CORE JAVA

Q. Why should one learn java?

* **Platform Independence** – (Write once, run anywhere - WORA).
* **High Demand and most popular** – Used in enterprise applications, Android development, and backend systems.
* **Object-Oriented** – Encourages modular and reusable code.
* **Rich API and community support.**
* **Multi-threading Support** – Efficient handling of multiple tasks.
* **Provides security**- because code runs on virtual machine.

Q. What is Java?

* Java is a high-level, object-oriented programming language developed by Sun Microsystems (now owned by Oracle).
* It follows Write Once, Run Anywhere (WORA), meaning compiled Java code can run on any platform with a Java Virtual Machine (JVM).
* Its features include🡪

1. **Platform Independent** – Runs on any OS with JVM.
2. **Object-Oriented** – Follows OOP principles like Encapsulation and Inheritance.
3. **Simple & Easy to Learn** – Has a clean syntax similar to C++.
4. **Secure** – Provides built-in security mechanisms.
5. **Robust & Reliable** – Features strong memory management and exception handling.
6. **Multithreading** – Supports concurrent execution of multiple tasks.
7. **High Performance** – Uses JIT compiler for faster execution.
8. **Distributed Computing** – Supports networking and remote method invocation (RMI).
9. **Dynamic & Extensible** – Allows dynamic loading of classes.
10. **Automatic Memory Management** – Uses Garbage Collection to free unused memory.

**History of JAVA 🡪**

- Java was created by James Gosling and his team at Sun Microsystems in 1991.

- Originally called "Oak".Later renamed Java, inspired by Indonesian coffee.

- Java 1.0 was released in May 1995 by Sun Microsystems.

- Introduced the "Write Once, Run Anywhere" (WORA) concept.

- Developed with vision of backward compatibility.

Q. Why is Java Platform Independent?

Java is platform-independent because it follows the **"Write Once, Run Anywhere" (WORA)** principle. This means Java programs can run on any operating system without modification.

Q. How is Java Platform Independent?

1. **Compilation to Bytecode**: Java code (.java file) is compiled by the Java Compiler (javac) into bytecode (.class file). This bytecode is not specific to any operating system.
2. **Java Virtual Machine (JVM)**: The bytecode is executed by the JVM, which is platform-specific.Each OS has its own JVM implementation, but all JVMs understand the same bytecode.
3. **Abstraction from Hardware & OS**: Java programs do not directly interact with the OS. Instead, the JVM translates bytecode into machine-specific instructions.

**Java Buzzwords** 🡪

Robust

Multithreaded

Architecture Neutral

Intrepreted and High Performance

Distributed

**What are the different types of Java applications?**

1. **Standalone Applications (Desktop Applications)** 🖥️
   * Runs on a single system without requiring a network.
   * Uses Swing, AWT, or JavaFX for GUI.
   * Example: Media players, antivirus software.
2. **Web Applications** 🌍
   * Runs on web servers and uses Servlets, JSP, or Spring Boot.
   * Example: E-commerce sites, banking applications.
3. **Enterprise Applications** 🏢
   * Large-scale, distributed applications using Java EE (Jakarta EE).
   * Example: ERP systems, CRM software.
4. **Mobile Applications** 📱
   * Android apps developed using Java + Android SDK.
   * Example: WhatsApp, Instagram (earlier versions).
5. **Distributed Applications** 🌐
   * Runs across multiple networked computers using RMI, EJB, CORBA.
   * Example: Cloud-based applications, big data frameworks.
6. **Embedded Systems** 🔧
   * Java is used in IoT devices, Smart TVs, and ATMs.
   * Example: Set-top boxes, embedded controllers.
7. **Scientific Applications** 🔬
   * Used in complex scientific and mathematical computations.
   * Example: MATLAB, data analysis software.

**What is a Java Package?**

A Java package is a collection of related classes, interfaces, and sub-packages grouped together. It helps in organizing code, avoiding name conflicts, and improving reusability. Example: java.util, java.io, javax.swing

✅ **Types of Packages**:

1. **Built-in Packages** – Provided by Java (e.g., java.util, java.io).
2. **User-defined Packages** – Created by programmers to organize custom classes.

**Why are Java Packages Important?**

|  |  |
| --- | --- |
| Feature | Description |
| 1. Code Organization | Helps structure large projects by grouping related classes. |
| 2. Name Conflict Resolution | Prevents class name conflicts by defining unique namespaces. |
| 3. Reusability | Classes in a package can be reused across different programs. |
| 4. Access Control | Allows better access control using access modifiers (public, private, etc.). |
| 5. Encapsulation | Hides implementation details and exposes only necessary functionality. |

Eclipse Working🡪

Ctrl+Shift+F 🡪 Beautify the code.

Rightclick-Source-Organize Imports 🡪To import the classes needed for file automatically

* **JDK** = JRE + Development Tools (for developers).
* **JRE** = JVM + Java Libraries (for running Java apps).
* **JVM** = Core part of JRE that actually runs Java programs.

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | JDK (Java Development Kit) | JRE (Java Runtime Environment) | JVM (Java Virtual Machine) |
| Definition | A software development kit (SDK) that includes tools for developing and compiling Java programs. | A runtime environment that provides libraries and resources to run Java applications. | A virtual machine that executes Java bytecode. |
| Purpose | Used for developing, compiling, debugging, and running Java applications. | Used for running Java applications but does not include development tools. | Provides the runtime environment for executing Java bytecode. |
| Includes | Contains JRE + development tools (compiler, debugger, etc.). | Contains JVM + Java libraries and runtime components. | Only responsible for running Java bytecode. |
| Components | JRE, Java compiler (javac), debugger (jdb), JavaDoc, and other development tools. | JVM, Java class libraries, and other runtime dependencies. | Class Loader, Memory Management (Heap & Stack), Execution Engine, JIT Compiler, etc. |
| Used By | Developers who write, compile, and debug Java applications. | End-users who only need to run Java applications. | Internally used by JRE and JDK to execute Java applications. |
| Installation Needed? | Yes, required for Java development. | Yes, required for running Java applications. | No separate installation; it comes with JRE/JDK. |

HELLO WORLD PROGRAM 🡪

public class Main {

public static void main(String[] args) {

System.out.println("Hello World");

}

}

Data types are divided into two groups:

* Primitive data types -byte, short, int, long, float, double, boolean and char
* Non-primitive data types - [String](https://www.w3schools.com/java/java_strings.asp), [Arrays](https://www.w3schools.com/java/java_arrays.asp) and [Classes](https://www.w3schools.com/java/java_classes.asp)

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Type** | **Size** | **Default Value** | **Example** |
| byte | 1 byte | 0 | byte b = 10; |
| short | 2 bytes | 0 | short s = 100; |
| int | 4 bytes | 0 | int x = 500; |
| long | 8 bytes | 0L | long y = 10000L; |
| float | 4 bytes | 0.0f | float pi = 3.14f; |
| double | 8 bytes | 0.0d | double d = 99.99; |
| char | 2 bytes | '\u0000' | char ch = 'A'; |
| boolean | 1 bit | false | boolean flag = true; |

|  |  |  |
| --- | --- | --- |
| Feature | Primitive Data Type | Non-Primitive Data Type |
| Definition | Stores actual values | Stores memory address of object |
| Storage | Stack memory | Heap memory (reference in stack) |
| Speed | Faster | Slower (requires object creation) |
| Modifiability | Immutable (except arrays) | Mutable |
| Can be null? | ❌ No | ✅ Yes |
| Examples | int, double, char, boolean | String, Array, Class, Interface |

Java divides the operators into the following groups:

* Arithmetic operators 🡪 +, -, \*, /, %, ++, --
* Assignment operators 🡪 =,+=,-=,/=,%=,>>=,<<=,^=,|=
* Comparison operators 🡪 ==, !=,>,<,<=,>=
* Logical operators 🡪&&, ||, !

