

1. Monthly Sales Analysis Program (1D Array)

Concept:

A 1D array stores 12 months of sales data. Using loops and conditionals, we can calculate totals, averages, and comparisons.

Program:

```
import java.util.Scanner;

class SalesAnalysis {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        double[] sales = new double[12];
        double total = 0, avg;
        int highMonth = 0, lowMonth = 0;

        // Accept sales data
        for (int i = 0; i < 12; i++) {
            System.out.print("Enter sales for month " + (i + 1) + ": ");
            sales[i] = sc.nextDouble();
            total += sales[i];
        }

        // Calculate average
        avg = total / 12;

        // Find highest and lowest
        for (int i = 1; i < 12; i++) {
            if (sales[i] > sales[highMonth]) highMonth = i;
            if (sales[i] < sales[lowMonth]) lowMonth = i;
        }

        // Display results
        System.out.println("\nTotal Sales: " + total);
        System.out.println("Average Sales: " + avg);
        System.out.println("Highest Sales Month: " + (highMonth + 1));
        System.out.println("Lowest Sales Month: " + (lowMonth + 1));
    }
}
```

```

        // Display sales above average
        System.out.println("\nSales above average:");
        for (int i = 0; i < 12; i++)
            if (sales[i] > avg)
                System.out.println("Month " + (i + 1) + ": " + sales[i]);
    }
}

```

Explanation:

- Uses arrays, loops, and conditionals.
 - Demonstrates data analysis using basic Java structures.
-

2. Arrays of Varying Lengths & Arrays as Vectors

Arrays of varying lengths (jagged arrays):

- Each row can have a different length.

Example:

```

int[][] jagged = {
    {10, 20},
    {30, 40, 50},
    {60}
};

```

Output:

```

10 20
30 40 50
60

```

Arrays as Vectors:

- Java's Vector class is a dynamic array that grows automatically.

Example:

```

import java.util.*;

```

```

class VectorDemo {
    public static void main(String[] args) {
        Vector<Integer> v = new Vector<>();
        v.add(10); v.add(20); v.add(30);
        System.out.println(v);
    }
}

```

Key Difference:

- Arrays are fixed-size; Vectors can grow dynamically.
- Jagged arrays allow variable-length rows.

3. (a) String Comparison & Searching

Methods:

- equals() → checks content equality
- compareTo() → lexicographic comparison
- indexOf() → finds index of substring

Example:

```

String s1 = "Apple", s2 = "Apples";
System.out.println(s1.equals(s2));    // false
System.out.println(s1.compareTo(s2)); // -1 (s1 < s2)
System.out.println(s2.indexOf("ple")); // 2

```

(b) String Modification & Conversion

```

class StringOps {
    public static void main(String[] args) {
        String str = "Hello";
        str = str.replace("Hello", "Welcome");
        System.out.println(str);

        int num = 100;
        String s = String.valueOf(num); // int → String
        int x = Integer.parseInt("200"); // String → int
    }
}

```

```
    System.out.println(s + " " + x);
}
}
```

4. Text Processing Tool using String and StringBuffer

```
class TextProcessor {
    public static void main(String[] args) {
        // String constructors
        String s1 = "Hello";           // from literal
        char[] ch = {'J', 'a', 'v', 'a'};
        String s2 = new String(ch);    // from char array
        byte[] b = {80, 81, 82};
        String s3 = new String(b);     // from byte array
        System.out.println(s1 + " " + s2 + " " + s3);

        // StringBuffer
        StringBuffer sb = new StringBuffer("Java");
        sb.append(" Programming");
        sb.insert(5, "Language ");
        sb.replace(0, 4, "Learn");
        sb.reverse();
        System.out.println("Modified: " + sb);
        System.out.println("Capacity: " + sb.capacity());
    }
}
```

Why StringBuffer?

