### Java: Overview

**Java** is a high-level, object-oriented programming language, designed to be platform-independent, robust, secure, and portable. It was developed by James Gosling at **Sun Microsystems** in 1995 (now owned by Oracle).

### Features of Java:

1. **Object-Oriented**: Everything in Java is an object, which means it follows the principles of OOP (Encapsulation, Inheritance, Polymorphism, Abstraction).
2. **Platform Independence**: Java programs are compiled into bytecode which can run on any system that has a Java Virtual Machine (JVM).
3. **Simple**: Java is designed to be easy to learn and use, with a syntax similar to C++ but eliminating complexities like pointers.
4. **Secure**: Java provides a secure environment with features like bytecode verification, runtime security checks, and the sandboxing of applications.
5. **Multithreading**: Java has built-in support for multithreading, which allows concurrent execution of two or more threads.
6. **Automatic Garbage Collection**: Java automatically manages memory by removing unused objects from memory to prevent memory leaks.
7. **Rich API**: Java offers a large set of built-in libraries for everything from networking to graphical user interfaces.

### Benefits of Java:

* **Write Once, Run Anywhere (WORA)**: Java code, once compiled into bytecode, can run on any platform with a JVM.
* **Distributed Computing**: Java is used in distributed applications like web servers, and its RMI (Remote Method Invocation) makes remote communication easy.
* **Enterprise Applications**: Java provides robust frameworks for large-scale applications (like Spring, Hibernate).

### Java SE,EE,LTS:

* **Java 17** is the latest Long-Term Support (LTS) version, released in **September 2021**.
* LTS versions are supported for a long period (8 years), making them stable and reliable for enterprise applications.
* Java SE(standard edition) is suitable for general-purpose programming, including desktop applications, whereas Java EE(enterprise edition) is aimed at developing and running large-scale, distributed, multi-tiered, reliable, and secure network applications.

### Difference Between JDK, JRE, and JVM:

**JVM (Java Virtual Machine)**: The engine that runs Java bytecode. It provides the environment where Java programs execute, allowing Java to be platform-independent.

* + **JVM Responsibilities**: Loading bytecode, verifying bytecode, interpreting/compiling bytecode into machine code, garbage collection.

**JRE (Java Runtime Environment)**: A package that includes JVM and libraries required to run Java applications.

* + **JRE Includes**: JVM, Java libraries (like java.util, java.lang), and other components necessary for running Java applications.

**JDK (Java Development Kit)**: A full-featured software development kit to develop Java applications. It includes JRE, compilers (like javac), debuggers, and other tools.

* + **JDK Includes**: JRE, compilers, development tools (e.g., javac, java).

### Working of Java Code:

1. **Write**: You write the Java source code using a text editor/IDE (e.g., HelloWorld.java).
2. **Compile**: The Java compiler (javac) translates the Java code into **bytecode** (HelloWorld.class). Bytecode is platform-independent.
3. **Run**: The Java bytecode is executed by the **JVM**. The JVM interprets or compiles the bytecode into machine-specific instructions.
   * When you run java HelloWorld, the JVM loads the bytecode and executes it.

**Compilation Process**:

* **Source Code (**.java**) → Bytecode (**.class**) → JVM → Machine Code**

### WORA (Write Once, Run Anywhere):

* **WORA** is a key feature of Java, meaning Java applications, once written and compiled into bytecode, can run on any platform (Windows, Linux, Mac) that has a JVM installed.
* The bytecode generated by Java’s compiler is not tied to any specific operating system or architecture, making Java applications highly portable.

**Java Applets-** Java applets are small applications written in Java that run within a web browser(so client-side technology.) .Applets were used to create dynamic and interactive content on web pages, such as games, animations, and data visualizations.   
**Note** - Applets typically run in a sandbox, which restricts access to local system resources for security reasons.

**Sandboxing** - Sandboxing is a security mechanism in Java that **restricts the actions of untrusted code**, typically code running from external sources like **applets** or **downloaded classes**.

It runs the code in a **controlled environment** (the "sandbox") to prevent it from accessing sensitive system resources (like file system, network, or OS commands) without permission.