Java Conditions and If Statements

Java has the following conditional statements:

* Use if to specify a block of code to be executed, if a specified condition is true
* Use else to specify a block of code to be executed, if the same condition is false
* Use else if to specify a new condition to test, if the first condition is false
* Use switch to specify many alternative blocks of code to be executed
* Break statement can be used to jump out of a loop.
* Continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

Example-

int time = 22; // if…else statement example

if (time < 10) {

System.out.println("Good morning.");

} else if (time < 18) {

System.out.println("Good day.");

} else {

System.out.println("Good evening.");

}

// Outputs "Good evening."

int day = 4; //switch statement example

switch (day) {

case 6:

System.out.println("Today is Saturday");

break;

case 7:

System.out.println("Today is Sunday");

break;

default:

System.out.println("Looking forward to the Weekend");

}

// Outputs "Looking forward to the Weekend"

LOOPS 🡪

* 1. While loop :  Loops through a block of code as long as a specified condition is true.

Ex:

int i = 0;

while (i < 5) {

System.out.println(i);

i++;

}

* 1. Do/while loop: This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Ex:

int i = 0;

do {

System.out.println(i);

i++;

}

while (i < 5);

//OUTPUT: 0 1 2 3 4

* 1. For loop: When you know exactly how many times you want to loop through a block of code, use the for loop.

Ex:

for (int i = 0; i <= 10; i = i + 2) {

System.out.println(i);

}

//OUTPUT: 0 2 4 6 8 10

**Arrays in Java**

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

Uses square brackets to declare.

Array is reference data type. Stores ordered collection of elements of same data type.

Ex: int[] myNum = {10, 20, 30, 40};

Loop Through an Array:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (int i = 0; i < cars.length; i++) {

System.out.println(cars[i]);

}

Multidimensional Arrays : A multidimensional array is an array of arrays. They are useful when you want to store data as a tabular form (rows n column).

Ex: int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };

**Methods in Java**

A **method** in Java is a block of code that performs a specific task and can be called multiple times.

Types of Methods in Java

|  |  |
| --- | --- |
| Type | Description |
| 1. Predefined (Built-in) Methods | Methods provided by Java libraries (e.g., Math.sqrt(), System.out.println()). |
| 2. User-defined Methods | Methods created by the user to perform a specific task. |
| 3. Static Methods | Declared using static keyword; called without creating an object (ClassName.methodName()). |
| 4. Instance Methods | Belong to an instance of a class; require an object to be called. |
| 5. Abstract Methods | Declared in an abstract class/interface without implementation; must be overridden. |

**What is a String?**

A string is a sequence of characters.In Java, String is an immutable (unchangeable) class in the java.lang package.

**Ways to Create a String**

1.Using string literal:

String s1 = "Hello";

2.Using new keyword:

String s2 = new String("Hello");

A String is immutable, meaning once created, it cannot be changed.

**Reasons for String Immutability 🡪**

1️. Memory Optimization – Strings are stored in the String Pool. If two variables hold the same value, they point to the same object.  
2️. Security – Prevents accidental modifications, ensuring security in passwords, database connections, etc.  
3️. Thread-Safety – Multiple threads can safely share a String without synchronization issues.  
4️. Efficient Hashing – Used in HashMaps and caching because hashcodes don’t change.

**String Methods in Java**

|  |  |  |
| --- | --- | --- |
| Method | Description | Example |
| length() | Returns string length | "hello".length() → 5 |
| charAt(index) | Returns character at index | "Java".charAt(1) → 'a' |
| toUpperCase() | Converts to uppercase | "java".toUpperCase() → "JAVA" |
| toLowerCase() | Converts to lowercase | "JAVA".toLowerCase() → "java" |
| substring(start, end) | Extracts substring | "Hello".substring(1,4) → "ell" |
| concat(str) | Concatenates strings | "Hello".concat(" World") → "Hello World" |
| equals(str) | Compares content (case-sensitive) | "java".equals("Java") → false |
| equalsIgnoreCase(str) | Compares content (case-insensitive) | "java".equalsIgnoreCase("Java") → true |
| trim() | Removes leading & trailing spaces | " Java ".trim() → "Java" |
| replace(old, new) | Replaces characters | "abc".replace('a', 'x') → "xbc" |
| split(delimiter) | Splits string into an array | "a,b,c".split(",") → ["a", "b", "c"] |
| isEmpty() | Checks if string is empty | "".isEmpty() → true |
| contains(str) | Checks if string contains substring | "Hello".contains("lo") → true |
| startsWith(prefix) | Checks if string starts with given prefix | "Java".startsWith("Ja") → true |
| endsWith(suffix) | Checks if string ends with given suffix | "Java".endsWith("va") → true |
| indexOf(str) | Returns index of first occurrence of substring | "Hello".indexOf("l") → 2 |
| lastIndexOf(str) | Returns index of last occurrence of substring | "Hello".lastIndexOf("l") → 3 |
| toCharArray() | Converts string to character array | "abc".toCharArray() → ['a', 'b', 'c'] |

**Difference between StringBuilder and String Buffer**

Both StringBuilder and StringBuffer are mutable versions of String, meaning they allow modifications without creating new objects.

|  |  |  |
| --- | --- | --- |
| Feature | StringBuffer | StringBuilder |
| Thread-Safety | **✅ Yes (synchronized)** | **❌ No (not synchronized)** |
| Performance | **Slower due to thread safety** | **Faster as no synchronization overhead** |
| Best Used When | **Multi-threaded environments** | **Single-threaded applications** |

**Difference Between == and .equals() in Java**

|  |  |  |
| --- | --- | --- |
| Feature | == (Equality Operator) | .equals() (Method) |
| Comparison Type | Compares memory addresses (references) | Compares content (values) |
| Used For | Primitive data types & object references | Object content comparison |
| Default Behavior in Objects | Checks if two references point to the same object | Checks if two objects have equal values (unless overridden) |
| Overridable? | ❌ No | ✅ Yes (Can be overridden in custom classes) |

REGULAR EXPRESSIONS IN JAVA

* A Regular Expression (RegEx) is a sequence of characters that defines a search pattern.
* Java provides the java.util.regex package for pattern matching using regular expressions.
* Main Classes in java.util.regex Package🡪

1. **Pattern** – Defines a pattern to be used in regex operations.
2. **Matcher** – Performs match operations on a given input string.
3. **PatternSyntaxException** – Handles syntax errors in regex patterns.

