MODULE 3

---

1. Explain the concept of irregular arrays in Java with an example.

An irregular array (also called a jagged array) in Java is a multidimensional array with unequal row lengths.

Unlike regular 2D arrays (which are rectangular), irregular arrays allow each row to have different column sizes.

Example:

```
class IrregularArray {
    public static void main(String[] args) {
        int arr[][] = new int[3][];
        arr[0] = new int[2]; // first row has 2 columns
        arr[1] = new int[3]; // second row has 3 columns
        arr[2] = new int[1]; // third row has 1 column

        int count = 1;
        for (int i = 0; i < arr.length; i++) {
            for (int j = 0; j < arr[i].length; j++) {
                arr[i][j] = count++;
                System.out.print(arr[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Output:

```
1 2
3 4 5
6
```

✅ Advantage: Saves memory since each row can have different sizes.

---

2. Describe the features of the Vector class in Java and demonstrate its methods.

Features of Vector:

- Implements List interface.
- Can grow dynamically.
- Is synchronized (thread-safe).
- Allows random access of elements.
- Can store heterogeneous objects.

Program:

```
import java.util.*;

class VectorExample {
    public static void main(String[] args) {
        Vector<String> v = new Vector<>();

        v.add("Apple");            // Add element
        v.add("Banana");
        v.insertElementAt("Mango", 1);  // Insert element at index
        v.remove("Banana");        // Remove element

        System.out.println("Vector elements: " + v);
    }
}
```

Output:

Vector elements: [Apple, Mango]

---

3. What is an ArrayList in Java? Explain how it differs from an array. Write a program.

ArrayList:

- A resizable array in Java (from java.util package).
- Grows automatically when elements are added.

Differences:

Feature	Array	ArrayList
Size	Fixed	Dynamic
Type	Can hold primitives	Only objects
Memory	Less flexible	Auto-managed
Performance	Slightly faster	Easier to use

Program:

```
import java.util.*;

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

        list.add(10);
        list.add(20);
        list.add(30);
        list.add(40);
        list.add(50);

        list.remove(2);  // Removes element at index 2 (30)

        System.out.println("ArrayList elements: " + list);
    }
}
```

Output:

ArrayList elements: [10, 20, 40, 50]

---

4. (a) Describe constructors of String class (b) Describe methods of StringBuffer class

(a) Constructors of String class:

1. String() – Creates empty string.
2. String(String s) – Copy constructor.
3. String(char[] ch) – From character array.
4. String(byte[] b) – From byte array.

Example:

```
String s1 = new String();
String s2 = new String("Java");
```

(b) StringBuffer methods:

1. append(): Adds data at the end.

   → sb.append("World");

2. ensureCapacity(): Ensures minimum buffer capacity.

   → sb.ensureCapacity(20);

3. reverse(): Reverses characters.

   → sb.reverse();

4. insert(): Inserts data at specified index.

   → sb.insert(2, "Hi");

Example:

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" Java");
sb.insert(5, " World");
sb.reverse();
System.out.println(sb);
```

---

5. Explain inheritance in Java with advantages and single inheritance example.

Concept:

Inheritance allows one class to acquire properties and methods of another class using the keyword extends.

Advantages:

- Code reusability
- Easier maintenance
- Promotes method overriding (polymorphism)

Example – Single Inheritance:

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

Output:

```
Eating...
Barking...
```

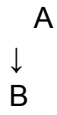
---

6. Differentiate between single, multilevel, and hierarchical inheritance.

Type	Description	Example

Single	One subclass inherits one superclass	A → B
Multilevel	Chain of inheritance	A → B → C
Hierarchical	Multiple subclasses inherit one superclass	A → B, A → C

Diagrams:

Single:


Multilevel:


Hierarchical:


---

7. How constructors are executed in multilevel inheritance hierarchy

In multilevel inheritance, constructors execute from base class to derived class automatically.

