JAVA

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

Core Concepts of Object-Oriented Programming (OOP) in Java

1. **Class and Object**

    - **Class**: A blueprint or template that defines the structure and behavior of objects. It includes fields (attributes) and methods (functions).
    - **Object**: An instance of a class that represents a real-world entity with state (attributes) and behavior (methods).

    ```
    class Car {
        String color;
        void drive() {
            System.out.println("Car is driving");
        }
    }
    Car myCar = new Car(); // Object
    ```

2.Encapsulation

- Encapsulation involves wrapping data (fields) and methods that operate on the data into a single unit (class). It restricts direct access to data using access modifiers (`private`, `protected`, `public`) and provides controlled access through getter and setter methods.

```
class Employee {
    private String name;
    public String getName() {
        return name;
```

```
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

3.Inheritance

- Inheritance allows a child class to inherit fields and methods from a parent class, promoting code reusability and establishing a hierarchical relationship. The `extends` keyword is used to implement inheritance in Java.

```
// Parent class
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

// Child class
class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Create an object of the Dog class
        Dog dog = new Dog();

        // Call methods from both the parent (Animal) and child (Dog) classes
        dog.eat();  // Method inherited from the Animal class
        dog.bark(); // Method defined in the Dog class
    }
}
```

1. Single Inheritance

- A child class inherits from a single parent class.

```
class Animal {
    void eat() {}
}
class Dog extends Animal {
    void bark() {}
}
```

2. Multilevel Inheritance

- A child class inherits from a parent class, and that parent class is itself a child of another class.

```
class Animal {
    void eat() {}
}
class Mammal extends Animal {
    void walk() {}
}
class Dog extends Mammal {
    void bark() {}
}
```

3. Hierarchical Inheritance

- Multiple child classes inherit from a single parent class.

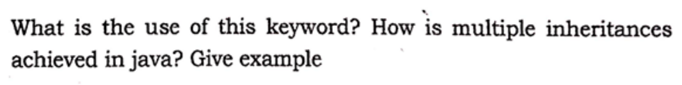
```
class Animal {
    void eat() {}
}
```

```
class Dog extends Animal {
    void bark() {}
}
class Cat extends Animal {
    void meow() {}
}
```

4. Hybrid Inheritance (Achieved through Interfaces)

- Combines multiple types of inheritance, such as single and hierarchical, but is only possible using **interfaces** in Java.

```
interface Animal {
    void eat();
}
interface Pet {
    void play();
}
class Dog implements Animal, Pet {
    public void eat() {}
    public void play() {}
}
```



What is the use of this keyword? How is multiple inheritances achieved in java? Give example

Multiple inheritance in Java is achieved through interfaces. While Java doesn't support multiple inheritance of classes (to avoid the "diamond problem"), a class can implement multiple interfaces. Here's an example:

```
// Interface 1
interface Animal {
    void eat();
}
```

```
// Interface 2
interface Bird {
    void fly();
}

// Class implementing multiple interfaces
class Parrot implements Animal, Bird {
    @Override
    public void eat() {
        System.out.println("Parrot can eat");
    }

    @Override
    public void fly() {
        System.out.println("Parrot can fly");
    }
}
```

4.Polymorphism

- **Definition**: Polymorphism means "many forms," and it allows the same entity (method or object) to behave differently in different contexts.
- **Method Overloading**: Same method name, different parameter lists (compile-time polymorphism).
- **Method Overriding**: Subclass redefines a method in the parent class (runtime polymorphism).

```
class Shape {
    void draw() {
        System.out.println("Drawing a shape");
    }
}
class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a circle");
    }
}
```

5.Abstraction

- Abstraction focuses on exposing only essential details to the user and hiding implementation details. It is achieved using abstract classes (`abstract` keyword) or interfaces (`interface`).

```
abstract class Vehicle {
    abstract void move();
}
class Bike extends Vehicle {
    void move() {
        System.out.println("Bike moves on two wheels");
    }
}
```

Q. Why java doesn’t support multiple inheritance ?Explain Diamond problem .

Java does not support multiple inheritance with classes to prevent ambiguity caused by the diamond problem. Instead, interfaces are used to achieve multiple inheritance-like behavior. In java we cannot extend multiple classes.

Diamond Problem

The **diamond problem** occurs in object-oriented programming when a class inherits from two classes that share a common base class, leading to ambiguity about which version of the inherited methods or properties should be used. For example, if Class A is the parent of both Class B and Class C, and Class D inherits from both B and C, Class D has two paths to inherit features from Class A. This creates confusion about whether to use the method from B, C, or A. Java resolves this issue by not allowing multiple inheritance through classes, but it allows multiple inheritance through **interfaces**, where the ambiguity is resolved by explicitly overriding conflicting methods in the subclass.

Interface

An
interface in Java is a blueprint of a class that contains a collection of abstract methods (methods without a body) and static constants. It is used to achieve **100% abstraction** and multiple inheritance in Java.

- **Multiple Inheritance**: A class can implement multiple interfaces, enabling Java to overcome the diamond problem.

Here's a tabular representation of the differences between an **abstract class** and an **interface** in Java:

Feature	Abstract Class	Interface
Purpose	Provides a common base class for related classes with shared code and abstract methods.	Specifies a contract that classes must implement; no implementation provided (before Java 8).
Method Implementation	Can have both abstract (no body) and concrete (with body) methods.	Before Java 8, only abstract methods. From Java 8 onwards, can include default and static methods.
Inheritance	A class can extend only one abstract class (single inheritance).	A class can implement multiple interfaces (multiple inheritance for behavior).
Variables	Can have instance variables (with any access modifier).	Can only have `public static final` (constant) variables.
Constructors	Can have constructors to initialize fields.	Cannot have constructors.
Access Modifiers for Methods	Methods can have any access modifier (`public` , `protected` , `private`).	All methods are `public` by default (except private methods introduced in Java 9).
Suitability	Used when classes share a common structure or base implementation.	Used to define a standard behavior or contract for unrelated classes.

Keyword Used	Declared with the `abstract` keyword.	Declared with the `interface` keyword.
Performance	Slightly faster than interfaces as they use normal inheritance.	Slightly slower as the methods are resolved dynamically (depending on implementation).

Let me know if you need further details or examples!

```
import java.util.Scanner;

// Interface declaration
interface Calculation {
    // Method for sum
    int sum(int[] numbers);

    // Method for average
    double average(int[] numbers);
}

// Implementing the interface
class CalculateNumbers implements Calculation {

    // Implementing the sum method
    public int sum(int[] numbers) {
        int sum = 0;
        for (int num : numbers) {
            sum += num;
        }
        return sum;
    }

    // Implementing the average method
    public double average(int[] numbers) {
        int sum = sum(numbers); // Calling the sum method
        return (double) sum / numbers.length;
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Taking input for number of elements
        System.out.print("Enter the number of elements: ");
        int n = sc.nextInt();

        int[] numbers = new int[n];

        // Taking input for array elements
        System.out.println("Enter " + n + " numbers:");
        for (int i = 0; i < n; i++) {
            numbers[i] = sc.nextInt();
        }

        // Creating object of CalculateNumbers class
        CalculateNumbers calc = new CalculateNumbers();

        // Calculating sum and average
        int totalSum = calc.sum(numbers);
        double avg = calc.average(numbers);

