeticException e) {
            System.out.println("Unchecked exception: " + e);
        }
    }
}
```

Q8	Create an ArrayList to store Faculty objects with attributes like name, department, and experienceYears. Add four faculty members, update the department of one faculty, remove another faculty by their name, and display all faculty details.	**10**	**C O3**

```java
import java.util.ArrayList;

class Faculty {
    String name, department;
    int experienceYears;

    Faculty(String name, String department, int experienceYears) {
        this.name = name;
        this.department = department;
        this.experienceYears = experienceYears;
    }

    public String toString() {
        return name + " | " + department + " | " + experienceYears + " years";
    }
}

public class FacultyManager {
    public static void main(String[] args) {
        ArrayList<Faculty> list = new ArrayList<>();

        // Adding faculty
        list.add(new Faculty("Dr. Sharma", "CS", 12));
        list.add(new Faculty("Dr. Mehta", "IT", 10));
        list.add(new Faculty("Dr. Verma", "ECE", 8));
        list.add(new Faculty("Dr. Singh", "ME", 15));

        // Update department of "Dr. Sharma"
        for (Faculty f : list) {
            if (f.name.equals("Dr. Sharma")) {
                f.department = "Cyber Security";
                break;
            }
        }

        // Remove "Dr. Verma"
        list.removeIf(f -> f.name.equals("Dr. Verma"));
        // Display faculty
        for (Faculty f : list) {
            System.out.println(f);
        }
    }
}
```

Q9

Write a **JDBC** program to perform the following operations on a Movie table (columns: id, title, rating):

a) Insert a new movie record into the table.
b) Assuming multiple movie entries exist, fetch all records and display the average rating of all movies.

```
import java.sql.*;

public class MovieDB {
    public static void main(String[] args) {
        try {
            Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb", "root",
"password");

            // Insert a new movie
            String insert = "INSERT INTO Movie (id, title, rating) VALUES (?, ?, ?)";
            PreparedStatement ps = con.prepareStatement(insert);
            ps.setInt(1, 101);
            ps.setString(2, "Inception");
            ps.setDouble(3, 8.8);
            ps.executeUpdate();

            // Fetch all movies and calculate average rating
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM Movie");

            int count = 0;
            double sum = 0;

            System.out.println("Movies List:");
            while (rs.next()) {
                System.out.println(rs.getInt("id") + " " + rs.getString("title") + " " +
rs.getDouble("rating"));
                sum += rs.getDouble("rating");
```

10

CO4

```
                count++;
            }

            System.out.println("Average Rating: " + (count > 0 ? (sum / count) : 0));

            con.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

OR

Create a **Java Swing application** for Movie Registration with the following features:

a) Input fields: Movie Title, Director, Rating
b) A Submit button that, when clicked, displays the entered movie details on the screen.

```
import javax.swing.*;
import java.awt.event.*;

public class MovieRegistration extends JFrame implements ActionListener {
    JTextField titleField, directorField, ratingField;
    JButton submitBtn;
    JTextArea outputArea;

    public MovieRegistration() {
        setTitle("Movie Registration");
        setSize(350, 300);
        setLayout(null);

        JLabel l1 = new JLabel("Title:");
        l1.setBounds(30, 30, 80, 25);
        add(l1);

        titleField = new JTextField();
        titleField.setBounds(100, 30, 200, 25);
        add(titleField);

        JLabel l2 = new JLabel("Director:");
        l2.setBounds(30, 70, 80, 25);
        add(l2);
        directorField = new JTextField();
        directorField.setBounds(100, 70, 200, 25);
        add(directorField);

        JLabel l3 = new JLabel("Rating:");
        l3.setBounds(30, 110, 80, 25);
        add(l3);

        ratingField = new JTextField();
        ratingField.setBounds(100, 110, 200, 25);
        add(ratingField);

        submitBtn = new JButton("Submit");
        submitBtn.setBounds(100, 150, 100, 30);
        submitBtn.addActionListener(this);
        add(submitBtn);

        outputArea = new JTextArea();
        outputArea.setBounds(30, 190, 270, 60);
        add(outputArea);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        String title = titleField.getText();
        String director = directorField.getText();
        String rating = ratingField.getText();

        outputArea.setText("Movie Registered:\nTitle: " + title + "\nDirector: " + director
+ "\nRating: " + rating);
    }

    public static void main(String[] args) {
        new MovieRegistration();
    }
}
```

SECTION-C
(2Qx20M=40 Marks)

Q10	Explain and provide examples of the following types of nested classes in Java: a) Member Inner Class b) Anonymous Inner Class c) Local Inner Class d) Static Nested Class For each type, explain its characteristics, how it can access the outer class’s members, and provide a short code snippet demonstrating its usage. Discuss the key differences between static and non-static nested classes.	**20**	**C O3**

Solution:

a) Member Inner Class

Characteristics:

- Defined inside a class, but outside any method.
- Can access all members (even private) of the outer class.
- Requires an instance of the outer class to be instantiated.

```
class Outer {
   private String message = "Hello from Outer class!";

   class Inner {
      void showMessage() {
         System.out.println(message); // Accessing outer class member
      }
   }

   public static void main(String[] args) {
      Outer outer = new Outer();
      Outer.Inner inner = outer.new Inner();
      inner.showMessage();
   }
}
```

b) Anonymous Inner Class

Characteristics:

- A class without a name.
- Used to implement interfaces or extend classes on the fly.
- Typically defined inside a method or block.

```
abstract class Greeting {
   abstract void sayHello();
}

public class Demo {
   public static void main(String[] args) {
      Greeting g = new Greeting() {
         void sayHello() {
                System.out.println("Hello from Anonymous Inner Class!");
            }
        };
        g.sayHello();
    }
}
```

c) Local Inner Class
Characteristics:

- Declared within a method.
- Can access final or effectively final local variables of the method.
- Cannot be static.

```
public class Outer {
    void display() {
        String msg = "Local Inner Class";
        class Local {
            void print() {
                System.out.println(msg);
            }
        }
        Local l = new Local();
        l.print();
    }

    public static void main(String[] args) {
        new Outer().display();
    }
}
```

d) Static Nested Class
Characteristics:

- Defined with the static keyword inside the outer class.
- Does not have access to non-static members of the outer class.
- Can be instantiated without an outer class object

```
class Outer {
        static int data = 100;

        static class StaticInner {
            void show() {
                System.out.println("Data: " + data);
            }
        }

        public static void main(String[] args) {
            Outer.StaticInner obj = new Outer.StaticInner();
            obj.show();
        }
    }
```

Static vs Non-Static Nested Classes (Key Differences)

Feature	Static Nested Class	Non-Static (Member) Inner Class
Access outer class members	Only static members	Both static and instance members
Instantiation	Outer.StaticInner obj = new Outer.StaticInner();	Outer.Inner obj = outer.new Inner();
Requires outer class instance	No	Yes
Can be declared static	Yes	No

OR

Write a Java program that manages movie records in a file using FileWriter and FileReader. The program should:
(1) prompt the user to enter movie details (ID, title, and rating),
(2) store these records in "movies.txt" in the format "ID:Title:Rating" (appending new entries without overwriting existing data), and
(3) display all stored movies by reading the file.

Implement proper exception handling for file operations. The program should first write the user's movie entry and then display all records including the newly added one, with each movie's details properly formatted in the console output.

```
import java.io.*;
import java.util.Scanner;

public class MovieFileManager {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String fileName = "movies.txt";
        try {
            // Step 1: Accept user input
            System.out.print("Enter Movie ID: ");
            int id = sc.nextInt();
            sc.nextLine(); // Consume newline

            System.out.print("Enter Movie Title: ");
            String title = sc.nextLine();

            System.out.print("Enter Movie Rating: ");
            double rating = sc.nextDouble();

            // Step 2: Append to file
            FileWriter fw = new FileWriter(fileName, true); // Append mode
            BufferedWriter bw = new BufferedWriter(fw);
            bw.write(id + ":" + title + ":" + rating);
            bw.newLine();
            bw.close();

            // Step 3: Read and display all records
            System.out.println("\nAll Movies:");
            FileReader fr = new FileReader(fileName);
            BufferedReader br = new BufferedReader(fr);
            String line;

            while ((line = br.readLine()) != null) {
                String[] parts = line.split(":");
                System.out.println("ID: " + parts[0] + ", Title: " + parts[1] + ", Rating: " +
parts[2]);
            }
            br.close();

        } catch (IOException e) {
            System.out.println("File error: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Input error: " + e.getMessage());
        } finally {
            sc.close();
        }
    }
```

	}		
Q11	a) Write a Java program to show multilevel and Hierarchical inheritance in Java. b) Why is multiple inheritance not supported directly in Java? How can we implement it? Discuss with examples.	**20**	**C O2**

```
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}

class Puppy extends Dog {
    void weep() {
        System.out.println("Weeping...");
    }

    public static void main(String[] args) {
        Puppy p = new Puppy();
        p.eat();
        p.bark();
        p.weep();
    }
}

class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}

class Cat extends Animal {
    void meow() {
        System.out.println("Meowing...");
    }
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();
        d.bark();

        Cat c = new Cat();
        c.eat();
        c.meow();
    }
}
```

Why Multiple Inheritance is Not Supported Directly in Java? How to Achieve It?
Reason:
Ambiguity Problem (Diamond Problem): If two superclasses have methods with the same signature, the compiler cannot decide which one to use.
Solution: Use Interfaces to achieve multiple inheritance.

Example Using Interfaces:

```
interface A {
    void show();
}

interface B {
    void display();
}

class C implements A, B {
    public void show() {
        System.out.println("Show from A");
    }

    public void display() {
        System.out.println("Display from B");
    }

    public static void main(String[] args) {
        C obj = new C();
        obj.show();
        obj.display();
    }
}
```