* Basic regex Syntax🡪

|  |  |  |  |
| --- | --- | --- | --- |
| Pattern | Description | Example | Matches |
| . | Any character except newline | "a.c" | abc, adc |
| \d | Any digit (0-9) | "\d" | 3, 7 |
| \D | Any non-digit | "\D" | A, b |
| \w | Any word character (a-z, A-Z, 0-9, \_) | "\w" | a, Z, 5 |
| \W | Any non-word character | "\W" | #, @ |
| \s | Any whitespace (space, tab, newline) | "\s" | ' ', \n |
| \S | Any non-whitespace character | "\S" | a, 1 |
| ^ | Start of a string | "^Hello" | "Hello world" |
| $ | End of a string | "world$" | "Hello world" |
| \* | 0 or more occurrences | "ab\*" | a, ab, abb |
| + | 1 or more occurrences | "ab+" | ab, abb |
| ? | 0 or 1 occurrence | "ab?" | a, ab |
| {n} | Exactly n occurrences | "a{3}" | aaa |
| {n,} | At least n occurrences | "a{2,}" | aa, aaa |
| {n,m} | Between n and m occurrences | "a{2,4}" | aa, aaa, aaaa |
| ` | ` | OR condition | `"cat |
| () | Grouping | "(ab)+" | ab, abab |

* Simple example 🡪

import java.util.regex.\*;

public class RegexExample {

public static void main(String[] args) {

// Define a pattern

Pattern pattern = Pattern.compile("\\d{3}-\\d{2}-\\d{4}"); // Matches "123-45-6789"

// Create a matcher for input string

Matcher matcher = pattern.matcher("My SSN is 123-45-6789.");

// Check if it matches

if (matcher.find()) {

System.out.println("Valid pattern found: " + matcher.group());

} else {

System.out.println("No match found.");

}

}

}

* Common Methods of Pattern and Matcher🡪

|  |  |
| --- | --- |
| Method | Description |
| Pattern.compile(String regex) | Compiles the regex pattern |
| matcher(String input) | Creates a matcher for the input string |
| find() | Finds next occurrence |
| matches() | Checks if entire input matches the pattern |
| group() | Returns the matched substring |
| start() | Returns start index of the match |
| end() | Returns end index of the match |
| replaceAll(String replacement) | Replaces all matches |

**Writing Single & Multi-line Comments**

* **Single-line Comment:** //
* **Multi-line Comment:** /\* This is comment \*/

**Default Values in Java :** (Default values when variables are not initialized)

Example: class Test {

int a; // Default 0

boolean b; // Default false

String str; // Default null

}

|  |  |
| --- | --- |
| Data Type | Default Value |
| byte, short, int, long | 0 |
| float, double | 0.0 |
| char | '\u0000' (null character) |
| boolean | false |
| Object (Reference) | null |

\*\*\*\*\*\*\*\*\*\*\*\*

Java OOP(Object Oriented Programming) Concepts

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects. It allows for better code organization, reusability, and scalability.

Benefits of OOPs in Java 🡪

* Enhances code reusability using inheritance.
* Increases modularity with encapsulation.
* Provides security through data hiding.
* Makes debugging and maintenance easier.

The four fundamental principles of OOP in Java are 🡪

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction

1. **Class V/S Object in Java**

**Difference Between Class and Object in Java**

|  |  |  |
| --- | --- | --- |
| Feature | Class | Object |
| Definition | A blueprint or template for creating objects. | An instance of a class. |
| Memory Allocation | No memory is allocated until an object is created. | Takes up memory when instantiated. |
| Declaration | Declared using the class keyword. | Created using the new keyword. |
| Usage | Defines properties (variables) and behaviors (methods). | Represents a real-world entity with specific values. |
| Example Code | class Car { String model;  void drive() {} } | Car c = new Car();  c.model = "Tesla"; |
| Real-World Analogy | A blueprint for a car. | A specific car built from that blueprint. |

**2. Program Structure**

* Basic Java Program Structure: (package → import → class → {} follows the correct Java order.)

// Package declaration (optional)

package mypackage;

// Import statements

import java.util.\*;

// Class declaration

class MyClass {

// Variables

int num = 10;

// Constructor

MyClass() {

System.out.println("Constructor called!");

}

// Method

void display() {

System.out.println("Hello, Java!");

}

// Main method (entry point)

public static void main(String[] args) {

MyClass obj = new MyClass(); // Object creation

obj.display();

}

}

**3. Access Modifiers:** Define visibility of variables, methods, and classes.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Modifier | Scope | Accessible in Same Class | Accessible in Same Package | Accessible in Subclass (Different Package) | Accessible Outside Package |
| public | Anywhere | ✅ Yes | ✅ Yes | ✅ Yes | ✅ Yes |
| private | Same class only | ✅ Yes | ❌ No | ❌ No | ❌ No |
| protected | Same package & subclasses | ✅ Yes | ✅ Yes | ✅ Yes | ❌ No |
| default | Same package only | ✅ Yes | ✅ Yes | ❌ No | ❌ No |

Example:

class Example {

private int a = 10; // Only accessible in the same class

public int b = 20; // Accessible everywhere

protected int c = 30; // Accessible in subclass & package

int d = 40; // Default, accessible within the package

}

**4. Understanding & Implementing OOP Concepts in Java**

**1. Class & Object**

* **Class**: A blueprint/template for creating objects.
* **Object**: An instance of a class with state (fields/variables) and behavior (methods).
* **Example**:

class Car {

String brand;

void drive() {

System.out.println("Car is driving");

}

}

public class Main {

public static void main(String[] args) {

Car myCar = new Car(); // Object creation

myCar.brand = "Toyota";

myCar.drive();

}

}

**2. Encapsulation**

* Wrapping data (variables) and code (methods) into a single unit (class).
* Data is hidden using **private** access modifier.
* Getters & setters are used for controlled access.
* Real-World Example - The ATM machine encapsulates its internal logic. Users interact through a secured interface (PIN authentication). Internal details (e.g., bank database, transaction processing) are hidden.
* **Example**:

class BankAccount {

private double balance; // Private variable

public void deposit(double amount) { // Public method

if (amount > 0) balance += amount;

}

public double getBalance() { // Getter method

return balance;

}

}

**3. Inheritance**

* Allows one class (child) to inherit properties & methods of another class (parent).
* Achieved using the **extends** keyword.
* **Types**:
  + **Single:** One class inherits from another. (A → B)
  + **Multilevel** : A class inherits from another class, which itself is inherited.(A → B → C)
  + **Hierarchical** :Multiple classes inherit from a single parent class.(A → B, A → C)
  + **Multiple Inheritance (via Interfaces)** - Java doesn’t support multiple inheritance with classes but allows it with interfaces.
* Real-World Example: A Car is a Vehicle, meaning it inherits common properties. The parent class (Vehicle) defines common functionality, while the child class (Car) extends it.
* **Example**:

class Animal {

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

}

class Dog extends Animal {

void bark() { System.out.println("Barking..."); }

}

**4. Polymorphism**

* Polymorphism means **"many forms"** and allows methods to perform different tasks based on the object that calls them.
* **Method Overloading** (Compile-time Polymorphism): Multiple methods with the same name but different parameters.