        // Displaying the results
        System.out.println("Sum: " + totalSum);
        System.out.println("Average: " + avg);
    }
}
```

Features of Java

1. **Platform Independence**

Java is a write-once, run-anywhere (WORA) language. Its programs are compiled into bytecode, which can run on any device equipped with a Java Virtual Machine (JVM), regardless of the underlying operating system.

2. **Object-Oriented**

   Java follows the principles of object-oriented programming, such as encapsulation, inheritance, and polymorphism. This makes the code modular, reusable, and easier to maintain.

3. **Simple**

   Java is designed to be easy to learn and use. It removes complex features like explicit pointers and operator overloading, making it beginner-friendly.

4. **Secure**

   Java provides a secure runtime environment by eliminating the use of explicit pointers and running code in the JVM sandbox. It also includes built-in security features like cryptography, authentication, and access control.

5. **Robust**

   Java emphasizes reliability with features like strong memory management, automatic garbage collection, and exception handling. These features help prevent crashes and runtime errors.

6. **Multithreaded**

   Java supports multithreading, allowing multiple threads of a program to run concurrently. This feature is ideal for applications that perform multiple tasks simultaneously, such as video games and web servers.

7. **High Performance**

   While Java is not as fast as compiled languages like C++, its Just-In-Time (JIT) compiler improves performance by converting bytecode to native machine code at runtime.

8. **Distributed Computing**

   Java supports distributed computing with built-in APIs like Remote Method Invocation (RMI) and support for networked applications. This makes it ideal for creating enterprise-level applications.

9. **Dynamic and Extensible**

   Java programs can dynamically load classes and libraries at runtime. This allows developers to extend application functionality without modifying the original code.

10. **Portability**

    Java code is highly portable due to its architecture-neutral bytecode. This enables Java programs to run seamlessly across diverse hardware and software environments.

11. **Automatic Memory Management**

    Java manages memory automatically through garbage collection. This reduces the chances of memory leaks and simplifies the development process.

12. **Rich Standard Library**

    Java comes with an extensive library of pre-built classes and methods, covering areas like data structures, networking, GUI development, and database connectivity. This accelerates development and reduces coding effort.

Java Architecture

Java Architecture

Click t

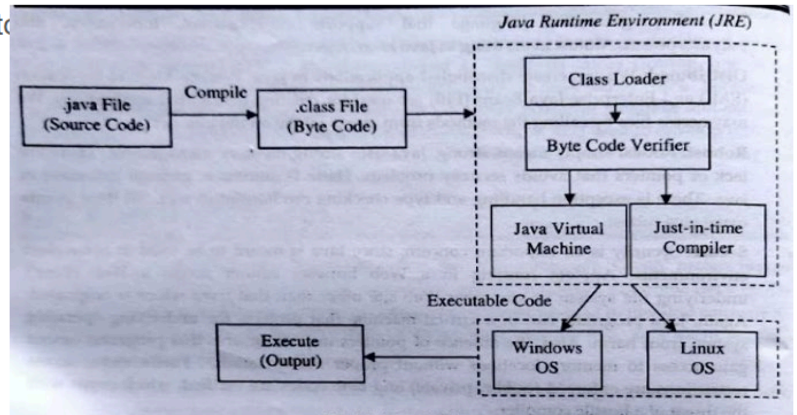


Fig 1.1 Java Architecture

Java Architecture Components

Java architecture consists of several key components that work together to execute Java programs:

1. **Java Development Kit (JDK)**

- It is a software development environment used in the development of Java applications
- Java Development Kit holds JRE, a compiler, class loader.

1. **Java Runtime Environment (JRE)**

It provides an environment in which Java programs are executed. JRE takes our Java code, integrates it with the required libraries, and then starts the JVM to execute it.

1. **Java Virtual Machine (JVM)**
   • JVM is an abstract machine that provides the environment in which Java bytecode is executed.

Here’s a more detailed yet concise explanation of the JVM's responsibilities:

1. **Loading and Verifying Bytecode**: The JVM loads compiled bytecode and verifies its correctness and security, ensuring that no harmful code is executed by checking for potential security breaches or violations.
2. **Interpreting Bytecode**: The JVM converts bytecode into machine code either by interpreting it one instruction at a time or using Just-In-Time (JIT) compilation to optimize performance by translating bytecode into native code at runtime.
3. **Memory Management**: JVM manages memory automatically through garbage collection, which identifies and frees memory used by objects no longer needed, preventing memory leaks and improving performance.
4. **Handling Security**: The JVM enforces security policies through the Java Security Manager and bytecode verification, restricting unauthorized actions like accessing the file system or network resources, making Java safe for distributed environments.

The process flow works as follows:

1. Source code (.java) is written by developers
2. Java compiler (javac) compiles the source code into bytecode (.class)
3. JVM loads and verifies the bytecode
4. JVM interprets or compiles the bytecode into native machine code
5. Program executes on the host machine

This architecture enables Java's "Write Once, Run Anywhere" capability, as the bytecode can run on any platform with a JVM installed.

Constructor

- A constructor is a block of code which used to initialize the object.
- Every time an object is created using the new keyword, at least one constructor is called.
- Types:
- Default constructor
- Parameterised constructor

1.Default constructor

- A constructor is called "Default Constructor" when it doesn't have any parameter.
- The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.
- Syntax:

```
<class_name>(){
}
```

2.Parameterised Constructor

- A constructor which has a specific number of parameters.
- Used to provide different values to distinct objects.

- Syntax:

<class_name> (variable1, variable 2, ...) {

}

Constructor overloading

Constructor overloading is a concept in object-oriented programming where a class can have multiple constructors with different parameter lists. These constructors allow creating objects in various ways, depending on the arguments passed during object creation.

Key Points:

- **Same Name**: All constructors in a class must have the same name as the class itself.
- **Different Parameters**: The parameter lists (type, number, or order) must differ for each constructor to distinguish them.
- **Purpose**: Constructor overloading provides flexibility in initializing objects with different types or amounts of data.

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

    // Constructor 1: Default constructor
    public Student() {
        name = "Unknown";
        age = 0;
        course = "Not enrolled";
    }

    // Constructor 2: Constructor with two parameters
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
        this.course = "Not enrolled";
    }

    // Constructor 3: Constructor with three parameters
    public Student(String name, int age, String course) {
        this.name = name;
        this.age = age;
        this.course = course;
    }

    // Method to display student details
    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Course: " + course);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(); // Calls default constructor
        Student s2 = new Student("Alice", 20); // Calls constructor with 2 param
        Student s3 = new Student("Bob", 22, "Computer Science"); // Calls cons

        s1.displayInfo();
        s2.displayInfo();
        s3.displayInfo();
    }
}
```

Super keyword-

The `super` keyword in Java is used in object-oriented programming to refer to the parent class (superclass) of the current object. It provides access to the methods, constructors, or properties of the parent class, allowing the child class to interact with its superclass directly.

Key Uses of `super` Keyword:

1. **Access Parent Class Methods**:
    - Used to call a method from the parent class that has been overridden in the child class.

```
class Parent {
    public void display() {
        System.out.println("Method in Parent class.");
    }
}

class Child extends Parent {
    public void display() {
        System.out.println("Method in Child class.");
    }

    public void callParentMethod() {
        super.display(); // Calls the parent class method
    }
}

public class Main {
    public static void main(String[] args) {
        Child child = new Child();
        child.display();       // Calls overridden method in Child class
        child.callParentMethod(); // Calls method in Parent class
    }
}
output:
```

```
Method in Child class.
Method in Parent class.
```

2. **Call Parent Class Constructor**:
    - Used to invoke the constructor of the parent class explicitly. It must be the first statement in the child class constructor.

```
class Parent {
    // Default constructor of Parent class
    public Parent() {
        System.out.println("Parent class default constructor called.");
    }
}

class Child extends Parent {
    // Default constructor of Child class
    public Child() {
        // Call the default constructor of the Parent class explicitly
        super(); // Optional, as Java implicitly calls the default constructor of the
        System.out.println("Child class default constructor called.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Create an object of Child class
        Child child = new Child(); // This will trigger the parent constructor first
    }
}
output:
Parent class default constructor called.
Child class default constructor called.
```

3. **Access Parent Class Variables**:

- Used to access a variable from the parent class if the child class has a variable with the same name.

```
class Parent {
    String name = "Parent";
}

class Child extends Parent {
    String name = "Child";

    public void displayNames() {
        System.out.println("Child name: " + name);         // Refers to child class vari
        System.out.println("Parent name: " + super.name);  // Refers to parent class
    }
}

public class Main {
    public static void main(String[] args) {
        Child child = new Child();
        child.displayNames();
    }
}
output:
Child name: Child
Parent name: Parent
```

Abstract Methods in Java

An **abstract method** is a method that is declared without an implementation (no method body) and is meant to be overridden in the subclasses. It serves as a blueprint for methods, ensuring that all subclasses provide their own specific implementation of the method.