Example:

```
class A {
   A() { System.out.println("A constructor"); }
}
class B extends A {
   B() { System.out.println("B constructor"); }
}
class C extends B {
   C() { System.out.println("C constructor"); }
}
```

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

Output:

```
A constructor
B constructor
C constructor
```

✅ Order: Parent → Child (top to bottom).

---

8. Program showing method overriding and runtime polymorphism

Concept:

When a subclass defines a method with the same name and parameters as the parent class, it overrides it.

At runtime, the object type decides which method runs.

Example:

```
class Animal {
    void sound() { System.out.println("Animal makes sound"); }
}
class Dog extends Animal {
    void sound() { System.out.println("Dog barks"); }
}
class Test {
    public static void main(String[] args) {
        Animal a = new Dog(); // Runtime polymorphism
        a.sound();
    }
}
```

Output:

```
Dog barks
```

---

9. Program demonstrating abstract class with abstract and non-abstract methods

Concept:

An abstract class can contain both abstract (unimplemented) and non-abstract (implemented) methods.

Example:

```
abstract class Shape {
    abstract void draw();  // abstract method
    void display() {       // non-abstract method
        System.out.println("This is a shape");
    }
}
class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a Circle");
    }
}
class Test {
    public static void main(String[] args) {
        Circle c = new Circle();
        c.display();
        c.draw();
    }
}
```

Output:

```
This is a shape
Drawing a Circle
```

---

10. Program to demonstrate behavior of final keyword

Concept:

The final keyword is used to:

1. Make variables constant.

2. Prevent method overriding.

3. Prevent inheritance of a class.

Program:

```
final class Vehicle {
    final int speedLimit = 80;   // final variable

    final void run() {           // final method
        System.out.println("Running safely at " + speedLimit + " km/h");
    }
}

class Car extends Vehicle { // ❌ Error: Cannot inherit final class
    // void run() {}        // ❌ Error: Cannot override final method
}

class Test {
    public static void main(String[] args) {
        Vehicle v = new Vehicle();
        v.run();
    }
}
```

Output:

Running safely at 80 km/h

MODULE 4

Here are clear, well-structured 5-mark answers for all 10 questions — perfect for exam preparation:

---

1. Steps to Create and Implement an Interface in Java (with example)

Steps:

1. Define an interface using the keyword interface.

2. Declare abstract methods inside it (no body).

3. Implement the interface in a class using implements keyword.
4. Provide definitions for all abstract methods in the implementing class.
5. Create an object of the implementing class to call the methods.

Example:

```
interface Drawable {
    void draw();
}

class Circle implements Drawable {
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}

class TestInterface {
    public static void main(String[] args) {
        Drawable d = new Circle();
        d.draw();
    }
}
```

Output:

```
Drawing a Circle
```

---

2. Program that implements multiple interfaces in a single class

In Java, a class can implement multiple interfaces — allowing multiple inheritance of type.

Example:

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

Output:

```
Printing...
Showing...
```

---

3. Difference between Class Inheritance and Interface Inheritance

Feature	Class Inheritance	Interface Inheritance
Keyword	extends	implements / extends
Type of Members	Can have implemented methods and fields	Only constants and abstract methods (until Java 8)
Multiple Inheritance	Not supported (only one superclass)	Supported (multiple interfaces)
Purpose	Code reusability	Define a contract
Example	class B extends A {}	class C implements I1, I2 {}

---

4. Nested Interfaces (with example)

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

It is used to group related interfaces together logically.

Example:

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

class Test implements Outer.Inner {
    public void display() {
        System.out.println("Nested Interface Implemented");
    }

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

Output:

Nested Interface Implemented

---

5. Access Control Levels in Packages (Member Access)

Access Modifier	Same Class	Same Package	Subclass (Different Package)	Other Packages
public	✔	✔	✔	✔
protected	✔	✔	✔	✖ (except subclasses)

default (no modifier)	✔	✔	✖	✖
private	✔	✖	✖	✖

Example:

```
// package p1
public class A {
  protected void show() { System.out.println("Hello"); }
}
// package p2
import p1.A;
class B extends A {
  void display() { show(); } // Allowed due to protected
}
```

---

6. Use of the import Statement (with examples)

The import statement is used to access classes from other packages.