### ****Garbage Collection in Java**** Garbage Collection (GC) in Java is the process of automatically freeing memory by destroying objects that are no longer reachable from any references.

Java uses a concept of **automatic memory management** to reduce memory leaks and optimize resource use.

#### ✅ Key Features:

No need for manual memory deallocation (free()).

JVM handles it using algorithms like **Mark and Sweep**, **Generational GC**, **G1 GC**, etc.

### Interview Questions:

1. **What is the difference between JDK, JRE, and JVM?**
2. **What is WORA in Java?**
3. **What is the role of JVM in Java?**
4. **What does 'platform independence' mean in Java?**
5. **Explain the Java compilation process.**
6. **Why is Java considered secure?**
7. **What is garbage collection in Java?**
8. **What are the major features of Java 17 (or the latest Java version)?**

These questions help to assess the fundamental understanding of Java and its environment, as well as awareness of recent Java versions.

### ****1. JShell (Java Shell)****

* **What it is**: JShell is an interactive tool introduced in Java 9 that allows you to test Java code snippets without writing a full program or compiling files. It's a REPL (Read-Eval-Print Loop) tool for Java.
* **Advantages**:
  + Quickly test code snippets.
  + Great for learning and prototyping.
  + No need to create .java files.

**How to use**:

$ jshell

jshell> int a = 10;

a ==> 10

jshell> System.out.println(a + 5);

15

**Interview Question**: What is JShell, and how is it different from compiling Java programs traditionally?

### ****2. Keywords****

* **What they are**: Reserved words in Java with predefined meanings that cannot be used as identifiers (variable or method names). Examples include class, static, final, int, if, etc.
* **Rules**:
  + Keywords are case-sensitive.
  + Cannot be redefined by the programmer.

**Example**:

int number = 5; // 'int' is a keyword

**Interview Question**: Why can’t keywords be used as identifiers in Java?

### ****3. Variables****

* **What they are**: Variables are containers for storing data during the program's execution.

#### Types of Variables:

1. **Local**: Declared inside a method/block and accessible only within it.
2. **Instance**: Declared in a class but outside methods; unique to each object.
3. **Static**: Declared in a class using static; shared across all objects.

**Example**:

class Example {

static int staticVar = 10; // Static variable

int instanceVar = 20; // Instance variable

void display() {

int localVar = 30; // Local variable

System.out.println(localVar);

}

}

* **Cannot declare a variable twice**:

int x = 10;

int x = 20; // Error: Duplicate declaration

**Interview Question**: What is the difference between instance and static variables?

### ****4. Expressions and Statements****

* **Expression**: A piece of code that produces a value (e.g., x + y, a > b).
* **Statement**: A complete instruction (e.g., int x = 5;).

**Example**:

int result = 10 + 20; // '10 + 20' is an expression, the whole line is a statement.

**Interview Question**: What is the difference between an expression and a statement?

### ****Primitive Data Types****

### ****Formula to Calculate Range****

For any **signed** data type (like byte, short, int, long):

**Range=−2(n−1) to 2(n−1)−1**

Where n = number of bits used to store the value.

For **unsigned** values (Java only supports signed primitives for integer types by default, except char):

**Range=0 to 2n−1**

#### Types of Primitive Data Types:

**Numeric Types**:

* + byte: 1 byte, range: -128 to 127.
  + short: 2 bytes, range: -32,768 to 32,767.
  + int: 4 bytes, range: -2^31 to 2^31-1.
  + long: 8 bytes, range: -2^63 to 2^63-1. Use L suffix for long literals.
  + float: 4 bytes, single-precision, use F suffix.
  + double: 8 bytes, double-precision (default for decimals).

**Character Type**:

* + char: 2 bytes, stores a single character in Unicode.  
    Example: char ch = 'A';

**Boolean Type**:

* + boolean: **1 bit**, holds true or false. But note that a single bit could theoretically store a boolean, **the smallest addressable unit of memory is a byte (8 bits)**. Therefore, a boolean usually occupies at least one byte in memory.

### ****6. Casting in Java****

* **What it is**: Converting one data type into another.
* **Types**:
  1. **Implicit (Widening)**: Smaller type to larger type. No data loss.  
     Example: int → double.
  2. **Explicit (Narrowing)**: Larger type to smaller type. Data may be lost.  
     Example: double → int.