An **abstract class** in Java is a class that cannot be instantiated and is meant to be subclassed. It can contain both abstract methods (without a body) and concrete methods (with a body), providing a base for other classes to implement or extend.

Characteristics of Abstract Methods:

1. Can only exist in an **abstract class** or an **interface**.
2. Declared using the `abstract` keyword.
3. Must end with a semicolon (`;`) instead of a method body.
4. A subclass inheriting an abstract method must override it, unless the subclass itself is declared as abstract.

```java
// Abstract class
abstract class Shape {
    // Abstract method
    abstract void draw();
}

// Concrete subclass
class Circle extends Shape {
    // Provide implementation for the abstract method
    @Override
    void draw() {
        System.out.println("Drawing a circle");
    }
}

// Concrete subclass
class Rectangle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a rectangle");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape shape1 = new Circle();
        Shape shape2 = new Rectangle();
        shape1.draw(); // Output: Drawing a circle
        shape2.draw(); // Output: Drawing a rectangle
    }
}
```

Dynamic Method Dispatch (also known as Runtime Polymorphism)

Dynamic method dispatch refers to the process of resolving a method call at **runtime** rather than at compile time. It occurs when a method is overridden in a subclass, and the decision about which method to execute is made based on the type of the object, not the type of the reference.

How They Are Related:

- **Abstract Methods**: Provide a way to define a method that must be implemented by subclasses, ensuring polymorphic behavior.
    - **Dynamic Method Dispatch**: Happens when an overridden method in a subclass is called through a parent class reference. This enables runtime binding of the method implementation.

Example: Dynamic Method Dispatch with Abstract Methods

```
abstract class Animal {
    abstract void sound(); // Abstract method
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
```

```
class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal; // Parent class reference

        // Runtime polymorphism
        animal = new Dog();
        animal.sound(); // Output: Dog barks

        animal = new Cat();
        animal.sound(); // Output: Cat meows
    }
}
```

Method Overloading (Compile-Time Polymorphism)

Method overloading occurs when two or more methods in the same class have the same name but different parameter lists (number, type, or order of parameters).

```
class Calculator {
    // Method to add two numbers
    int add(int a, int b) {
        return a + b;
    }

    // Overloaded method to add three numbers
    int add(int a, int b, int c) {
        return a + b + c;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        // Call the add method with two parameters
        System.out.println("Sum of two numbers: " + calc.add(5, 10)); // Output: 15

        // Call the add method with three parameters
        System.out.println("Sum of three numbers: " + calc.add(5, 10, 15)); // Output
    }
}
```

Method Overriding (Runtime Polymorphism)

Method overriding occurs when a subclass provides its specific implementation for a method that is already defined in its parent class. The method in the subclass must have the **same name, return type, and parameters** as the method in the parent class.

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Parent reference, child object
        animal.sound(); // Output: Dog barks
    }
}
```

Here's a detailed table comparing **Method Overloading** and **Method Overriding**:

Feature	**Method Overloading**	**Method Overriding**
Definition	Same method name, but different parameters (type, number, or order).	Same method name and parameters, but with a different implementation in subclass.
Occurs	Within the same class.	Between a parent class and a subclass.
Polymorphism Type	Compile-time polymorphism.	Runtime polymorphism.
Method Signature	Must differ in the number, type, or order of parameters.	Must have the same method signature (name, return type, and parameters).
Return Type	Return type can be different.	Return type must be the same or covariant (in subclass).
Binding	Method binding happens at compile time.	Method binding happens at runtime (dynamic binding).
Purpose	Used to perform similar actions with different types or numbers of inputs.	Used to provide a specific implementation of an inherited method in a subclass.
Inheritance	Not required.	Inheritance is required (must be a subclass method).

These differences highlight how method overloading allows for method variations within the same class, while method overriding is used to modify the behavior of inherited methods from a parent class.

Throw (Keyword)

- Used to explicitly throw an exception within a method or block of code.

- Creates an instance of an exception class and transfers control to the nearest catch block that can handle the thrown exception.
- Can throw both checked and unchecked exceptions.

Example:

```
public void checkAge(int age) {
    if (age < 18) {
        throw new ArithmeticException("Not eligible");
    } else {
        System.out.println("Eligible for voting");
    }
}
```

In this example, throw is used to explicitly throw an ArithmeticException when the input age is less than 18.

Throws (Clause)

- Used in a method signature to declare the exceptions that a method can throw.
- Indicates to the caller that the method may throw exceptions of specific types.
- Only checked exceptions can be declared using throws.

Multithreading:

The process of executing multiple threads is known as multithreading .**Multithreading** is the concurrent execution of more than one part (thread) of a program to maximize CPU utilization and improve performance.

Advantages of Multithreading:

1. **Improved Performance**:
    - By allowing multiple threads to run concurrently, multithreading can significantly improve the performance of CPU-bound tasks. It

makes efficient use of CPU resources, especially in a multi-core processor environment.

2. **Better Resource Utilization**:
   - Multithreading helps in utilizing CPU resources more efficiently by performing multiple operations at the same time, minimizing idle time and improving throughput.
3. **Enhanced Responsiveness**:
   - Multithreading improves the responsiveness of applications. For example, in GUI applications, while one thread is handling user input, another can perform background tasks, ensuring the user interface remains responsive.
4. **Simplified Program Structure**:
   - Complex problems can be broken into smaller threads, each performing a specific task. This can make the program easier to design and maintain.
5. **Scalability**:
   - Multithreading can help scale applications across multiple processors or cores, leading to better overall performance as system resources (like CPUs) increase.
6. **Concurrency**:
   - Multiple tasks can run concurrently, even if they are not dependent on each other, improving the efficiency of operations, especially in real-time or network-based applications.
7. **Better System Efficiency**:
   - In systems where multiple tasks need to be executed simultaneously (e.g., web servers, database servers), multithreading helps maximize the system's efficiency by parallelizing work.
8. **Asynchronous Task Execution**:

With multithreading, one thread can execute background tasks without blocking the main thread, leading to asynchronous operations. This is helpful in tasks like downloading data or waiting for user input without freezing the system.

Application of multithreading

- **Parallel Processing**: Enables the execution of multiple tasks concurrently to utilize multiple processors or cores efficiently, speeding up computations.
- **Web Servers**: Handles multiple client requests simultaneously, improving response time and resource management by creating a new thread for each request.
- **GUI Applications**: Keeps the user interface responsive by running long tasks (like downloading files or computations) in background threads.
- **Real-Time Systems**: Provides timely execution of tasks in systems where specific timing and synchronization are critical (e.g., embedded systems).
- **Gaming**: Manages multiple aspects of a game, such as rendering, player input, and AI, in parallel to ensure smooth gameplay.
- **Database Management**: Processes multiple queries and transactions at the same time, reducing wait times and improving throughput.
- **Network Applications**: Handles multiple data streams or connections concurrently, such as in chat applications or peer-to-peer networks.
- **File Systems**: Allows multiple files to be read or written concurrently, improving performance when dealing with large datasets.
- **Scientific Simulations**: Runs various simulations in parallel, such as climate modeling or physics simulations, reducing the time required

for complex calculations.

- **Video and Audio Processing**: Allows for concurrent encoding, decoding, or filtering of media streams, enhancing performance in real-time processing.