class MathUtil {

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

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

}

* **Method Overriding** (Runtime Polymorphism): Child class provides a specific implementation of a method from the parent class.

class Animal {

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

}

class Dog extends Animal {

void sound() { System.out.println("Dog barks"); }

}

|  |  |  |
| --- | --- | --- |
| Feature | Method Overloading | Method Overriding |
| Definition | Same method name, different parameters | Same method name, same parameters, but different class |
| Where It Occurs | Same class | Parent-child relationship (inheritance) |
| Return Type | Can be different | Must be the same or covariant |
| Access Modifier | Can be different | Cannot have a more restrictive access modifier |
| Binding Type | Compile-time (Static binding) | Runtime (Dynamic binding) |
| Usage | Improves code readability and flexibility | Provides specific implementation in subclass |

|  |  |  |
| --- | --- | --- |
| Feature | Compile-Time Polymorphism | Runtime Polymorphism |
| Definition | Method binding occurs at compile time | Method binding occurs at runtime |
| Binding Type | Static Binding | Dynamic Binding |
| Achieved By | **Method Overloading** | **Method Overriding** |
| Method Resolution | Done by the compiler | Done by the JVM at runtime |
| Flexibility | Less flexible (fixed at compile time) | More flexible (dynamic behavior) |
| Performance | Faster (resolved at compile time) | Slower (resolved at runtime) |
| Inheritance Needed? | ❌ No | ✅ Yes |
| Example | Multiple methods with the same name but different parameters | A subclass provides a specific implementation of a method from the superclass |

|  |  |  |
| --- | --- | --- |
| Feature | Upcasting | Downcasting |
| Conversion Type | Subclass → Superclass | Superclass → Subclass |
| Explicit Casting | Not required | Required |
| Access to Methods | Only superclass methods | Both superclass and subclass methods |
| Risk of ClassCastException | No | Yes (if not checked properly) |
| Definition | Parent calss reference can hold child class object |  |
|  | Done when there is IS-A relationship |  |

**instanceof Operator in Java**

The instanceof operator in Java is used to **check whether an object is an instance of a specific class or subclass**. It helps prevent **ClassCastException** when performing downcasting.

✅ **Use instanceof when**:

* Checking object type before downcasting.
* Implementing polymorphism-based behavior.
* Working with collections containing mixed types.

🚫 **Avoid using instanceof when**:

* You can achieve the same behavior using method overriding.
* It leads to tight coupling and violates OOP principles.

**5. Abstraction**

* Hides implementation details & shows only necessary features.
* **Achieved using:**
  + **Abstract Class** (using abstract keyword, can have abstract & concrete methods).
  + **Interface** (100% abstract, only method declarations).
* **Example (Abstract Class):**

abstract class Vehicle {

abstract void start();

}

class Car extends Vehicle {

void start() { System.out.println("Car starts with a key"); }

}

* **Example (Interface):**

interface Animal {

void makeSound();

}

class Dog implements Animal {

public void makeSound() { System.out.println("Dog barks"); }

}

**6. Interface vs Abstract Class**

|  |  |  |
| --- | --- | --- |
| Feature | Abstract Class | Interface (implements methods in class) |
| Methods | Can have both abstract & concrete methods | Only abstract methods (Java 7) & default/static methods (Java 8+) |
| Fields | Can have instance variables | Only constants (final static variables) |
| Inheritance | Single inheritance (extends) | Multiple inheritance (implements) |

**Implement the Interface**

* An interface will be implemented by a class using the keyword "implements"
* This class should implement all the methods in the interface. Otherwise, that class becomes abstract
* Implementing class can have its own methods
* Implementing class can also extend only one super class or abstract class

**7. Constructor**

* Special method used to initialize objects.
* **Types**: Default, Parameterized, Copy Constructor.
* **Example**:

class Car {

String brand;

Car(String b) { brand = b; }

}

**8. this & super Keywords**

* **this**: Refers to the current object.
* Uses of this Keyword

|  |  |  |
| --- | --- | --- |
| Use Case | Description | Example |
| 1. Referring to instance variables | Used when parameter names and instance variables are the same | this.variable = variable; |
| 2. Calling another constructor (this()) | Calls another constructor within the same class | this(value); |
| 3. Calling a method from the same class | Calls a method inside the same class | this.methodName(); |
| 4. Returning the current object | Used in method chaining | return this; |
| 5. Passing current object as an argument | Passes the current object to another method | obj.display(this); |

* **super**: Refers to the parent class members. It helps in method overriding, accessing parent class constructors, and resolving name conflicts.

**Uses of super in Java**

|  |  |
| --- | --- |
| Feature | Purpose |
| Access Parent Class Methods | Calls a method from the parent class when overridden. |
| Access Parent Class Variables | Refers to the superclass variable when hidden by a subclass variable. |
| Call Parent Class Constructor | Calls a superclass constructor to initialize inherited properties. |

* **Example**:

class Parent {

String name = "Parent";

}

class Child extends Parent {

String name = "Child";

void show() {

System.out.println(this.name); // Child

System.out.println(super.name); // Parent

}

}

**9. Static Keyword**

* **static variable**:

✅ A static variable belongs to the class, not an individual object.  
✅ Shared among all objects of the class (single copy in memory).  
✅ Initialized only once when the class is loaded.

* **static method**:

✅ A **static method** belongs to the class, not an instance. Should have a method body.   
✅ It **cannot access non-static (instance) variables or methods** directly. A static method in an interface can be invoked only by using the interface name and not by using reference of the interface.  
✅ It **can be called without creating an object. Uses word “static”.**✅ **Used for utility functions** (e.g., Math.pow(), Collections.sort()).

