**JAVA FEATURES**

**1) Inheritance in Java:**

**Inheritance** is one of the core concepts of Object-Oriented Programming (OOP) in Java. It allows a class (subclass/child class) to **acquire the properties and behaviors (fields and methods)** of another class (superclass/parent class).

Syntax:

|  |
| --- |
| class Parent {  // fields and methods  }  class Child extends Parent {  // additional fields and methods  } |

Benefits of Inheritance:

* Code reusability
* Method overriding (runtime polymorphism)
* Hierarchical classification

Types of Inheritance in Java:

Java supports the following types of inheritance:

**1. Single Inheritance**

A class inherits from one superclass.

**Example:**

|  |
| --- |
| class Animal {  void sound() {  System.out.println("Animal makes a sound");  }  }  class Dog extends Animal {  void bark() {  System.out.println("Dog barks");  }  }  public class Main {  public static void main(String[] args) {  Dog d = new Dog();  d.sound(); // inherited  d.bark(); // own method  }  } |

**2. Multilevel Inheritance**

A class is derived from a class that is already derived from another class.

**Example:**

|  |
| --- |
| class Animal {  void sound() {  System.out.println("Animal sound");  }  }  class Dog extends Animal {  void bark() {  System.out.println("Dog barks");  }  }  class Puppy extends Dog {  void weep() {  System.out.println("Puppy weeps");  }  }  public class Main {  public static void main(String[] args) {  Puppy p = new Puppy();  p.sound(); // from Animal  p.bark(); // from Dog  p.weep(); // from Puppy  }  } |

**3. Hierarchical Inheritance**

Multiple classes inherit from a single parent class.

**Example:**

|  |
| --- |
| class Animal {  void eat() {  System.out.println("Animal eats");  }  }  class Dog extends Animal {  void bark() {  System.out.println("Dog barks");  }  }  class Cat extends Animal {  void meow() {  System.out.println("Cat meows");  }  }  public class Main {  public static void main(String[] args) {  Dog d = new Dog();  d.eat();  d.bark();  Cat c = new Cat();  c.eat();  c.meow();  }  } |

**4. Multiple Inheritance (Not supported using classes in Java)**

Java does **not support multiple inheritance with classes** to avoid ambiguity (Diamond Problem). However, it is supported using **interfaces**.

**Example using Interfaces:**

|  |
| --- |
| interface A {  void methodA();  }  interface B {  void methodB();  }  class C implements A, B {  public void methodA() {  System.out.println("Method A");  }  public void methodB() {  System.out.println("Method B");  }  }  public class Main {  public static void main(String[] args) {  C obj = new C();  obj.methodA();  obj.methodB();  }  } |

In Java, super and this are **keywords** used to refer to objects:

* this refers to the **current class instance**
* super refers to the **parent class instance**

**2)this Keyword:**

* **Refer to current class instance variable**
* **Invoke current class methods or constructors**
* **Pass current object as a parameter**

Example: Using this to differentiate local and instance variable

|  |
| --- |
| class Student {  int id;  String name;  Student(int id, String name) {  this.id = id; // refers to instance variable  this.name = name;  }  void display() {  System.out.println(id + " " + name);  }  }  public class Main {  public static void main(String[] args) {  Student s = new Student(101, "Rahul");  s.display();  }  } |

**3)super Keyword:**

* **Call parent class constructor**
* **Access parent class method**
* **Access parent class variable**

📌 Example: Using super to call parent class constructor and method

|  |
| --- |
| class Animal {  String color = "White";  Animal() {  System.out.println("Animal constructor");  }  void sound() {  System.out.println("Animal makes sound");  }  }  class Dog extends Animal {  String color = "Black";  Dog() {  super(); // calls Animal() constructor  System.out.println("Dog constructor");  }  void printColor() {  System.out.println(super.color); // refers to Animal's color  }  void sound() {  super.sound(); // calls Animal's sound method  System.out.println("Dog barks");  }  }  public class Main {  public static void main(String[] args) {  Dog d = new Dog();  d.printColor();  d.sound();  }  } |

**4) Polymorphism in Java:**

**Polymorphism** means **"many forms"**. In Java, polymorphism allows an object to behave differently based on the context.