Types:

1. Single class import:

```
import java.util.ArrayList;
```

1.

2. Wildcard import:

```
import java.util.*;
```

2.

3. Static import:

```
import static java.lang.Math.*;
System.out.println(sqrt(16));
```

Example:

```
import java.util.ArrayList;

class TestImport {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<>();
        list.add(10);
        System.out.println(list);
    }
}
```

---

7. Role of Static Import (with example)

Purpose:

static import allows access to static members (fields or methods) without class qualification.

Example:

```
import static java.lang.Math.*;

class StaticImportDemo {
    public static void main(String[] args) {
        System.out.println(sqrt(25));   // no need for Math.sqrt()
        System.out.println(pow(2, 3));  // no need for Math.pow()
    }
}
```

Output:

```
5.0
8.0
```

✅ Simplifies code by removing class references.

---

8. Advantages of Using Packages in Java

1. Code Organization: Groups related classes logically.

2. Name Conflict Avoidance: Prevents naming clashes between classes.
3. Access Control: Allows control of class visibility (using access modifiers).
4. Reusability: Classes from a package can be reused easily.
5. Maintenance: Simplifies project management and large codebases.

Example:

package banking; — all classes related to banking can be grouped here.

---

9. Fundamentals of File Handling in Java

Concept:

File handling allows reading, writing, and managing files using the java.io package.

Common Classes:

- File – represents a file/directory path.
- FileReader, FileWriter – for character files.
- FileInputStream, FileOutputStream – for binary files.
- BufferedReader, BufferedWriter – for efficient reading/writing.

Example:

```
import java.io.*;

class FileExample {
   public static void main(String[] args) throws IOException {
      FileWriter fw = new FileWriter("demo.txt");
      fw.write("Hello Java File Handling");
      fw.close();
      System.out.println("File written successfully!");
   }
}
```

---

10. Difference between Byte Stream and Character Stream Classes

Feature	Byte Stream	Character Stream
Package	java.io	java.io
Base Classes	InputStream, OutputStream	Reader, Writer
Data Type	Handles binary data (images, audio, etc.)	Handles text data
Bytes/Chars	Works with 8-bit bytes	Works with 16-bit Unicode characters
Examples	FileInputStream, FileOutputStream	FileReader, FileWriter

Example:

```
// Byte Stream
FileInputStream fin = new FileInputStream("a.jpg");

// Character Stream
FileReader fr = new FileReader("notes.txt");
```

MODULE 5

---

1. Explain the hierarchy of exceptions in Java with a neat diagram.

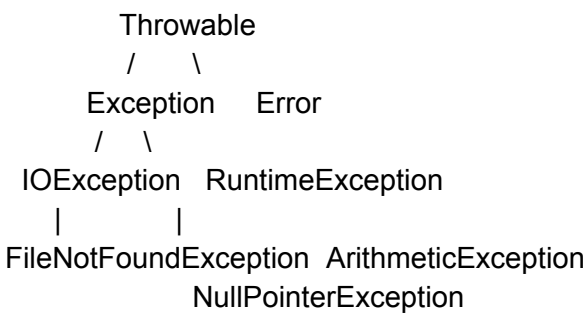
Explanation:

In Java, all exceptions are part of the java.lang.Throwable class hierarchy.

The root class is Throwable, which has two main subclasses:

1. Exception → Represents recoverable conditions (user or logic errors).
2. Error → Represents serious system errors that should not be handled by programs.

Diagram:



Key Points:

- Checked Exceptions (e.g., IOException) → Checked at compile-time.
- Unchecked Exceptions (RuntimeException, etc.) → Checked at runtime.
- Errors (e.g., OutOfMemoryError) → Unrecoverable.

---

2. Handle user input error using try-catch block

Scenario:

User enters a character when a number is expected — causes InputMismatchException.