**Example**:

class Counter {

static int count = 0;

Counter() { count++; }

}

**10.default method in java🡪**

* A default method can be declared only within an interface.
* A default method must be marked with the default keyword and should have a method body.
* A default method is not assumed to be static, final, or abstract.
* It may be used or overridden by a class that implements the interface.
* A default method is public by default.
* Code will not compile if it is marked as private or protected

**11. Final Keyword**

* **final variable**: Cannot be changed.
* **final method**: Cannot be overridden.
* **final class**: Cannot be inherited.
* **Example**: final class Car { }
* The final keyword in Java is used to restrict modification. It can be applied to variables, methods, and classes to prevent changes in their behavior.

|  |  |  |
| --- | --- | --- |
| Usage | Description | Example |
| 1. final Variable | Prevents value from being changed (constant) | final int MAX = 100; |
| 2. final Method | Prevents method from being overridden in a subclass | final void display() {} |
| 3. final Class | Prevents class from being extended (no subclass allowed) | final class Animal {} |

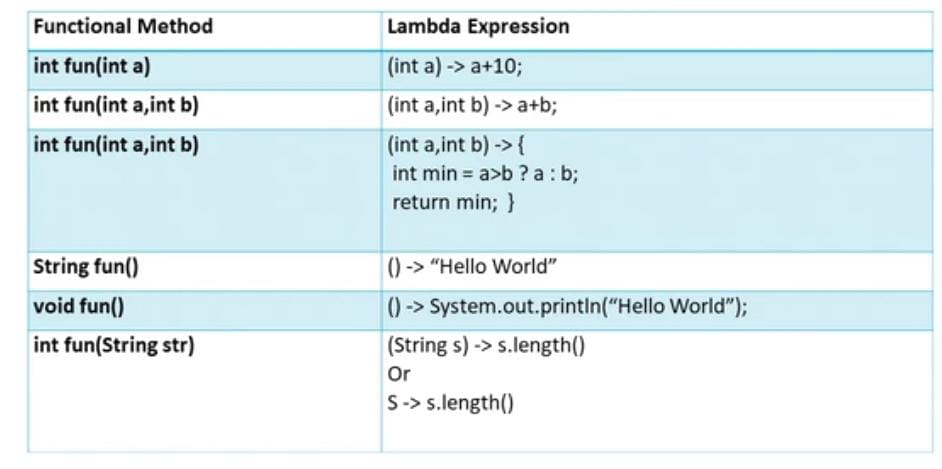
1. **Lambda Expression 🡪**

* Enables functional programming. Lambda expressions are used to pass Code as Data.
* Avoid boilerplate code - boilerplate code or boilerplate refers to sections of code that have to be included in many places with little or no alteration
* Enables support for parallel processing
* Lambda expression avoids unwanted anonymous inner class
* Syntax

(argument-list) -> (body)

argument-list: It can be empty or non-empty as well.

arrow-token: It is used to link arguments-list and body of expression.

body: It contains expressions and statements for lambda expression.

**EXCEPTION HANDLING IN JAVA🡪**

Exception handling in Java is a mechanism that allows developers to handle runtime errors, preventing abnormal program termination. It helps in maintaining the normal flow of the application even when an error occurs.

What is an Exception?

* An exception is an event that disrupts the normal flow of a program.
* It occurs at runtime due to issues like invalid input, file not found, division by zero, etc.
* **Checked vs Unchecked Exceptions in Java**

|  |  |  |
| --- | --- | --- |
| Feature | Checked Exceptions | Unchecked Exceptions |
| Definition | Exceptions checked at compile-time. | Exceptions that occur at runtime. |
| Extends | Exception (except RuntimeException). | RuntimeException and its subclasses. |
| Handling Required? | Yes, must be handled using try-catch or throws. | No, handling is optional. |
| Compiler Check? | Yes, the compiler forces handling. | No, the compiler does not check for handling. |
| Causes | External issues like file access, database errors, etc. | Programming errors like null reference, divide by zero, etc. |
| Examples | IOException, SQLException, FileNotFoundException | NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException |

* **Exceptions Mechanism in Java**
* **try** – Defines a block of code that may cause an exception.
* **catch** – Handles exceptions thrown inside the try block.
* **finally** – A block that always executes (used for cleanup).
* **throw** – Used to explicitly throw an exception.
* **throws** – Declares exceptions a method might throw.

1. What is a Custom Exception?

A custom exception (also called a user-defined exception) is a class that extends Exception or RuntimeException to create application-specific exceptions.

2. Why Use Custom Exceptions?

✔ To provide meaningful error messages specific to the application.  
✔ To improve code readability and debugging by differentiating standard exceptions from business logic errors.  
✔ To encapsulate business rules (e.g., age validation, invalid transactions).

3. Steps to Create a Custom Exception

Step 1: Define a Class That Extends Exception or RuntimeException

* Extend Exception (Checked Exception) → Must be handled using try-catch or throws.
* Extend RuntimeException (Unchecked Exception) → No mandatory handling required.

Step 2: Implement a Constructor with a Message

* Use a constructor that accepts a String message to provide details about the exception.

Step 3: Use throw to Raise the Exception

* Use throw keyword inside a method to trigger the exception when a condition is met.

Step 4: Handle It Using try-catch

* Use try-catch to catch and process the custom exception.

**Checked vs. Unchecked Custom Exceptions**

|  |  |  |
| --- | --- | --- |
| Feature | Checked Custom Exception (extends Exception) | Unchecked Custom Exception (extends RuntimeException) |
| Handling Required? | Yes (try-catch or throws required) | No handling required |
| Checked by Compiler? | Yes | No |
| Use Case | Business rules (e.g., age validation, file operations) | Programming errors (e.g., invalid input, negative values) |

**Syntax of a Custom Exception**

**Checked Custom Exception (Extends Exception)**

// Step 1: Define Custom Exception

class AgeException extends Exception {

public AgeException(String message) {

super(message);

}

}

// Step 2: Throw Custom Exception

public class CustomExceptionExample {

public static void checkAge(int age) throws AgeException {

if (age < 18) {

throw new AgeException("Age must be 18 or above.");

} else {

System.out.println("Access granted!");

}

}

public static void main(String[] args) {

try {

checkAge(16); // Will throw AgeException

} catch (AgeException e) {

System.out.println("Exception caught: " + e.getMessage());

}

}

}

✔ **Output:**

Exception caught: Age must be 18 or above.

**Unchecked Custom Exception (Extends RuntimeException)**

// Step 1: Define Custom RuntimeException

class InvalidAmountException extends RuntimeException {

public InvalidAmountException(String message) {

super(message);

}

}

// Step 2: Throw Custom Exception

public class UncheckedCustomException {

public static void withdraw(double amount) {

if (amount < 0) {

throw new InvalidAmountException("Amount cannot be negative!");

}

System.out.println("Withdrawal successful: $" + amount);

}

public static void main(String[] args) {

withdraw(-500); // Will throw InvalidAmountException

}

}

✔ **Output:**

Exception in thread "main" InvalidAmountException: Amount cannot be negative!

**COLLECTIONS,GENERICS AND STREAM API IN JAVA🡪**

The Collections Framework in Java provides a set of classes and interfaces to store and manipulate groups of objects efficiently.

It provides efficient ways to handle data structures like **lists, sets, maps, and queues**.

The Java Collections Framework is a hierarchy of interfaces and classes found in the java.util package. It consists of:

1. Interfaces – Define the abstract data structures.
2. Concrete Classes – Implement various collection types.
3. Algorithms – Provide static methods for sorting, searching, etc.