There are two main types:

1. **Compile-time polymorphism (Method Overloading)**
2. **Runtime polymorphism (Method Overriding)**

**1. Method Overloading (Compile-time Polymorphism)**

**Definition:**  
Multiple methods in the same class with the **same name** but **different parameters** (type, number, or order).

|  |
| --- |
| class Calculator {  int add(int a, int b) {  return a + b;  }  double add(double a, double b) {  return a + b;  }  int add(int a, int b, int c) {  return a + b + c;  }  }  public class Main {  public static void main(String[] args) {  Calculator c = new Calculator();  System.out.println(c.add(5, 3)); // 8  System.out.println(c.add(2.5, 3.5)); // 6.0  System.out.println(c.add(1, 2, 3)); // 6  }  } |

**2. Method Overriding (Runtime Polymorphism)**

**Definition:**  
A **subclass provides its own implementation** of a method that is already defined in the **superclass**.

|  |
| --- |
| 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 a = new Dog(); // Polymorphism  a.sound(); // Calls Dog's sound() method  }  } |

**5) Abstract Class in Java**

**➤ What is it?**

An **abstract class** is a class that **cannot be instantiated**, and it may contain **abstract methods** (without a body) as well as **concrete methods** (with a body).

**🔹 Use when:**

* You want to **provide a common base class** with some **shared implementation**.
* You want to enforce **partial abstraction** (some methods are defined, some are abstract).

Example:

|  |
| --- |
| abstract class Animal {  abstract void sound(); // abstract method  void sleep() { // concrete method  System.out.println("Sleeping...");  }  }  class Dog extends Animal {  void sound() {  System.out.println("Dog barks");  }  }  public class Main {  public static void main(String[] args) {  Animal a = new Dog();  a.sound(); // Dog barks  a.sleep(); // Sleeping...  }  } |

**6) Interface in Java**

**➤ What is it?**

An **interface** is a **fully abstract** class (until Java 7), meaning all methods are **implicitly abstract and public**. From **Java 8 onwards**, it can also have **default and static methods with body**.

**🔹 Use when:**

* You want to define a **contract** (set of methods) that multiple classes can implement.
* You want **full abstraction** and **multiple inheritance** of types.

Example:

|  |
| --- |
| interface Vehicle {  void start(); // implicitly public and abstract  }  class Car implements Vehicle {  public void start() {  System.out.println("Car is starting");  }  }  public class Main {  public static void main(String[] args) {  Vehicle v = new Car();  v.start(); // Car is starting  }  } |

**7) What is Encapsulation in Java?**

**Encapsulation** is the process of **wrapping data (variables) and code (methods)** together into a single unit (class) and restricting direct access to some components of an object.

**🔒 Achieved by:**

1. Declaring variables **private**
2. Providing **public getters and setters** to access/update them

Example: Encapsulation

|  |
| --- |
| class Student {  private int id;  private String name;  // Getter and Setter  public int getId() {  return id;  }  public void setId(int id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  }  public class Main {  public static void main(String[] args) {  Student s = new Student();  s.setId(1);  s.setName("Ravi");  System.out.println(s.getId() + " " + s.getName());  }  } |

**Benefits:**

* Data hiding
* Controlled access
* Makes class easy to maintain

**8) What is a POJO Class?**

**POJO (Plain Old Java Object)** is a simple Java class with:

* Private fields
* Public constructors
* Public getters/setters
* **No business logic**, no inheritance, no annotations required

Example:

|  |
| --- |
| public class Product {  private int id;  private String name;  // Constructor  public Product(int id, String name) {  this.id = id;  this.name = name;  }  // Getters & Setters  public int getId() { return id; }  public void setId(int id) { this.id = id; }  public String getName() { return name; }  public void setName(String name) { this.name = name; }  } |

POJOs are often used in **data transfer**, **ORM (like Hibernate)**, etc.

**9) What is a Java Bean Class?**

A **JavaBean** is a **special type of POJO** that follows **strict rules**:

| **Feature** | **JavaBean Rule** |
| --- | --- |
| No-arg constructor | Required |
| Private variables | Required |
| Public getters/setters | Required |
| Serializable | Optional (recommended for frameworks) |

Example:

|  |
| --- |
| public class EmployeeBean implements java.io.Serializable {  private int id;  private String name;  // No-arg constructor  public EmployeeBean() {}  // Getters & Setters  public int getId() { return id; }  public void setId(int id) { this.id = id; }  public String getName() { return name; }  public void setName(String name) { this.name = name; }  } |

JavaBeans are commonly used in **Java frameworks** like **Spring**, **JSP**, etc.

**10) What is an Exception in Java?**

An **exception** is an **event** that occurs during program execution and **disrupts the normal flow** of instructions.