- Mutable → can change contents without creating new objects.
 - Thread-safe and efficient for repeated text modification.
-

5. Protected Members in Inheritance

```
class BankAccount {
    protected double balance;
    protected int accountNumber;

    BankAccount(double b, int acc) {
        balance = b;
        accountNumber = acc;
    }
}
```

```

    }

    protected void displayAccountDetails() {
        System.out.println("Account: " + accountNumber + ", Balance: " + balance);
    }
}

class SavingsAccount extends BankAccount {
    private double interestRate;

    SavingsAccount(double b, int acc, double rate) {
        super(b, acc);
        interestRate = rate;
    }

    void showDetails() {
        displayAccountDetails(); // accessing protected method
        System.out.println("Interest Rate: " + interestRate);
    }
}

class Main {
    public static void main(String[] args) {
        SavingsAccount s = new SavingsAccount(5000, 101, 5.5);
        s.showDetails();
    }
}

```

Explanation:

- protected allows subclass access but prevents access from unrelated classes.

6. Constructor Execution in Inheritance

Single Inheritance Example:

```

class A {
    A() { System.out.println("A Constructor"); }
}
class B extends A {
    B() { System.out.println("B Constructor"); }
}

```

```
public class Test { public static void main(String[] args) { new B(); } }
```

Output:

A Constructor

B Constructor

Multilevel Example:

```
class X { X() { System.out.println("X"); } }  
class Y extends X { Y() { System.out.println("Y"); } }  
class Z extends Y { Z() { System.out.println("Z"); } }  
public class Demo { public static void main(String[] args) { new Z(); } }
```

Output:

X

Y

Z

Constructors execute from parent to child.

7. Method Overriding and Superclass Reference

```
class Animal {  
    void sound() { System.out.println("Generic sound"); }  
}  
class Dog extends Animal {  
    void sound() { System.out.println("Bark"); }  
}  
  
public class TestPoly {  
    public static void main(String[] args) {  
        Animal a = new Dog(); // superclass ref → subclass object  
        a.sound();           // Runtime polymorphism  
    }  
}
```

Explanation:

- Overriding → subclass redefines superclass method.
- The object type decides which method runs at runtime.

8. Abstract Class Example

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

class Dog extends Animal {
    void sound() { System.out.println("Bark"); }
}

class Cat extends Animal {
    void sound() { System.out.println("Meow"); }
}

public class Main {
    public static void main(String[] args) {
        Animal d = new Dog();
        Animal c = new Cat();
        d.sound();
        c.sound();
    }
}
```

Explanation:

- Abstract class can't be instantiated.
- It defines a template; subclasses implement behavior.

9. Using super Keyword

```
class Parent {
    int num = 10;
    Parent() { System.out.println("Parent Constructor"); }
    void display() { System.out.println("Parent num = " + num); }
}

class Child extends Parent {
    int num = 20;
    Child() {
        super(); // invokes parent constructor
    }
}
```

```

        System.out.println("Child Constructor");
    }
    void display() {
        super.display();        // call parent method
        System.out.println("Child num = " + num);
        System.out.println("Parent num = " + super.num); // access parent variable
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.display();
    }
}

```

10. Final Keyword Usage

```

final class Vehicle { // cannot be extended
    final int speedLimit = 100; // cannot be changed
    final void display() { System.out.println("Speed Limit: " + speedLimit); }
}

```

// class Car extends Vehicle { } // ❌ Error: cannot inherit from final class

```

public class TestFinal {
    public static void main(String[] args) {
        Vehicle v = new Vehicle();
        v.display();
    }
}

```

Explanation:

- final variable → constant.
- final method → cannot be overridden.
- final class → cannot be inherited.

Used to enforce immutability and security in design.

MODULE 4

Excellent — these are 8-mark Java long-answer questions from Interfaces, Packages, and File I/O. Below are detailed, exam-ready explanations and programs for each.

1. Concept of Interfaces in Java & Multiple Inheritance

Concept:

An interface in Java is a collection of abstract methods and constants.

It defines what a class should do, not how it should do it.

Key Points:

- Declared using the interface keyword.
- Implemented using implements.
- Supports multiple inheritance because a class can implement multiple interfaces.

Example (Multiple Inheritance):