*Types of Collections JAVA*

1.List (Ordered collection, allows duplicates,provide indexing access to elements) 🡪

|  |  |
| --- | --- |
| Implementation Class | Features |
| ArrayList | Dynamic array, fast random access (O(1)), slow insertions & deletions |
| LinkedList | Doubly linked list, fast insertions & deletions (O(1)), slow random access (O(n)) |
| Vector | Synchronized version of ArrayList (thread-safe but slow) |
| Stack | Last-In-First-Out (LIFO) structure |

Ex:

import java.util.\*;

public class ListExample {

public static void main(String[] args) {

List<String> list = new ArrayList<>();

list.add("Apple");

list.add("Banana");

list.add("Cherry");

System.out.println(list.get(1));} // Output: Banana

}

2.Set (Unordered collection, no duplicates) 🡪

|  |  |
| --- | --- |
| Implementation Class | Features |
| HashSet | Unordered, fast lookup (O(1)) |
| LinkedHashSet | Ordered version of HashSet (insertion order maintained) |
| TreeSet | Sorted (natural order), backed by a **Red-Black tree** (O(log n)) |

Ex:

import java.util.\*;

public class SetExample {

public static void main(String[] args) {

Set<Integer> set = new HashSet<>();

set.add(10);

set.add(20);

set.add(10); // Duplicate, won't be added

System.out.println(set); }// Output: [10, 20]

}

3.Queue (FIFO structure, used for scheduling)🡪

|  |  |
| --- | --- |
| Implementation Class | Features |
| PriorityQueue | Elements are sorted based on priority |
| LinkedList | Can be used as a queue |

Ex:

import java.util.\*;

public class QueueExample {

public static void main(String[] args) {

Queue<Integer> queue = new LinkedList<>();

queue.add(10);

queue.add(20);

queue.add(30);

System.out.println(queue.poll()); // Output: 10 (FIFO)

}

}

4. Map (Key-Value pairs,unique keys but value can be duplicated)🡪

|  |  |
| --- | --- |
| Implementation Class | Features |
| HashMap | Unordered, fast lookup (O(1)), allows null keys |
| LinkedHashMap | Maintains insertion order |
| TreeMap | Sorted (natural order), uses Red-Black tree (O(log n)) |
| Hashtable | Thread-safe version of HashMap, does not allow null keys |

Ex:

import java.util.\*;

public class MapExample {

public static void main(String[] args) {

Map<String, Integer> map = new HashMap<>();

map.put("Apple", 100);

map.put("Banana", 50);

map.put("Cherry", 75);

System.out.println(map.get("Apple")); // Output: 100

}

}

**Performance Comparisons**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Feature | ArrayList | LinkedList | HashSet | TreeSet | HashMap | TreeMap |
| Insertion | Fast (O(1)) | Slow (O(n)) | Fast (O(1)) | Slow (O(log n)) | Fast (O(1)) | Slow (O(log n)) |
| Deletion | Slow (O(n)) | Fast (O(1)) | Fast (O(1)) | Slow (O(log n)) | Fast (O(1)) | Slow (O(log n)) |
| Search | Fast (O(1)) | Slow (O(n)) | Fast (O(1)) | Slow (O(log n)) | Fast (O(1)) | Slow (O(log n)) |
| Order | Maintained | Maintained | Unordered | Sorted | Unordered | Sorted |

**Difference Between Collection and Collections in Java**

|  |  |  |
| --- | --- | --- |
| Feature | Collection | Collections |
| Type | Interface | Utility Class |
| Package | java.util | java.util |
| Purpose | Represents a group of objects | Provides utility methods for collections |
| Usage | Used to create different collection types like List, Set, Queue, etc. | Used for sorting, searching, synchronization, and modifying collections |
| Example | List<Integer> list = new ArrayList<>(); | Collections.sort(list); |

🡪Generics help in type safety and eliminate the need for typecasting.

🡪The Stream API is used for functional-style programming and processing collections efficiently.

Example: Filtering and Sorting using Streams

✅ Key Stream Operations:

*1.Intermediate Operations (return stream)*

filter(), map(), sorted(), distinct(), limit(), skip()

*2.Terminal Operations (return value)*

collect(), forEach(), count(), reduce(), allMatch(), anyMatch(), noneMatch()

**STREAM API**

Prequisite 🡪 Collection framework, Lambda Expression, Predicate, Consumer, Supplier

**1. Predicate<T>**

* Represents a boolean-valued function of one argument.
* Typically used for filtering, conditional checks, and validation.
* Contains methods like test(), and(), or(), negate()

**2. Consumer<T>**

* Represents an operation that accepts a single input argument and returns no result.
* Typically used for performing actions (like printing, modifying objects, logging).
* Contains accept() method.

**3. Supplier<T>**

* Represents a function that provides a result (produces a value) without taking any input.
* Typically used for factory methods, lazy evaluation, and fetching data.
* Contains get() method.

Java Streams, introduced in **Java 1.8 version**, provide a functional approach to processing sequences of data. Perform on bulk operations.

They help in performing operations such as filtering, mapping, and reducing code length.

They are related to Collection Frameworks(Group of objects) and are different from IO Streams (sequence of data).

**Stream vs Collection**

|  |  |  |
| --- | --- | --- |
| Feature | Stream | Collection |
| Storage | Doesn't store elements | Stores elements |
| Computation | Lazy | Eager |
| Modification | Cannot modify original elements | Can modify elements |
| Iteration | Internal | External |

**Stream API** and **I/O Streams** in Java:

|  |  |  |
| --- | --- | --- |
| Feature | Stream API (java.util.stream) | I/O Streams (java.io & java.nio) |
| Purpose | Used for processing collections of data (functional programming). | Used for reading/writing data from files, network, or console. |
| Type | Works on **collections, arrays, and sequences of data**. | Works on **files, sockets, and byte/character streams**. |
| Operations | Supports **filter, map, reduce, sorting, and other functional transformations**. | Supports **reading/writing data in bytes or characters**. |
| Parallelism | Supports **parallel streams** for better performance. | No built-in parallel support, needs multi-threading. |
| Memory Usage | Operates in a **lazy and optimized** manner (only processes when needed). | Consumes memory based on the size of the file or data stream. |
| Data Source | Works with **collections (List, Set, Map, etc.)**. | Works with **files, network sockets, input/output devices**. |
| Modifies Original Data? | **No**, it creates a pipeline of transformations. | **Yes**, it reads/writes directly to the source. |
| Example Usage | list.stream().filter(...).map(...).collect(...) | FileInputStream, BufferedReader, DataOutputStream |

Q. Create a list and filter all even numbers from List.

WAY 1: List<Integer> list1= List.of(2,4,50,21); //this is immutable list.

WAY 2: List<Integer> list2= new ArrayList<>();

list2.add(12);

list2.add(20);

WAY 3: List<Integer> list3= Arrays.asList(23,24,65,86);

Using Streams🡪 Stream <Integer> stream=list1.stream();

List<Integer> newList= stream.filter(i->i%2==0).collect(Collectors.toList);

Can also be written as🡪

List<Integer> newList=list1. stream.filter(i->i%2==0).collect(Collectors.toList);

**Creating Streams**

Streams can be created from various sources:

**A. From Collections**

*List<String> list = Arrays.asList("Apple", "Banana", "Cherry");*

*Stream<String> stream = list.stream();*

**B. From Arrays**

*String[] array = {"A", "B", "C"};*

*Stream<String> stream = Arrays.stream(array);*

**C. Using Stream.of()**

*Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);*

**D. Generating Infinite Streams**

*Stream<Integer> infiniteStream = Stream.iterate(1, n -> n + 2);*

Methods to Create a Stream Object🡪

|  |  |
| --- | --- |
| Source | Method |
| Collections (List, Set, Map) | collection.stream() |
| Arrays | Arrays.stream(array), Stream.of(array) |
| Infinite Streams | Stream.generate(), Stream.iterate() |
| I/O Streams (BufferedReader) | bufferedReader.lines() |
| Splitting Strings | Pattern.compile().splitAsStream() |
| Manually Built Streams | Stream.builder() |

Call stream method on Object and not directly on any source.