All exceptions are objects derived from the class:  
java.lang.Throwable

**2. Types of Exceptions**

|  |
| --- |
| **Throwable**  **/ \**  **Exception Error**  **/ \**  **Checked Unchecked** |

**a) Checked Exceptions**

* Handled at compile time
* Must be caught or declared using throws
* Examples:
  + IOException
  + SQLException
  + FileNotFoundException

Example:

|  |
| --- |
| import java.io.\*;  public class CheckedExample {  public static void main(String[] args) throws IOException {  BufferedReader reader = new BufferedReader(new FileReader("file.txt"));  System.out.println(reader.readLine());  }  } |

**b) Unchecked Exceptions**

* **Handled at runtime**
* Compiler does **not check**
* Examples:
  + ArithmeticException
  + NullPointerException
  + ArrayIndexOutOfBoundsException

Example:

|  |
| --- |
| public class UncheckedExample {  public static void main(String[] args) {  int a = 10 / 0; // ArithmeticException  }  } |

**Exception Hierarchy**

|  |
| --- |
| Object  └── Throwable  ├── Exception (Checked)  │ ├── IOException  │ ├── SQLException  │ └── etc.  └── RuntimeException (Unchecked)  ├── ArithmeticException  ├── NullPointerException  └── etc. |

**11) Exception Handling in Java**

Keywords used:

| **Keyword** | **Purpose** |
| --- | --- |
| try | Wraps risky code |
| catch | Catches and handles exception |
| finally | Executes regardless of exception |
| throw | Used to throw an exception manually |
| throws | Declares that a method might throw an exception |

Example: Try-Catch-Finally

|  |
| --- |
| public class ExceptionHandlingExample {  public static void main(String[] args) {  try {  int a = 10 / 0;  } catch (ArithmeticException e) {  System.out.println("Cannot divide by zero");  } finally {  System.out.println("This block always executes");  }  }  } |

**Example: throw and throws**

|  |
| --- |
| public class ThrowExample {  static void validateAge(int age) throws ArithmeticException {  if (age < 18)  throw new ArithmeticException("Not eligible to vote");  else  System.out.println("Eligible to vote");  }  public static void main(String[] args) {  validateAge(16); // will throw exception  }  } |

**12) What is Collections in Java?**

* The Java Collections Framework (JCF) is a unified architecture for storing and manipulating groups of objects.
* It provides interfaces (like List, Set, Map) and classes (like ArrayList, HashSet, HashMap) to handle collections.

**1. List – Ordered Collection (Allows Duplicates)**

* Elements are ordered and **can be accessed by index**
* **Duplicates allowed**

**📌 Example using ArrayList:**

|  |
| --- |
| import java.util.\*;  public class ListExample {  public static void main(String[] args) {  List<String> fruits = new ArrayList<>();  fruits.add("Apple");  fruits.add("Banana");  fruits.add("Apple"); // duplicate allowed  System.out.println(fruits); // [Apple, Banana, Apple]  System.out.println(fruits.get(1)); // Banana  }  } |

**Common Implementations:**

* ArrayList (fast for reading)
* LinkedList (fast for insertion/deletion)

**2. Set – Unordered Collection (No Duplicates)**

* No duplicates allowed
* No indexing

**📌 Example using HashSet:**

|  |
| --- |
| import java.util.\*;  public class SetExample {  public static void main(String[] args) {  Set<String> colors = new HashSet<>();  colors.add("Red");  colors.add("Green");  colors.add("Red"); // duplicate ignored  System.out.println(colors); // [Red, Green]  }  } |

**Common Implementations**:

* HashSet (unordered)
* LinkedHashSet (insertion order)
* TreeSet (sorted)

1. **Map – Key-Value Pairs**

* Stores **key-value pairs**
* Keys are **unique**, values can be **duplicate**

**📌 Example using HashMap:**

|  |
| --- |
| import java.util.\*;  public class MapExample {  public static void main(String[] args) {  Map<Integer, String> students = new HashMap<>();  students.put(101, "Ravi");  students.put(102, "Priya");  students.put(101, "Kumar"); // overwrites key 101  System.out.println(students); // {101=Kumar, 102=Priya}  System.out.println(students.get(102)); // Priya  }  } |

**Common Implementations**:

* HashMap (unordered)
* LinkedHashMap (insertion order)
* TreeMap (sorted by keys)

**13) What is a Subclass in Java?**

* A **subclass** (also called **child class** or **derived class**) is a class that **inherits** properties (fields and methods) from another class, known as the **superclass** (parent class).

|  |
| --- |
| class Superclass { // fields and methods  }  class Subclass extends Superclass {  // additional fields and methods  } |

Example:

|  |
| --- |
| // Superclass  class Animal {  void eat() {  System.out.println("This animal eats food.");  }  }  // Subclass  class Dog extends Animal {  void bark() {  System.out.println("The dog barks.");  }  }  public class SubclassExample {  public static void main(String[] args) {  Dog d = new Dog();  d.eat(); // inherited method  d.bark(); // subclass method  }  } |

**Key Points:**

* Use extends keyword to create a subclass.
* A subclass **inherits all non-private members** of the superclass.
* A subclass can **add its own methods/fields**.
* A subclass can also **override** superclass methods (polymorphism).