```
// Class to create and run the first thread
class Thread1 extends Thread {
    public void run() {
        System.out.println("Thread 1 is running.");
    }
}

// Class to create and run the second thread
class Thread2 extends Thread {
    public void run() {
        System.out.println("Thread 2 is running.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Create instances of both threads
        Thread1 t1 = new Thread1();
        Thread2 t2 = new Thread2();

        // Start the threads
        t1.start();
        t2.start();

        // Optional: Wait for threads to finish
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Both threads have finished.");
    }
}
```

In Java, a thread can be created in two primary ways: by extending the `Thread` class or by implementing the `Runnable` interface. Here's how each method works:

1. Creating a Thread by Extending the `Thread` Class

You can create a new thread by creating a subclass of the `Thread` class and overriding the `run()` method. The `run()` method defines the code that will be executed by the thread when it starts.

Steps:

1. Create a subclass of the `Thread` class.
2. Override the `run()` method to specify the task the thread will perform.
3. Create an instance of the subclass and call `start()` to initiate the thread.

Example:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("This is a thread created by extending the
Thread class.");
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start(); // Start the thread
    }
}
```

2. Creating a Thread by Implementing the `Runnable` Interface

Instead of extending the `Thread` class, you can implement the `Runnable` interface and pass it to a `Thread` object. This is considered a more flexible approach because Java allows you to implement multiple interfaces but only extends one class.

Steps:

1. Implement the `Runnable` interface by overriding its `run()` method.
2. Create a `Thread` object, passing the `Runnable` object to its constructor.
3. Call `start()` on the `Thread` object to begin executing the `run()` method.

Example:

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("This is a thread created by implementing t
he Runnable interface.");
    }
}

public class Main {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread t = new Thread(myRunnable);
        t.start(); // Start the thread
    }
}
```

Key Differences:

- **Thread class**: When you extend `Thread`, your class directly becomes a thread.
- **Runnable interface**: When you implement `Runnable`, you can separate the task from the thread itself, which allows for more flexibility (e.g., using the same `Runnable` for multiple threads).

In general, using `Runnable` is preferred when you want to separate the task (the `run()` method) from the thread, as it provides more flexibility for future changes.

Make a thread using runnable interface to display number from I to 20; each number should be displayed in the interval of 2 seconds.

```
class NumberPrinter implements Runnable {
    public void run() {
        for (int i = 1; i <= 20; i++) {
            try {
                System.out.println(i);  // Print the number
                Thread.sleep(2000);     // Sleep for 2 seconds (2000 millisec
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted: " + e.getMessage())
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        // Create an instance of the Runnable class
        NumberPrinter printer = new NumberPrinter();
        // Create a new Thread and pass the Runnable object
        Thread t = new Thread(printer);

        // Start the thread
        t.start();
    }
}
```

. WAP which will display your name in one thread and your address in another thread in every 500 milliseconds. There should be 10000 iteration.

```
class NamePrinter implements Runnable {
    public void run() {
        for (int i = 0; i < 10000; i++) {
            try {
                System.out.println("Name: John Doe"); // Replace with your
                Thread.sleep(500); // Sleep for 500 milliseconds
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted: " + e.getMessage())
            }
        }
    }
}

class AddressPrinter implements Runnable {
    public void run() {
        for (int i = 0; i < 10000; i++) {
            try {
                System.out.println("Address: 123 Main Street, City, Country"
                Thread.sleep(500); // Sleep for 500 milliseconds
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted: " + e.getMessage())
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        // Create thread objects for both name and address printers
        NamePrinter namePrinter = new NamePrinter();
        AddressPrinter addressPrinter = new AddressPrinter();

        // Create Thread objects for each runnable
        Thread nameThread = new Thread(namePrinter);
        Thread addressThread = new Thread(addressPrinter);

        // Start both threads
        nameThread.start();
        addressThread.start();
    }
}
```

Thread Lifecycle in Java

In Java, a **thread** is a lightweight sub-process or a small unit of a process that runs concurrently with other threads within the same program. It allows for parallel execution of tasks and is managed by the Java Thread class or implemented via the Runnable interface.

The **thread lifecycle** in Java defines the different states a thread goes through during its execution. A thread can be in one of the following states:

New (Born State):

- A thread lifecycle begins in new state. In this state , a thread is created but not yet started. It is in the **New** state immediately after the `Thread` object is instantiated.

    Thread t = new Thread();

Runnable :

After the thread is started using start() method it enters Runnable state. A thread is considered to be in Runnable when it is ready to be executed but is waiting for CPU time.

Running :

When thread scheduler allocates the CPU time it enters running state and run() method is being executed by the CPU.

Waiting :

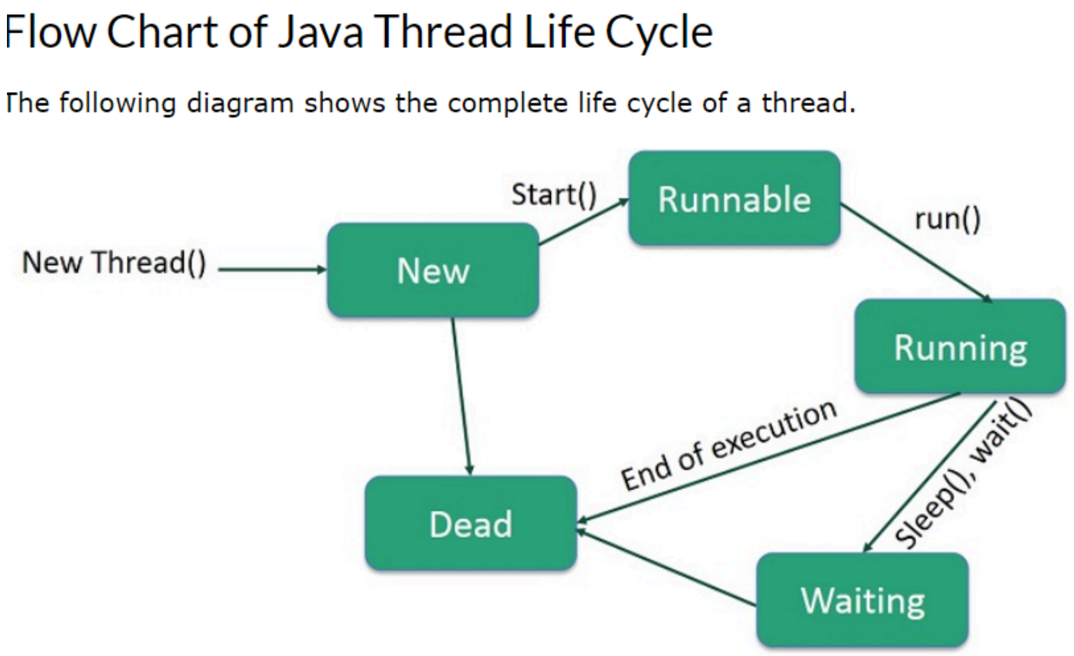
The thread enters the waiting state when it is waiting indefinitely for the another thread to perform action .

- The `wait()` method is typically used to move a thread to this state.

It can also wait for specified period for this sleep() method is used

Terminate:

Thread enters this state when the run() method finishes its tasks or thread is terminated.



Flow Chart of Java Thread Life Cycle

The following diagram shows the complete life cycle of a thread.

Thread synchronization :

Thread Synchronization in Java

Thread synchronization is a mechanism in Java that ensures that only one thread can access a shared resource (like a method or a block of code) at a time. This is crucial in multithreaded applications where multiple threads may try to access and modify shared data concurrently, potentially causing data inconsistency or corruption.

Why Synchronization is Necessary:

1. **Data Consistency**:
    - If multiple threads access shared resources simultaneously and at least one of them modifies the resource, it can lead to inconsistent or incorrect results. Synchronization ensures that only one thread can access the resource at a time, preventing conflicts.
2. **Avoiding Race Conditions**:
    - A **race condition** occurs when the outcome of a process depends on the timing or order of execution of threads.

Synchronization helps in avoiding race conditions by making critical sections of code mutually exclusive.

3. **Thread Safety**:
   - Synchronization ensures that shared data is accessed in a thread-safe manner, meaning that no other thread can interfere with the ongoing operations of another thread.

How to Synchronize Threads in Java:

1. **Synchronized Methods**:
   - You can use the `synchronized` keyword to make an entire method synchronized. When a method is synchronized, only one thread can access it at a time for an object. This ensures that the method's code block is executed by only one thread at a time.