Stream operations are categorized into:

1. **Intermediate Operations** (return a Stream)
2. **Terminal Operations** (return a result)

**A. Intermediate Operations**

These operations **do not process** the elements until a terminal operation is invoked.

|  |  |
| --- | --- |
| Operation | Description |
| filter(Predicate<T>) | Filters elements based on a condition. |
| map(Function<T, R>) | Transforms elements to another form. |
| sorted() / sorted(Comparator<T>) | Sorts elements in natural or custom order. |
| distinct() | Removes duplicates. |
| limit(n) / skip(n) | Limits the number of elements or skips first 'n' elements. |

**Examples:**

1.List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

List<String> filteredNames = names.stream()

.filter(name -> name.startsWith("A"))

.collect(Collectors.toList());

2.List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> squaredNumbers = numbers.stream()

.map(n -> n \* n)

.collect(Collectors.toList());

**B. Terminal Operations**

These operations **trigger** stream processing and return a result.

|  |  |
| --- | --- |
| Operation | Description |
| collect(Collectors.toList()) | Converts stream to a List. |
| forEach(Consumer<T>) | Iterates over each element. |
| reduce(BinaryOperator<T>) | Reduces elements to a single value. |
| count() | Counts the number of elements. |
| anyMatch(), allMatch(), noneMatch() | Checks conditions on elements. |
| findFirst(), findAny() | Retrieves an element. |

**Examples:**

1.List<String> list = Arrays.asList("one", "two", "three");

list.stream().forEach(System.out::println);

2.List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sum = numbers.stream().reduce(0, Integer::sum);

**ANNOTATIONS IN JAVA 🡪**

Annotations in Java are a form of metadata that provide information about the code without directly affecting its execution.

* Metadata: Information about code.
* No Execution: Don't run code directly.
* @interface: How to define them.
* @Annotation: How to use them.
* Retention: SOURCE, CLASS, RUNTIME (how long they last).
* Target: Where they apply (class, method, etc.).
* Built-in: @Override, @Deprecated, @SuppressWarnings.
* Custom: You can make your own.
* Members: Annotations can hold values.
* Reflection: Access at runtime (if RUNTIME).
* Uses: Compiler hints, tools, runtime changes.

**JDBC 🡪**

* JDBC (Java Database Connectivity) is an API in Java that allows applications to interact with databases.
* It enables Java programs to perform database operations like **Create, Read, Update, and Delete (CRUD).**
* JDBC follows a **4-layer** architecture:

1. **JDBC API** – Provides interfaces for interacting with databases.
2. **JDBC Driver Manager** – Manages different database drivers.
3. **JDBC Driver** – Translates Java calls into database-specific calls.
4. **Database** – Stores and retrieves data.

* Steps to Connect to a Database using JDBC (LECEC)🡪

1. **Load the JDBC Driver**

Class.forName("com.mysql.cj.jdbc.Driver"); // Load MySQL Driver

1. **Establish a Connection**

Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/dbname", "user", "password");

1. **Create a Statement**

Statement stmt = con.createStatement();

1. **Execute SQL Query**

ResultSet rs = stmt.executeQuery("SELECT \* FROM users");

while (rs.next()) {System.out.println(rs.getString("name"));

}

1. **Close the Connection**

con.close();

* **Types of JDBC Drivers**

1. **JDBC-ODBC Bridge Driver** (Type 1) – Uses ODBC, slow and outdated.
2. **Native-API Driver** (Type 2) – Uses OS-specific client API.
3. **Network Protocol Driver** (Type 3) – Translates JDBC calls to database-specific calls over the network.
4. **Thin Driver** (Type 4) – Directly connects to the database; widely used.

*MySQL, Oracle, and PostgreSQL use Type 4 drivers.*

* **Important JDBC Interfaces**

1. Connection – Manages database connection.
2. Statement – Executes SQL queries.
3. PreparedStatement – Precompiled SQL query for better performance.
4. ResultSet – Stores retrieved data.

* **Common JDBC Exceptions & Solutions**

|  |  |  |
| --- | --- | --- |
| Exception | Reason | Solution |
| ClassNotFoundException | Driver not found | Ensure JDBC driver is added to the project |
| SQLException | Invalid query or connection issue | Check SQL syntax and database connection |
| NullPointerException | Connection object not initialized | Ensure Connection is properly established |

OOPS QUESTIONS

What are exceptions in Java? Explain the difference between checked and unchecked exceptions.

What is the purpose of the try-catch block? Give an example.

What is a thread in Java? How can you create and manage threads?

What are the differences between an Array and an Array List in Java?

What is the purpose of garbage collection in Java? How does it work?

What are wrapper classes in Java? Why are they needed?

A wrapper class is a class that converts primitive data types into objects. Java provides wrapper classes for all eight primitive types in the java.lang package.

|  |  |
| --- | --- |
| Primitive Type | Wrapper Class |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |

Why are Wrapper Classes Needed?

✅ **Object Manipulation** – Many Java APIs require objects instead of primitives (e.g., Collections).  
✅ **Autoboxing & Unboxing** – Automatically converts between primitives and wrapper objects.  
✅ **Utility Methods** – Wrapper classes provide useful methods (e.g., Integer.parseInt()).  
✅ **Synchronization** – Objects are required for synchronization in multithreading.  
✅ **Generics Support** – Generics work only with objects, not primitives.

What is autoboxing and unboxing in Java? Provide examples.

In Java, Autoboxing and Unboxing are features that automatically convert primitive types into their corresponding wrapper objects and vice versa.

1.Autoboxing (Primitive → Wrapper Object)

Automatic conversion of primitive data types into wrapper class objects.

🔹 Example:

public class AutoboxingExample {

public static void main(String[] args) {

int num = 10;

Integer obj = num; // Autoboxing (int → Integer)

System.out.println(obj); // Output: 10

}

}

*Here, int num is automatically converted into an Integer object.*

2.Unboxing (Wrapper Object → Primitive)

Automatic conversion of wrapper class objects into primitive types.

🔹 Example:

public class UnboxingExample {

public static void main(String[] args) {

Integer obj = 20;

int num = obj; // Unboxing (Integer → int)

System.out.println(num); // Output: 20

}

}

📌 *Here, Integer obj is automatically converted into an int.*

|  |  |  |
| --- | --- | --- |
| Feature | Autoboxing | Unboxing |
| Definition | Primitive → Wrapper object | Wrapper object → Primitive |
| Example | int → Integer | Integer → int |
| Use Case | Storing primitives in Collections, Generics | Retrieving values from Collections |
| Example Code | Integer obj = 10; | int num = obj; |

Explain the difference between HashMap and HashSet.