**Example**:

int x = (int) 10.99; // Explicit casting

double y = 20; // Implicit casting

**Interview Question**: When would you use explicit casting in Java?

### ****7. Float and Double****

* **float**: 4 bytes, single-precision. Use f or F as a suffix.
* **double**: 8 bytes, double-precision, default type for decimals.

**Example**:

float pi = 3.14f; // 'f' suffix for float

double area = 3.14159;

**Interview Question**: Why is double preferred over float for calculations?

### ****9. String Data Type****

* **What it is**: A class used to store a sequence of characters. Strings are immutable (cannot be changed).
* **Example**:

String message = "Hello, World!";

**Interview Question**: What is the difference between String and StringBuilder?

### ****10. Operators and Operands****

* **Operators**: Perform operations on operands (e.g., +, -, \*, /, %, etc.).
* **Operands**: The data on which operators act.

**Example**:

int sum = 10 + 20; // '+' is the operator, 10 and 20 are operands.

### ****11. Abbreviating Operators****

* **What they are**: Shorthand for operations (e.g., +=, -=, ++, etc.).

**Example**:

int x = 5;

x += 2; // Same as x = x + 2

### ****12. Code Blocks and Scope****

* **Code Block**: Enclosed in {}.
* **Scope**: Defines where a variable is accessible.

**Example**:

{

int x = 10; // Scope limited to this block

}

System.out.println(x); // Error: x is out of scope

### ****13. Wrapper Classes****

**What they are**: Each primitive type in Java has a corresponding wrapper class that converts primitives into objects.  
Examples:

* + int → Integer
  + char → Character
  + boolean → Boolean

**Why use them**:

* + Collection frameworks (e.g., ArrayList) store objects, not primitives.
  + Provides utility methods for primitives.

**Example**:

Integer x = 10; // Autoboxing: primitive to object

int y = x.intValue(); // Unboxing: object to primitive

### ****Autoboxing and Unboxing in Java****

#### ✅ ****Definition:****

**Autoboxing**: Automatic conversion of a **primitive** data type into its corresponding **wrapper class** object.

**Unboxing**: Automatic conversion of a **wrapper class object** back into its corresponding **primitive** type

### 🔄 ****Example:****

public class Example {

public static void main(String[] args) {

// Autoboxing: int → Integer

int num = 10;

Integer obj = num; // Autoboxing

// Unboxing: Integer → int

Integer obj2 = 20;

int num2 = obj2; // Unboxing

System.out.println(obj + num2); // 30

}

}

**\* NullPointerException** can occur during unboxing if the wrapper is null:

Integer i = null;

int x = i; // ❌ NullPointerException

### ****14. If-Else Control Statement****

* **What it is**: Used for decision-making. Executes code based on conditions.

**Example**:

if (x > 0) {

System.out.println("Positive");

} else {

System.out.println("Non-Positive");

}

### ****15. Methods****

* **What they are**: A block of code that performs a specific task.
* **Method Overloading**: Method Overloading is a feature in Java where multiple methods can have the same name in the same class, but must differ by number, type, or order of parameters.

### ****Key Points:****

Overloading is resolved at **compile-time** (also called **compile-time polymorphism** or **static binding**.).

Method signature = method name + parameter list (type, number, order).

You **cannot overload** methods **only by return type**:

**Example**:

void greet() {

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

}

void greet(String name) { // Overloading

System.out.println("Hello, " + name);

}

**Interview Question**: What is method overloading, and how does it differ from overriding?

### ****1. Compile Time****

**What it is:**  
The phase when the **Java compiler** (javac) checks your code for **syntax errors**, type checking, and converts .java files into .class (bytecode) files.

#### ✅ Things checked at compile time:

Syntax errors

Data type mismatch

Missing semicolons or brackets

Variable not declared

Method overloading resolve

### ****Load Time**** The time when the ****JVM loads classes**** into memory ****before execution begins****.

The **ClassLoader** subsystem loads .class files.

Static blocks and static variables are initialized here.