```
public synchronized void methodName() {
    // Code block
}
```

2.Synchronized Blocks:

- We can also synchronize specific parts of code within a method using **synchronized blocks**. This is useful if we only need to synchronize a small portion of the method to improve performance.

```
public void methodName() {
    synchronized (this) {
        // Critical section of code
    }
}
```

3.Synchronized Static Methods:

- If we need to synchronize access to a static method (which belongs to the class, not an instance),we can synchronize the method using the `synchronized` keyword, just like we would for instance methods.

```
public static synchronized void methodName() {
    // Code block
}
```

Thread priority in Java determines the order in which threads are scheduled for execution by the JVM, with higher-priority threads given preference. It helps manage resource allocation, ensuring important tasks run before less critical ones in multi-threaded applications.

```
class Table {
    // Method to print the table, not synchronized
    void printTable(int n) {
        for (int i = 1; i <= 5; i++) {
            // Print the multiplication result
            System.out.println(n * i);
            try {
                // Pause execution for 400 milliseconds
                Thread.sleep(400);
            } catch (Exception e) {
                // Handle any exceptions
                System.out.println(e);
            }
        }
    }
}

class MyRunnable1 implements Runnable {
    Table t;
    // Constructor to initialize Table object
    MyRunnable1(Table t) {
        this.t = t;
    }
    // Run method to execute thread
    public void run() {
        // Call printTable method with argument 5
        t.printTable(5);
    }
}

class MyRunnable2 implements Runnable {
    Table t;
    // Constructor to initialize Table object
    MyRunnable2(Table t) {
        this.t = t;
    }
    // Run method to execute thread
    public void run() {
        // Call printTable method with argument 100
        t.printTable(100);
    }
}

class TestSynchronization1 {
    public static void main(String args[]) {
        // Create a Table object
        Table obj = new Table();
        // Create MyRunnable1 and MyRunnable2 objects with the same Table objec
        MyRunnable1 r1 = new MyRunnable1(obj);
        MyRunnable2 r2 = new MyRunnable2(obj);

        // Create Thread objects passing the Runnable objects
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);

        // Start both threads
        t1.start();
        t2.start();
    }
}
```

JDBC

RMI

Difference Between Runtime Exception and Compile-Time Exception:

Aspect	**Runtime Exception**	**Compile-Time Exception**
Definition	Exceptions that occur during the program's execution (runtime).	Exceptions that are checked by the compiler at compile time.
Checking	Not checked by the compiler; occurs at runtime.	Checked by the compiler; must be handled in the code.
Handling Requirement	Can be optionally handled using `try-catch` blocks.	Must be handled using `try-catch` or declared using `throws`.
Examples	`NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException`.	`IOException`, `SQLException`, `ClassNotFoundException`.
Causes	Typically caused by logic errors or unforeseen circumstances during execution.	Typically caused by external factors or recoverable errors that the compiler can foresee.
Inheritance	Inherits from `RuntimeException`, which is a subclass of `Exception`.	Directly inherits from `Exception` (not a subclass of `RuntimeException`).
Program Compilation	Program compiles successfully even if not handled.	Program will not compile unless handled or declared.
Usage Context	Used for programming bugs that can be avoided with better coding practices.	Used for situations that are expected and need to be explicitly addressed.

Summary:

- **Runtime exceptions** are unchecked and indicate programming bugs, whereas **compile-time exceptions** are checked and need explicit handling.

Here is a revised table incorporating the role of the `final` and `static` keywords explicitly in the difference between early and late binding:

Feature	Early Binding	Late Binding
Definition	Method or function call is resolved at compile time.	Method or function call is resolved at runtime.
Other Name	Static Binding	Dynamic Binding
Method Overloading	Supports method overloading	Does not support method overloading
Method Overriding	Does not support method overriding	Supports method overriding
Used With	`static` and `final` methods, as they cannot be overridden	Non-static methods and non-final methods, as they can be overridden
Efficiency	More efficient (faster execution)	Less efficient (slower execution)
When Binding Happens	During compile time	During runtime
Polymorphism	Does not support runtime polymorphism	Supports runtime polymorphism
Relation to `static`	Always uses early binding since `static` methods belong to the class, not instances	Not applicable to `static` methods, only instance methods
Relation to `final`	`final` methods are bound early since they cannot be overridden	Not applicable to `final` methods, only methods that can be overridden

This version clearly distinguishes the role of the `static` and `final` keywords in early and late binding. Static methods and final methods always use early binding since they cannot be overridden, while late binding occurs with non-static, non-final methods that are eligible for overriding.

Difference Between Early Binding and Late Binding:

Aspect	**Early Binding (Static Binding)**	**Late Binding (Dynamic Binding)**

Definition	Method calls are resolved at compile time.	Method calls are resolved at runtime.
Binding Mechanism	Done by the compiler during the compilation phase.	Done by the JVM during runtime.
Methods Involved	Applied to `static`, `private`, and `final` methods.	Applied to overridden methods in polymorphism.
Performance	Faster due to compile-time resolution.	Slower because it requires runtime resolution.
Flexibility	Less flexible; method implementation is fixed at compile time.	More flexible; depends on the actual object type at runtime.
Inheritance Role	Does not support polymorphism.	Supports polymorphism through overridden methods.
Example	Calling a `static` method or accessing a `final` method.	Calling an overridden method through a parent class reference.
Usage Context	Used for methods that are not meant to change or require polymorphism.	Used in scenarios where polymorphic behavior is required.

Key Points:

- **Static Methods**: Always use early binding because they belong to the class, not the object.
- **Final Methods**: Use early binding as they cannot be overridden, ensuring compile-time resolution.
- **Polymorphism**: Demonstrates late binding with overridden methods called dynamically based on the object type.

Syllabus

Package

- A **java package** is a group of similar types of classes, interfaces and sub-packages.

- Package in java can be categorized in two form, built-in package and user-defined package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Compiling:

javac -d directoryname filename

⇒ javac -d . PackageDemo.java

Running the program:

java packagename.Classname

⇒ java myPack.PackageDemo

Here are the benefits of using packages in Java:

1. **Organized Code Structure**

   Packages help group related classes and interfaces together, making code easier to organize and manage.

2. **Avoids Name Conflicts**

   Using packages ensures that class names in different packages do not conflict, even if they are the same.

3. **Access Protection**

   Packages provide access control mechanisms using `public`, `protected`, `default`, and `private` access modifiers, allowing better encapsulation.

4. **Reusability**

   Classes in a package can be reused across multiple projects or parts of the same application.

5. **Namespace Management**

   Packages act as namespaces, making it easier to locate and use classes.

6. **Easier Maintenance**

Organizing code into packages improves readability and makes large projects easier to maintain.

7. **Standardized Structure**

   Packages allow for a structured hierarchy, facilitating better project organization and collaboration.

Java's standard library is organized into packages (e.g., `java.util`, `java.io`), making it convenient to use pre-defined classes and methods.

1. **Built-in Libraries**

Access modifier

Private:

- The access level of a private modifier is only within the class.
- It cannot be accessed from outside the class.

Default:

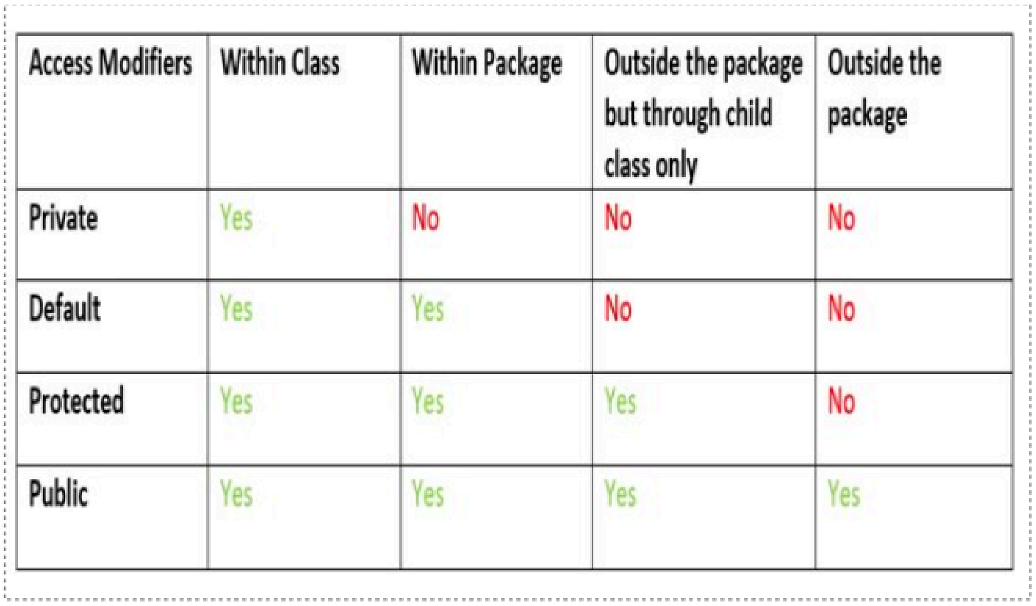
- The access level of a default modifier is only within the package.
- It cannot be accessed from outside the package. If we do not specify any access level, it will be the default.

Protected:

- The access level of a protected modifier is within the package and outside the package through child class.
- If you do not make the child class, it cannot be accessed from outside the package.

Public:

- The access level of a public modifier is everywhere.
- It can be accessed from within the class, outside the class, within the package and outside the package.



Access Modifiers	Within Class	Within Package	Outside the package but through child class only	Outside the package
Private	Yes	No	No	No
Default	Yes	Yes	No	No
Protected	Yes	Yes	Yes	No
Public	Yes	Yes	Yes	Yes

Untitled

File handling

Servlet

GUI

Here's a comparison of **Error** vs **Exception** in Java:

Aspect	**Error**	**Exception**
Definition	Serious problems that occur during runtime and typically cannot be recovered from.	Conditions that a program can handle and recover from.
Hierarchy	Subclass of `Throwable`, not a subclass of `Exception`.	Subclass of `Throwable` and `Exception`.
When occurs	Typically caused by the JVM or hardware issues (e.g., memory errors).	Typically caused by program logic or external issues (e.g., file not found).

Recoverability	Usually not recoverable (e.g., `OutOfMemoryError`, `StackOverflowError`).	Generally recoverable (e.g., `IOException`, `SQLException`).
Handling	Not required to handle (unchecked by the compiler).	Can and often should be handled using `try-catch` or declared with `throws`.
Examples	`OutOfMemoryError`, `StackOverflowError`, `VirtualMachineError`.	`NullPointerException`, `IOException`, `ArithmeticException`.
Impact on Program	Causes the program or JVM to crash or behave unexpectedly.	Can be caught, allowing the program to recover and continue execution.
Use Case	Represents conditions that should not occur under normal operation.	Represents conditions that might occur and should be expected in certain cases.

In summary, **Errors** represent severe system-level failures that are usually beyond the application's control, while **Exceptions** represent issues that can be anticipated and managed by the application.

Why is it important to handle exception

Handling exceptions is important because it helps prevent the program from crashing and allows you to manage unexpected errors, ensuring the program can recover or fail gracefully. It also makes debugging easier by pinpointing issues and preventing data loss or inconsistent states.

How to create custom exception in java

Here are the steps to create a custom exception in Java:

1. **Extend the `Exception` class**: Create a class that extends the `Exception` class (or `RuntimeException` for unchecked exceptions).
2. **Create a constructor**: Provide one or more constructors to pass error messages or other details to the custom exception.

Example:

```
// Step 1: Extend Exception class
public class MyCustomException extends Exception {

    // Step 2: Create constructor
    public MyCustomException(String message) {
        super(message);
    }
}
```

Now, you can throw and handle `MyCustomException` like any other exception.

what is user defined exception in java? What are different techniques to handle exception ?

User-defined exception in Java:

A **user-defined exception** is a custom exception class created by a programmer to handle specific scenarios that are not covered by Java's built-in exceptions. It allows you to create meaningful and domain-specific error messages.

Different techniques to handle exceptions in Java:

1. **Try-Catch Block**: Wraps the code that may throw an exception. The `catch` block handles the exception if it occurs.

   ```
   try {
       // risky code
   } catch (ExceptionType e) {
       // handling code
   }
   ```

2. **Finally Block**: Executes code after try-catch, regardless of whether an exception was thrown or not.

   ```
   finally {
       // cleanup code
   ```

```
}
```

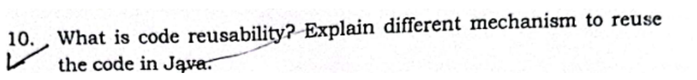
3. **Throw Keyword**: Used to explicitly throw an exception.

```
throw new MyCustomException("Error message");
```

4. **Throws Keyword**: Declares an exception in the method signature, indicating it must be handled by the caller.

```
public void myMethod() throws MyCustomException {
    // risky code
}
```

Exception handling in Java is a mechanism to manage runtime errors, ensuring the normal flow of the application. It allows you to catch and handle exceptions to prevent the program from crashing and manage errors gracefully.


10. What is code reusability? Explain different mechanism to reuse the code in Java.

Code reusability refers to the ability to reuse existing code components across different parts of a program or in different programs. In Java, there are several mechanisms to achieve code reusability:

- **Inheritance**: Allows a class to inherit properties and methods from another class, enabling code reuse through parent-child relationships.
- **Interfaces**: Provides a contract for classes to implement specific methods, promoting code reuse through standardized behavior.
- **Composition**: Allows building complex objects by combining simpler objects, enabling code reuse through object relationships.
- **Packages**: Organizes related classes and interfaces, making them easily reusable across different programs.
- **Methods**: Creating reusable methods that can be called from different parts of the program.

Code Reusability:

Code reusability refers to the practice of using existing code to perform common functions or logic in multiple parts of an application or even across different applications. By reusing code, developers can avoid redundancy, reduce development time, minimize errors, and ensure consistency in functionality.

Mechanisms to Reuse Code in Java:

Java offers several mechanisms to promote code reusability, each serving different purposes:

1. Inheritance:

- **Definition**: Inheritance is a key feature of Object-Oriented Programming (OOP) that allows a new class (subclass) to inherit fields and methods from an existing class (superclass).
- **Reusability**: It enables the reuse of code written in the parent class without duplicating it in the child class. Subclasses can also add new functionality or override inherited methods.
- **Example**:

```
class Animal {
    void eat() {
        System.out.println("This animal eats.");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}
```

- **Key Point**: The `Dog` class can reuse the `eat()` method from the `Animal` class while having its own `bark()` method.

2. Composition (Has-a Relationship):

- **Definition**: Composition involves using objects of one class within another class, allowing reuse of code by delegation.
- **Reusability**: Instead of inheriting methods, a class contains another class as a member, thereby reusing its functionality without an inheritance hierarchy.
- **Example**:

```
class Engine {
    void start() {
        System.out.println("Engine started.");
    }
}

class Car {
    Engine engine = new Engine(); // Composition

    void startCar() {
        engine.start(); // Reusing Engine functionality
        System.out.println("Car started.");
    }
}
```

- **Key Point**: The `Car` class reuses the `Engine` class functionality through composition.

3. Interfaces:

- **Definition**: An interface in Java is a reference type that contains abstract methods (until Java 8, which introduced default methods).
- **Reusability**: By implementing an interface, multiple classes can use the same set of methods, ensuring consistency and reusability across different classes.
- **Example**:

```
interface Drawable {
   void draw(); // abstract method
}

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

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

- **Key Point**: Both `Circle` and `Square` classes reuse the `draw()` method from the `Drawable` interface.