Both HashMap and HashSet are part of the Java Collections Framework and use hashing for efficient data storage and retrieval. However, they have different use cases.

🔹 Key Differences

|  |  |  |
| --- | --- | --- |
| Feature | HashMap<K, V> | HashSet<E> |
| Type | Stores key-value pairs | Stores unique elements |
| Duplicates Allowed? | ❌ No duplicate keys (values can be duplicate) | ❌ No duplicates at all |
| Order Maintained? | ❌ No ordering guarantee | ❌ No ordering guarantee |
| Underlying Implementation | Uses an array of linked lists (buckets) | Internally uses a HashMap with dummy values |
| Performance (O(1) for insert, delete, search) | ✅ Fast for key lookups | ✅ Fast for element searches |
| Use Case | When key-value pairs are needed (e.g., storing user data) | When only unique elements are required (e.g., storing unique IDs) |

What are annotations in Java? How are they useful?

What is the Java Collections Framework? List some commonly used classes.

The **Java Collections Framework (JCF)** is a set of classes and interfaces in java.util that provides a standard way to store, manipulate, and retrieve collections of objects efficiently.

🔹 Commonly Used Java Collections Classes

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Interface | Common Implementations | Usage |
| List (Ordered Collection) | List<E> | ArrayList, LinkedList, Vector, Stack | Stores elements in an ordered manner, allows duplicates |
| Set (Unique Elements) | Set<E> | HashSet, LinkedHashSet, TreeSet | Stores unique elements, unordered or sorted |
| Queue (FIFO Structure) | Queue<E> | PriorityQueue, LinkedList (Queue), ArrayDeque | Follows First-In-First-Out (FIFO) |
| Map (Key-Value Pairs) | Map<K, V> | HashMap, TreeMap, LinkedHashMap, Hashtable | Stores key-value pairs, keys must be unique |

package shopping.amazon; // Corrected package declaration

public class Employee {

private int id;

private String name;

private String department;

private double salary;

// Constructor - initializes id and name

public Employee(int id, String name) {

this.id = id;

this.name = name;

this.department = "Unknown"; // Default values

this.salary = 0.0;

}

// Getter and Setter Methods

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 String getDepartment() {

return department;

}

public void setDepartment(String department) {

this.department = department; }

public double getSalary() {

return salary;

}

public void setSalary(double salary) {

this.salary = salary;

}

}

**package shopping.amazon; // Corrected package declaration**

**public class TestEmployee {**

**public static void main(String[] args) {**

**Employee e1 = new Employee(1, "Aari"); // Creating Employee object**

**System.out.println("Employee ID: " + e1.getId());**

**System.out.println("Employee Name: " + e1.getName());**

**}**

**}**

**OOPS BASED QUESTIONS**

What is Java? Explain its key features.  
What are the different types of Java applications?  
Explain the difference between JDK, JRE, and JVM.  
What is the difference between a class and an object in Java?  
What are the basic OOP principles in Java? Explain with examples.  
What is the purpose of the main() method in Java?  
What are constructors in Java? How are they different from methods?  
What are Java access modifiers? Explain with examples.  
What is the difference between == and .equals() in Java?  
What is method overloading and method overriding? Give examples.  
What is the difference between primitive and non-primitive data types in Java?  
Explain the use of the this keyword in Java.  
What are static methods and variables in Java? How are they used?  
What is the significance of the final keyword in Java?  
What are Java packages? Why are they important?  
What is an interface in Java? How is it different from an abstract class?  
What is the difference between compile-time and runtime polymorphism in Java?  
What are exceptions in Java? Explain the difference between checked and unchecked exceptions.  
What is the purpose of the try-catch block? Give an example.  
What is a thread in Java? How can you create and manage threads?  
What are the differences between an Array and an Array List in Java?  
What is a String in Java? How are Strings immutable?  
What is the purpose of garbage collection in Java? How does it work?  
What are wrapper classes in Java? Why are they needed?  
What is autoboxing and unboxing in Java? Provide examples.  
What is the purpose of the super keyword in Java?  
Explain the difference between HashMap and HashSet.  
What are annotations in Java? How are they useful?  
What is the Java Collections Framework? List some commonly used classes.  
What is the difference between StringBuilder and String Buffer?

OOPS  
-------------------------------------------------------------  
What is Object-Oriented Programming (OOP), and how is it different from procedural programming?  
What are the four main principles of OOP in Java? Explain each with examples.  
What is a class in Java? How is it different from an object?  
What is inheritance in Java? What are the different types of inheritance supported in Java?  
What is polymorphism in Java? Explain its types with examples.  
What is encapsulation in Java? How does it promote data hiding?  
What is abstraction in Java? How is it achieved using abstract classes and interfaces?  
What are abstract classes in Java? How are they different from concrete classes?  
What is an interface in Java? How is it different from an abstract class?  
Explain the concept of multiple inheritance in Java. How is it achieved using interfaces?  
What is the role of the super keyword in Java? Provide an example.  
What is the this keyword in Java? How is it used to resolve ambiguity?  
What are constructors in Java? How are they different from methods?  
What is a default constructor? How is it different from a parameterized constructor?  
What is method overloading in Java? How is it different from method overriding?  
What are the rules for method overriding in Java?  
What is the purpose of the final keyword in Java? How does it affect classes, methods, and variables?  
What is dynamic method dispatch in Java? How does it relate to runtime polymorphism?  
What is an inner class in Java? What are its types?  
What are getters and setters in Java? Why are they important in OOP?  
Explain the difference between association, aggregation, and composition in Java.  
What is the difference between a shallow copy and a deep copy of an object in Java?  
What is the role of the Object class in Java? What are its common methods?  
What is the difference between instanceof and getClass() in Java?  
What is a static method in Java? Can it be overridden? Why or why not?  
Explain the difference between a static block and an instance block in Java.  
What is a singleton class in Java? How is it implemented?  
What is the difference between IS-A and HAS-A relationships in Java?  
What is the difference between composition and inheritance in Java? When should you prefer one over the other?  
What are anonymous classes in Java? Provide an example.  
What is object cloning in Java? How is it achieved using the Cloneable interface?  
What is the difference between abstraction and encapsulation? Can they be used together?  
What are access modifiers in Java? How do they affect the visibility of classes and members?  
What is a sealed class in Java? How does it enforce inheritance rules?  
What is a functional interface in Java? How does it relate to lambda expressions?  
What are the advantages and disadvantages of using inheritance in Java?  
How does Java handle object serialization and deserialization? What is the role of the Serializable interface?  
What is a blueprint design pattern in the context of OOP? How is it implemented in Java?  
What is covariance and contravariance in method overriding in Java?  
What is a dependency injection in OOP? How does it help in decoupling code?

 Getter and setter how to write and how it works

How do we create our methods