**14) Terminal and Intermediate Operators in Java (Streams API)**

* In Java (especially Java 8 and above), **Streams** allow you to process collections in a functional style.  
  Stream operations are **either intermediate or terminal**.

**1. Intermediate Operators**

* **Return a Stream**
* Are **lazy** – they don’t process anything until a terminal operation is invoked.
* Used for **transforming** or **filtering** data.

**Common Intermediate Operators:**

| **Operator** | **Description** |
| --- | --- |
| filter() | Select elements based on a condition |
| map() | Transform each element |
| sorted() | Sort the elements |
| distinct() | Remove duplicates |
| limit(n) | Limit to first n elements |
| skip(n) | Skip first n elements |

**Example:**

|  |
| --- |
| package intermediateoperations;  import java.util.Arrays;  import java.util.List;  import java.util.stream.Collectors;  public class MainExample {  public static void main(String[] args) {  List<List<Integer>> numbers = Arrays.asList(  Arrays.asList(1, 9, 3, 2, 3, 4, 4, 5, 6, 7, 8, 9, 10),  Arrays.asList(1, 2, 3, 5, 7)  );  // Filter even numbers, map, skip, sort, distinct, and limit  List<Integer> evenNumbers = numbers.stream()  .flatMap(List::stream)//nested list to single stream  .peek(n -> System.out.println("Before filtering: " + n))// Peek to see the elements before filtering  .filter(n -> n % 2 == 0)//2,4,4,6,8,10,2  .map(n -> n \* 3)//6,12,12,18,24,30,6  .skip(4)//skip first 4 elements=>24,30,6  .sorted()//6,24,30  .distinct()//all are unique  .limit(2)//take only 2 elements=>6,24  .collect(Collectors.toList());  System.out.println(evenNumbers);  }  } |

**2. Terminal Operators**

* **Produce a result** or a **side-effect**
* **Triggers** the processing of the stream pipeline

**Common Terminal Operators:**

| **Operator** | **Description** |
| --- | --- |
| forEach() | Perform action for each element |
| collect() | Collect result into list, set, etc. |
| count() | Count number of elements |
| reduce() | Combine elements into a single result |
| anyMatch() | Check if any element matches condition |
| findFirst() | Return the first matching element |

**Example:**

|  |
| --- |
| package TerminalOperations;  import java.util.\*;  import java.util.stream.\*;  public class MainExample {  public static void main(String[] args) {  List<Integer> numbers = Arrays.*asList*(5, 3, 7, 2, 8, 10, 1, 4, 6, 9);  // 1. anyMatch() - checks if any number is even  boolean anyEven = numbers.stream().anyMatch(n -> n % 2 == 0);  System.***out***.println("Any even number? " + anyEven);  // 2. allMatch() - checks if all numbers are less than 20  boolean allLessThan20 = numbers.stream().allMatch(n -> n < 20);  System.***out***.println("All numbers < 20? " + allLessThan20);  // 3. noneMatch() - checks if no number is negative  boolean noneNegative = numbers.stream().noneMatch(n -> n < 0);  System.***out***.println("No negative numbers? " + noneNegative);  // 4. collect() - collect even numbers into a List  List<Integer> evens = numbers.stream()  .filter(n -> n % 2 == 0)  .collect(Collectors.*toList*());  System.***out***.println("Even numbers collected: " + evens);  // 5. count() - count of numbers > 5  long countGreaterThan5 = numbers.stream().filter(n -> n > 5).count();  System.***out***.println("Count > 5: " + countGreaterThan5);  // 6. findAny() - returns any number  Optional<Integer> any = numbers.stream().findAny();  System.***out***.println("Any number: " + any.orElse(null));  // 7. findFirst() - find the first number  Optional<Integer> first = numbers.stream().findFirst();  System.***out***.println("First number: " + first.orElse(null));  // 8. forEach() - print all numbers  System.***out***.print("All numbers: ");  numbers.stream().forEach(n -> System.***out***.print(n + " "));  System.***out***.println();  // 9. min() - find the minimum number  Optional<Integer> min = numbers.stream().min(Integer::compareTo);  System.***out***.println("Minimum number: " + min.orElse(null));  // 10. max() - find the maximum number  Optional<Integer> max = numbers.stream().max(Integer::compareTo);  System.***out***.println("Maximum number: " + max.orElse(null));  // 11. reduce() - sum of all numbers  int sum = numbers.stream().reduce(0, Integer::*sum*);  System.***out***.println("Sum using reduce(): " + sum);  // 12. toArray() - convert to array and print  Integer[] array = numbers.stream().toArray(Integer[]::new);  System.***out***.println("Array from stream: " + Arrays.*toString*(array));  }  } |