4. Abstract Classes:

- **Definition**: An abstract class is a class that cannot be instantiated and is designed to be subclassed. It may contain both abstract methods (without implementation) and concrete methods (with implementation).
- **Reusability**: Abstract classes allow common functionality to be shared among subclasses while still allowing for individual specialization.
- **Example**:

```
abstract class Animal {
   abstract void sound();

   void eat() {
      System.out.println("This animal eats.");
   }
}
```

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

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

- **Key Point**: `Dog` and `Cat` reuse the `eat()` method from `Animal` while providing their own implementation of `sound()`.

5. Method Overloading and Overriding:

- **Definition**: Overloading allows multiple methods with the same name but different parameters, while overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass.
- **Reusability**: Overloading and overriding allow developers to reuse method names and logic, adapting them to different contexts.
- **Example (Overriding)**:

```
class Animal {
    void sound() {
        System.out.println("Some sound.");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks.");
    }
}
```

- **Key Point**: The `Dog` class reuses and customizes the `sound()` method by overriding the base implementation from `Animal`.

6. Packages:

- **Definition**: A package is a namespace that organizes classes and interfaces. Java's standard library (`java.util`, `java.io`) provides reusable classes and methods.
- **Reusability**: By organizing related classes into packages, developers can easily reuse them across different projects by importing the package.
- **Example**:

```
import java.util.ArrayList;

public class Example {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Hello");
    }
}
```

- **Key Point**: The `ArrayList` class from the `java.util` package is reused in the `Example` class.

Yes, you can include **polymorphism** and **constructor overloading** as additional mechanisms to promote code reusability in Java. Here's how you can explain them:

Polymorphism:

Polymorphism is a core concept in Object-Oriented Programming (OOP) that allows one interface to be used for a general class of actions. In Java, it allows methods to behave differently based on the object that invokes them. There are two types of polymorphism:

1. **Compile-time Polymorphism (Method Overloading)**:
    - **Definition**: It occurs when multiple methods with the same name but different parameter lists exist in a class. The method that gets invoked depends on the method signature (i.e., the number or type of arguments passed).
    - **Reusability**: The same method name can be reused with different parameters, enhancing readability and reducing redundant code.
    - **Example**:

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

        double add(double a, double b) {
            return a + b;
        }
    }
    ```

    - **Key Point**: The `add()` method is reused for both `int` and `double` types.
2. **Runtime Polymorphism (Method Overriding)**:
    - **Definition**: It occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. At runtime, the correct method is called based on the object.
    - **Reusability**: Method overriding allows subclass-specific implementations while still reusing the method signature from the parent class.
    - **Example**:

```
class Animal {
    void sound() {
        System.out.println("Some sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
```

- **Key Point**: The `sound()` method is reused but behaves differently in each subclass.

Constructor Overloading:

Constructor overloading is a special type of **compile-time polymorphism** where a class can have more than one constructor with different parameter lists. This allows creating objects in multiple ways.

- **Definition**: Constructor overloading means having multiple constructors in a class, each differing in the number or type of parameters.
- **Reusability**: It allows the creation of objects with different initialization states using the same class. This reduces the need to write different methods for object creation.
- **Example**:

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

    // Constructor 1: No arguments
    Person() {
        this.name = "Unknown";
        this.age = 0;
    }

    // Constructor 2: With name
    Person(String name) {
        this.name = name;
        this.age = 0;
    }

    // Constructor 3: With name and age
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

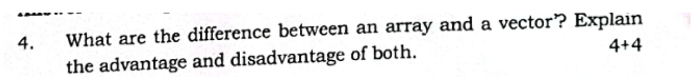
- **Key Point**: The `Person` class reuses the constructor name with different parameters, allowing object creation with or without initial values.

Conclusion:

Both **polymorphism** and **constructor overloading** enhance **code reusability** in Java. Polymorphism allows methods to behave differently based on the object, while constructor overloading provides flexibility in object creation. These techniques, along with others like inheritance and interfaces, help in writing more flexible and maintainable code.

Conclusion:

Java offers several mechanisms like inheritance, composition, interfaces, abstract classes, method overloading/overriding, and packages to encourage code reusability. These mechanisms help in reducing duplication, improving code quality, and maintaining consistency across the application. By leveraging these techniques, developers can write more efficient, maintainable, and scalable code.


4. What are the difference between an array and a vector? Explain the advantage and disadvantage of both. 4+4

Differences Between Array and Vector

Here's a comparison of **Array** and **Vector** in Java, along with their **advantages** and **disadvantages**:

Array:

Definition: An array is a fixed-size data structure that stores elements of the same type. The size of the array is specified at the time of creation and cannot be changed later.

Advantages:

1. **Faster Access**: Arrays provide fast access to elements via indexing, making them efficient for random access.
2. **Memory Efficiency**: Since arrays have a fixed size, they can be more memory-efficient than dynamic structures.
3. **Simplicity**: Arrays are simple to use, especially when the number of elements is known in advance.

Disadvantages:

1. **Fixed Size**: Once an array's size is defined, it cannot be changed, making it inflexible if the number of elements varies.
2. **Manual Resizing**: To resize an array, you need to manually create a new array and copy elements, which can be inefficient.
3. **Lack of Built-in Methods**: Arrays do not have built-in methods for adding, removing, or manipulating elements, unlike more advanced data structures like `ArrayList` or `Vector`.

Vector:

Definition: A `Vector` is a dynamic array that can grow or shrink in size as elements are added or removed. It is part of the Java Collection Framework.

Advantages:

1. **Dynamic Size**: Vectors can grow or shrink dynamically, which makes them flexible when the number of elements is unknown or changes frequently.
2. **Built-in Methods**: Vectors provide a variety of built-in methods, such as `add()`, `remove()`, `get()`, and `size()`, to manage elements efficiently.
3. **Thread-Safety**: Vectors are thread-safe by default, meaning they can be used in multi-threaded environments without additional synchronization.

Disadvantages:

1. **Slower Performance**: Since Vectors are synchronized for thread safety, their operations can be slower compared to other collections like `ArrayList`.
2. **Memory Overhead**: Due to their ability to resize dynamically, Vectors may have extra memory overhead to accommodate resizing.
3. **Outdated**: Vectors are considered somewhat outdated compared to newer dynamic data structures like `ArrayList`, which are more commonly used in modern Java development.

Summary Comparison:

Aspect	**Array**	**Vector**
Size	Fixed size, cannot grow or shrink	Dynamic size, can grow or shrink as needed
Access Speed	Fast, constant-time access via index	Slower due to synchronization and resizing
Memory Usage	Memory efficient but fixed	Higher memory usage due to dynamic resizing
Flexibility	Inflexible after creation	Flexible, adjusts size dynamically
Built-in Methods	No built-in methods for dynamic operations	Built-in methods for adding, removing, etc.
Thread Safety	Not thread-safe by default	Thread-safe by default

---

Conclusion:

- **Array** is suitable when you know the size in advance and require fast access with minimal overhead.
- **Vector** is useful for scenarios where the size may change over time, but its performance overhead may be a concern in high-performance applications. For most modern applications, `ArrayList` (a non-synchronized, dynamic array) is preferred over **Vector**.

String:

In Java, a **String** is an immutable sequence of characters. Once a String object is created, its value cannot be changed. If any modification is made, a new String object is created. Strings are widely used for handling text in Java, and their immutability ensures thread safety. However, frequent modifications to Strings can be inefficient due to the creation of new objects, leading to higher memory usage and slower performance.