```
interface A {  
    void show();  
}  
  
interface B {  
    void display();  
}  
  
// Class implementing both interfaces  
class C implements A, B {  
    public void show() {  
        System.out.println("Interface A method");  
    }  
    public void display() {  
        System.out.println("Interface B method");  
    }  
}
```

```

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

```

Explanation:

Java doesn't support multiple inheritance through classes to avoid ambiguity, but it achieves it safely using interfaces, since they only declare methods (no state).

2. Multiple and Nested Inheritance using Interfaces

Multiple Inheritance:

A single class can implement more than one interface.

Nested Inheritance:

An interface can extend another interface, forming an inheritance chain.

Example:

```

interface A { void msgA(); }
interface B extends A { void msgB(); }
class C implements B {
    public void msgA() { System.out.println("From A"); }
    public void msgB() { System.out.println("From B"); }
}
public class Main {
    public static void main(String[] args) {
        C obj = new C();
        obj.msgA();
        obj.msgB();
    }
}

```

Explanation:

Interface B extends A, inheriting its methods.

C implements B, thereby also implementing A.

3. Creation of Interfaces and Accessing via Interface References

```
interface Vehicle {
    void start();
    void stop();
}

class Car implements Vehicle {
    public void start() { System.out.println("Car started"); }
    public void stop() { System.out.println("Car stopped"); }
}

public class InterfaceDemo {
    public static void main(String[] args) {
        Vehicle v = new Car(); // interface reference
        v.start();
        v.stop();
    }
}
```

Explanation:

An interface reference can point to an object of any class that implements it — enabling polymorphism.

4. Multiple Interfaces in a Single Class

```
interface Printable {
    void print();
}

interface Showable {
    void show();
}

class Demo implements Printable, Showable {
    public void print() { System.out.println("Printing..."); }
    public void show() { System.out.println("Showing..."); }
}
```

```
public class MultiInterface {  
    public static void main(String[] args) {  
        Demo d = new Demo();  
        d.print();  
        d.show();  
    }  
}
```

Explanation:

A single class (Demo) implements both Printable and Showable, demonstrating multiple interface implementation.

5. Extending Interfaces

```
interface Animal {  
    void eat();  
}  
interface Dog extends Animal {  
    void bark();  
}  
  
class Labrador implements Dog {  
    public void eat() { System.out.println("Eating"); }  
    public void bark() { System.out.println("Barking"); }  
}  
  
class TestInterfaceExtend {  
    public static void main(String[] args) {  
        Labrador l = new Labrador();  
        l.eat();  
        l.bark();  
    }  
}
```

Explanation:

Interface Dog extends Animal, inheriting its abstract methods.

The class implementing Dog must define both.

6. Nested Interfaces

Concept:

A nested interface is an interface declared inside another interface or class.

It helps in logically grouping related functionalities.

Example:

```
class Outer {
    interface Inner {
        void msg();
    }
}

class Implementor implements Outer.Inner {
    public void msg() {
        System.out.println("Nested interface implemented");
    }
}

public class NestedInterfaceDemo {
    public static void main(String[] args) {
        Outer.Inner obj = new Implementor();
        obj.msg();
    }
}
```

Explanation:

The nested interface Inner is implemented using its full name Outer.Inner.

7. Concept & Structure of Packages in Java

Concept:

A package is a namespace that organizes related classes and interfaces.

It prevents name conflicts and supports modular programming.

Syntax:

```
package mypackage;
public class A {
    public void display() { System.out.println("Inside package class"); }
```

```
}
```

Program:

```
// File: mypackage/Message.java
package mypackage;
public class Message {
    public void show() { System.out.println("Hello from package!"); }
}
```

```
// File: Test.java
import mypackage.Message;
class Test {
    public static void main(String[] args) {
        Message m = new Message();
        m.show();
    }
}
```

Advantages:

- Code reusability
- Encapsulation
- Avoids naming conflicts
- Easier maintenance

8. Member Access Control in Packages

Modifier	Within Class	Same Package	Subclass (diff package)	Outside Package
public	✓	✓	✓	✓
protected	✓	✓	✓	✗

default (no modifier)	✓	✓	✗	✗
private	✓	✗	✗	✗

Example:

```
package demo;
public class AccessTest {
    public int pub = 1;
    protected int prot = 2;
    int def = 3;
    private int pri = 4;

    public void display() {
        System.out.println(pub + " " + prot + " " + def + " " + pri);
    }
}
```

When accessed from another package, only public is visible.

9. Importing User-Defined Packages & Static Import

```
// File: mathops/Operations.java
package mathops;
public class Operations {
    public static int add(int a, int b) { return a + b; }
    public static int multiply(int a, int b) { return a * b; }
}
```

```
// File: TestStaticImport.java
import static mathops.Operations.*; // static import
```

```
class TestStaticImport {
    public static void main(String[] args) {
        System.out.println(add(5, 3));    // no need to prefix class name
        System.out.println(multiply(4, 2));
    }
}
```

Explanation:

- import → brings classes into scope.
 - static import → brings static members directly into scope.
-

10. Java File I/O — Reading and Writing (Character Streams)

Concept:

Character streams use classes like `FileReader` and `FileWriter` to handle text data.

Program:

```
import java.io.*;

class FileReadWrite {
    public static void main(String[] args) {
        try {
            // Writing to file
            FileWriter fw = new FileWriter("output.txt");
            fw.write("Hello, this is a file example.");
            fw.close();

            // Reading from file
            FileReader fr = new FileReader("output.txt");
            int i;
            System.out.println("File contents:");
            while ((i = fr.read()) != -1)
                System.out.print((char) i);
            fr.close();
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Explanation:

- `FileWriter` → writes text to a file.

- FileReader → reads text character by character.
 - try-catch ensures proper exception handling and resource safety.
-

MODULE 5

Excellent — these are 8-mark advanced Java programming questions combining Exception Handling, Multithreading, and Lambda expressions.

Below are detailed exam-ready answers with explanations, code, and flow analysis.

1. Nested try-catch for Division & Array Operations

Concept:

Nested try-catch allows handling different exceptions at different levels.

Program:

```
class NestedTryDemo {
    public static void main(String[] args) {
        int[] arr = {10, 20, 30};
        try {
            System.out.println("Outer try block starts");
            int a = 10, b = 0;

            try {
                System.out.println("Inner try block starts");
                System.out.println(a / b); // Division by zero
                System.out.println(arr[5]); // Array out of bounds
            } catch (ArithmeticException e) {
                System.out.println("Inner catch: Division by zero handled");
            }

            System.out.println("Accessing array element: " + arr[5]); // triggers outer catch
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Outer catch: Array index out of range");
        }
        System.out.println("Program continues...");
    }
}
```

```
}  
}
```

Flow of Execution:

1. Inner try throws `ArithmeticException` → caught by inner catch.
2. Control returns to outer try.
3. Outer try throws `ArrayIndexOutOfBoundsException` → caught by outer catch.
4. Program ends normally.

Output:

Outer try block starts

Inner try block starts

Inner catch: Division by zero handled

Outer catch: Array index out of range

Program continues...

2. Consequences of an Uncaught Exception

Explanation:

If an exception is not caught, the JVM terminates the program abnormally after printing the stack trace.

Case Study Example:

```
class UncaughtExample {  
    public static void main(String[] args) {  
        int[] data = {1, 2, 3};  
        System.out.println(data[5]); // Uncaught exception  
        System.out.println("Program continues...");  
    }  
}
```

Output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5  
    at UncaughtExample.main(UncaughtExample.java:4)
```

Consequence:

- Program stops immediately after the exception.
 - Remaining statements are never executed.
 - To avoid this, exceptions must be handled with try-catch.
-

3. Custom Exception — InvalidInputException

Concept:

User-defined exceptions extend the Exception class.

Program:

```
import java.util.*;

class InvalidInputException extends Exception {
    InvalidInputException(String msg) {
        super(msg);
    }
}

class StudentGrading {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        try {
            System.out.print("Enter marks: ");
            int marks = sc.nextInt();
            if (marks < 0)
                throw new InvalidInputException("Marks cannot be negative!");
            System.out.println("Marks accepted: " + marks);
        } catch (InvalidInputException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Output:

Enter marks: -20

Error: Marks cannot be negative!