Code:

```
import java.util.*;

class InputError {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      try {
         System.out.print("Enter a number: ");
         int num = sc.nextInt();   // may throw InputMismatchException
         System.out.println("You entered: " + num);
```

```
        } catch (InputMismatchException e) {
            System.out.println("Error: Please enter a valid number!");
        }
    }
}
```

Output:

```
Enter a number: a
Error: Please enter a valid number!
```

---

3. How does the finally block ensure resource management in file handling?

Explanation:

The finally block executes whether or not an exception occurs.

It is commonly used to close files, network connections, or database links to prevent resource leaks.

Scenario Example:

```
import java.io.*;

class FileDemo {
    public static void main(String[] args) {
        FileReader fr = null;
        try {
            fr = new FileReader("data.txt");
            int ch;
            while ((ch = fr.read()) != -1)
                System.out.print((char) ch);
        } catch (IOException e) {
            System.out.println("File not found!");
        } finally {
            try {
                if (fr != null)
                    fr.close();  // ensures file is closed
                System.out.println("\nFile closed successfully.");
            } catch (IOException e) {
                System.out.println("Error closing file.");
            }
        }
```

```
    }
}
```

✅ Ensures that the file is always closed, even if an exception occurs.

---

4. Program using multiple catch blocks for calculator errors

Program:

```
class Calculator {
    public static void main(String[] args) {
        try {
            int a = 10, b = 0;
            int[] arr = {1, 2, 3};

            int result = a / b;              // may throw ArithmeticException
            System.out.println(arr[5]);      // may throw ArrayIndexOutOfBoundsException

        } catch (ArithmeticException e) {
            System.out.println("Error: Division by zero!");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: Invalid array index!");
        } catch (Exception e) {
            System.out.println("General error occurred.");
        }
    }
}
```

Output:

Error: Division by zero!

---

5. Custom Exception – NotEligibleException for age < 18

Program:

```
class NotEligibleException extends Exception {
    NotEligibleException(String msg) {
        super(msg);
    }
}
```

```
class VotingSystem {
    public static void main(String[] args) {
        int age = 16;
        try {
            if (age < 18)
                throw new NotEligibleException("Not eligible to vote!");
            else
                System.out.println("You can vote.");
        } catch (NotEligibleException e) {
            System.out.println("Exception: " + e.getMessage());
        }
    }
}
```

Output:

Exception: Not eligible to vote!

✅ Demonstrates user-defined exception handling for custom conditions.

---

6. Compare the two ways of creating threads in Java

(a) Extending the Thread class

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread using Thread class");
    }
}
class Test1 {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
    }
}
```

(b) Implementing the Runnable interface

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread using Runnable interface");
    }
}
```

```
class Test2 {
    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable());
        t.start();
    }
}
```

Comparison:

Feature	Extending Thread	Implementing Runnable
Inheritance	Not possible to extend another class	Can still extend another class
Flexibility	Less flexible	More flexible
Real-world Use	Rarely used	✅ Preferred (used in real apps, executors, etc.)

Preferable:

👉 Implementing Runnable — allows better reusability and scalability.

---

7. Inner class accessing outer class variable

Concept:

An inner class has access to all members (including private ones) of its outer class.

Program:

```
class Outer {
    private int num = 100;

    class Inner {
        void display() {
            System.out.println("Outer class variable: " + num);
```

```
        }
    }

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

Output:

Outer class variable: 100

✅ Demonstrates how inner classes enhance encapsulation and modularity.

---

8. Lambda expressions and code readability (example: greeting message)

Concept:

A lambda expression provides a shorter syntax for implementing functional interfaces (interfaces with one abstract method).

It improves readability by eliminating boilerplate code like anonymous classes.

Example:

```
interface Greeting {
    void sayHello(String name);
}

class LambdaDemo {
    public static void main(String[] args) {
        Greeting g = (name) -> System.out.println("Hello, " + name + "!");
        g.sayHello("Ananya");
    }
}
```

Output:

Hello, Ananya!

✅ Benefit:

- Compact and readable.
- Commonly used in event handling, streams, and collections.