StringBuffer:

StringBuffer is a mutable sequence of characters that allows modifications to its content without creating new objects. It is more efficient than **String** when performing multiple concatenations or modifications because it modifies the existing object instead of creating new ones. **StringBuffer** is thread-safe, meaning its methods are synchronized, making it suitable for use in multi-threaded environments, although this can introduce some performance overhead compared to **StringBuilder**.

StringBuilder:

StringBuilder is similar to **StringBuffer** in that it is a mutable sequence of characters, allowing efficient modification of strings. However, unlike **StringBuffer**, **StringBuilder** is not synchronized, making it faster in single-threaded scenarios. It is ideal for situations where thread safety is not a concern, and high performance is required, such as in simple text manipulation operations within a single thread. It offers methods like append(), insert(), and delete() to modify the string contents.

Here's a comparison between **ArrayList** and **LinkedList** in Java:

Aspect	ArrayList	LinkedList
Data Structure Type	Resizable array (array-based)	Doubly linked list
Memory Allocation	Contiguous memory block	Non-contiguous memory; each element is a node with references to the next and previous elements
Access Time	Faster for random access (constant time O(1))	Slower for random access (linear time O(n))
Insertion/Deletion	Slower for insertions/deletions (O(n)) at arbitrary positions because elements need to be shifted	Faster (O(1)) for insertions/deletions at the beginning or middle, but still O(n) for searching
Performance for Searching	Better performance for searching elements because of direct index access	Slower for searching since it requires traversing the list node by node
Memory Overhead	Lower memory overhead (only stores the element)	Higher memory overhead due to storing references (next and previous pointers) for each element
Resizing	Automatically resizes as elements are added, but involves copying elements to a new array	No resizing needed, as nodes are linked dynamically
Use Case	Ideal for scenarios where fast access and fewer insertions/deletions are required	Better for scenarios where frequent insertions and deletions are needed

Summary:

- **ArrayList** is suitable for cases requiring frequent random access and fewer insertions or deletions.
- **LinkedList** is more efficient when frequent insertions and deletions occur, especially at the beginning or middle of the list. However, it has higher memory overhead and slower random access.

Tree Interface:

The **Tree** interface in Java is a part of the `java.util` package and represents a collection of elements in a hierarchical (tree-like) structure. It extends the **NavigableSet** interface, meaning it is a type of set that supports ordered elements and provides operations to navigate through elements efficiently.

In Java, the **TreeSet** class implements the **NavigableSet** interface, which in turn extends the **Set** interface and provides a collection that uses a **red-black tree** for storing its elements.

Key Features of Tree Interface:

1. **Sorted Order**: Elements are stored in a natural order (or according to a custom comparator).
2. **No Duplicates**: It does not allow duplicate elements.
3. **Navigation Operations**: Provides methods for navigating the tree, such as finding the first, last, or elements greater/less than a given value.

Common Methods in TreeSet (Implementing Tree Interface):

- `first()`: Returns the first (lowest) element.
- `last()`: Returns the last (highest) element.
- `higher(E e)`: Returns the element strictly greater than the given element.
- `lower(E e)`: Returns the element strictly less than the given element.
- `ceiling(E e)`: Returns the least element greater than or equal to the given element.
- `floor(E e)`: Returns the greatest element less than or equal to the given element.

Example:

```
import java.util.*;

public class TreeSetExample {
    public static void main(String[] args) {
        TreeSet<Integer> treeSet = new TreeSet<>();

        // Adding elements
        treeSet.add(10);
        treeSet.add(20);
        treeSet.add(30);
        treeSet.add(5);

        // Display the TreeSet
        System.out.println("TreeSet: " + treeSet);  // Output: [5, 10, 20, 30]

        // Navigation methods
        System.out.println("First Element: " + treeSet.first());  // Output: 5
        System.out.println("Last Element: " + treeSet.last());    // Output: 30
        System.out.println("Element higher than 15: " + treeSet.higher(15));  // Ou
tput: 20
    }
}
```

Set Interface:

The **Set** interface in Java is a part of the `java.util` package and represents a collection that does not allow duplicate elements. It is a generic interface that is implemented by various classes, such as **HashSet**, **LinkedHashSet**, and **TreeSet**.

Key Features of Set Interface:

1. **No Duplicates**: A Set does not allow duplicate elements.
2. **Unordered**: Elements in a Set are not stored in any particular order (except for some implementations like **LinkedHashSet** and **TreeSet**).
3. **Efficiency**: **HashSet** is highly efficient for operations like add, remove, and contains (constant time complexity on average).

Common Methods in Set:

- `add(E e)`: Adds an element to the set if it's not already present.
- `remove(Object o)`: Removes the specified element from the set.
- `contains(Object o)`: Checks if the set contains the specified element.

- `size()`: Returns the number of elements in the set.
- `clear()`: Removes all elements from the set.

Example:

```
import java.util.*;

public class HashSetExample {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();

        // Adding elements
        set.add("Apple");
        set.add("Banana");
        set.add("Orange");
        set.add("Apple");  // Duplicate element, will not be added

        // Display the Set
        System.out.println("Set: " + set);  // Output: [Apple, Banana, Orange]

        // Operations
        System.out.println("Size of Set: " + set.size());  // Output: 3
        System.out.println("Contains 'Apple': " + set.contains("Apple"));  // Outpu
t: true
    }
}
```

Key Differences Between Tree and Set:

Feature	**Tree (TreeSet)**	**Set (HashSet, LinkedHashSet, TreeSet)**
Ordering	Elements are ordered (natural or custom comparator).	No inherent ordering (except LinkedHashSet and TreeSet).
Duplicates	Does not allow duplicates.	Does not allow duplicates.

Implementation	Uses a red-black tree (in TreeSet).	Implemented by HashSet (hash table) or TreeSet (red-black tree).
Performance	Slower compared to HashSet for basic operations (due to sorting).	HashSet is faster for basic operations, LinkedHashSet maintains insertion order.
Navigation	Supports navigation (higher, lower, first, last).	Does not support navigation (except TreeSet).

Conclusion:

- **TreeSet** implements the **Tree** interface, providing an ordered collection of elements with fast lookup and navigation.

Set is a more general interface that ensures uniqueness of elements, with implementations like HashSet (unordered) and TreeSet (ordered).

Array

In Java, an array is a collection of elements of the same data type stored in contiguous memory locations. It has a fixed size that is defined when the array is created. Each element in the array is accessed using an index, starting from 0. Arrays provide a way to store multiple values in a single variable rather than declaring separate variables for each value. You can initialize an array either with a fixed size or directly with values. Arrays can be of any data type, including primitive types and objects.

General Syntax:

```
// Declaration
datatype[] arrayName;   // or datatype arrayName[];

// Initialization with size
arrayName = new datatype[size];
```

```
// Initialization with values
datatype[] arrayName = {value1, value2, value3};
```

You can access elements using `arrayName[index]` where `index` is the position of the element.

```
import java.util.Scanner;

class Circle {
    // Instance variable for radius
    private double radius;

    // Constructor to initialize the radius
    public Circle(double radius) {
        this.radius = radius;
    }

    // Method to calculate the area of the circle
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

public class AreaOfCircle {
    public static void main(String[] args) {
        // Create a Scanner object to read user input
        Scanner scanner = new Scanner(System.in);

        // Ask the user for the radius
        System.out.print("Enter the radius of the circle: ");
        double radius = scanner.nextDouble();

        // Create a Circle object with the given radius
        Circle circle = new Circle(radius);

        // Calculate and display the area
        double area = circle.calculateArea();
        System.out.println("The area of the circle with radius " + radius + " is: " + ar

        // Close the scanner
        scanner.close();
    }
}
```

<u>Time sufficient</u>