Explanation:

The user-defined exception gives a custom error message for invalid inputs.

4. Rethrowing Exceptions in Nested try-catch

Concept:

A caught exception can be re-thrown to be handled by another catch block.

Program:

```
class RethrowDemo {
    static void compute() throws ArithmeticException {
        try {
            int x = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Exception handled in compute(), rethrowing...");
            throw e; // rethrow
        }
    }

    public static void main(String[] args) {
        try {
            compute();
        } catch (ArithmeticException e) {
            System.out.println("Exception caught again in main()");
        }
    }
}
```

Output:

Exception handled in compute(), rethrowing...
Exception caught again in main()

Explanation:

- Exception first handled inside compute() and then re-thrown.

- The rethrown exception is handled again in the caller (main).

5. Multithreaded Program — Numbers and Characters

Program:

```
class PrintNumbers extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++)
            System.out.println("Number: " + i);
    }
}

class PrintChars extends Thread {
    public void run() {
        for (char c = 'A'; c <= 'E'; c++)
            System.out.println("Char: " + c);
    }
}

class MultiThreadDemo {
    public static void main(String[] args) {
        PrintNumbers t1 = new PrintNumbers();
        PrintChars t2 = new PrintChars();

        t1.start();
        t2.start();
    }
}
```

Possible Output:

```
Number: 1
Char: A
Number: 2
Char: B
...
```

Explanation:

Both threads run simultaneously, and their outputs interleave depending on CPU scheduling.

6. finally Block in File Operations

Program:

```
import java.io.*;
```

```
class FinallyFileDemo {
    public static void main(String[] args) {
        FileReader fr = null;
        try {
            fr = new FileReader("data.txt");
            int i;
            while ((i = fr.read()) != -1)
                System.out.print((char) i);
        } catch (IOException e) {
            System.out.println("File error: " + e.getMessage());
        } finally {
            try {
                if (fr != null) {
                    fr.close();
                    System.out.println("\nFile closed successfully.");
                }
            } catch (IOException e) {
                System.out.println("Error closing file.");
            }
        }
    }
}
```

Significance:

The finally block executes even if an exception occurs, ensuring file closure and preventing resource leaks.

7. Lambda Expression Implementing Runnable

Program:

```
class LambdaRunnableDemo {
    public static void main(String[] args) {
```

```

Runnable task = () -> {
    for (int i = 1; i <= 5; i++)
        System.out.println("Running thread using Lambda: " + i);
};
Thread t = new Thread(task);
t.start();
}
}

```

Explanation:

- Runnable is a functional interface (single abstract method).
- Lambda replaces anonymous classes, making code concise.

Output:

```

Running thread using Lambda: 1
Running thread using Lambda: 2
...

```

8. Validating User Credentials & Data Upload with Exceptions + Threads

Program:

```

class InvalidUserException extends Exception {
    InvalidUserException(String msg) { super(msg); }
}

class CredentialValidator {
    void validate(String user, String pass) throws InvalidUserException {
        if (!user.equals("admin") || !pass.equals("1234"))
            throw new InvalidUserException("Invalid Credentials!");
    }
}

class DataUploader extends Thread {
    public void run() {
        System.out.println("Uploading data...");
        for (int i = 1; i <= 3; i++)
            System.out.println("Uploading file " + i);
    }
}

```

```

        System.out.println("Upload Complete!");
    }
}

public class SecureUploadSystem {
    public static void main(String[] args) {
        CredentialValidator validator = new CredentialValidator();
        try {
            validator.validate("admin", "1234"); // try invalid to test
            DataUploader uploader = new DataUploader();
            uploader.start(); // run thread concurrently
        } catch (InvalidUserException e) {
            System.out.println("Access Denied: " + e.getMessage());
        }
    }
}

```

Explanation:

- Exception handling ensures only valid users can start upload.
- Multithreading enables simultaneous data upload after validation.

Output:

```

Uploading data...
Uploading file 1
Uploading file 2
Uploading file 3
Upload Complete!

```