### ****Initialization Time****

JVM initializes static variables, executes static blocks (so this how, code in static block executes prior to main method code), etc.

For objects, instance variables and constructors are initialized.

### ****Run Time**** The phase when the ****Java Virtual Machine (JVM)**** runs the compiled .class file. At this stage, your program actually ****executes****. ✅ Errors at runtime:

Divide by zero

Null pointer exception

Array index out of bounds

File not found

### ****General Interview Questions****:

1. **What are primitive data types in Java?**
2. **What is autoboxing and unboxing?**
3. **Why is Java case-sensitive?**
4. **Explain the difference between explicit and implicit casting.**
5. **What are wrapper classes, and when would you use them?**

## ****1. Switch Statement****

The switch statement allows multi-way branching based on the value of an expression.

### ****Syntax:****

switch (expression) {

case value1:

// Code to execute

break;

case value2:

// Code to execute

break;

default:

// Code if no case matches

}

### ****Example:****

int day = 3;

switch (day) {

case 1:

System.out.println("Monday");

break;

case 2:

System.out.println("Tuesday");

break;

case 3:

System.out.println("Wednesday");

break;

default:

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

}

## ****2. Valid and Invalid Switch Type Values****

### ****Valid Types:****

* byte, short, char, int
* String (Java 7+)
* enum (Java 5+)

### ****Invalid Types:****

* boolean (True/False)
* float, double, long
* Custom objects

#### ****Example of Invalid Switch:****

double value = 10.5;

switch(value) { // Error: double is not allowed

case 10.5:

System.out.println("Ten point five");

}

## ****3. Enhanced Switch Statement (Java 14+)****

The enhanced switch eliminates the need for break and supports lambda-like syntax.

### ****Example:****

String day = "MONDAY";

String message = switch (day) {

case "MONDAY" -> "Start of the week!";

case "FRIDAY" -> "Weekend is near!";

default -> "Regular day";

};

System.out.println(message);

## ****4. Loops in Java****

### ****4.1 For Loop****

Used when the number of iterations is known.

for (int i = 0; i < 5; i++) {

System.out.println("Iteration: " + i);

}

### ****4.2 While Loop****

Executes until a condition becomes false.

int i = 0;

while (i < 5) {

System.out.println("Count: " + i);

i++;

}

### ****4.3 Do-While Loop****

Executes at least once before checking the condition.

int i = 0;

do {

System.out.println("Value: " + i);

i++;

} while (i < 5);

## ****5. Break and Continue****

* **Break**: Exits the loop immediately.
* **Continue**: Skips the current iteration and moves to the next.

### ****Example:****

for (int i = 0; i < 5; i++) {

if (i == 3) {

continue; // Skips when i == 3

}

System.out.println(i);

}

## ****6. Class as a Template****

A **class** is a blueprint for creating objects.

### ****Example:****

class Car {

String model;

int year;

}

## ****7. Object Instance with new Keyword****

An **object** is created using the new keyword.

### ****Example:****

Car myCar = new Car(); // Creating an object of Car class

## ****8. Static vs Instance Fields (variable)****

### ****Instance Field:****

* Belongs to an object.
* Each object gets its own copy.

### ****Static Field:****

* Belongs to the class.
* Shared among all objects.

#### ****Example:****

class Example {

int instanceVar = 10; // Instance variable

static int staticVar = 20; // Static variable

}

## ****9. Static vs Instance Methods****

### ****Instance Method:****

* Requires an object to be called.

### ****Static Method:****

* Called without an object. Directly by ClassName.

#### ****Example:****

class Example {

static void staticMethod() {

System.out.println("Static method");

}

void instanceMethod() {

System.out.println("Instance method");

}

}

Example.staticMethod(); // Call without objectExample obj = new Example();

obj.instanceMethod(); // Call with object

## ****10.**** parseInt ****and**** parseDouble

Used to convert String to int or double.

### ****Example:****

int num = Integer.parseInt("100");

double price = Double.parseDouble("99.99");

## ****11. Reading Data from Console****

* System.console(): Reads input securely (only works in real terminals).
* Scanner: Common way to read input.

### ****Example Using**** System.console()

Console console = System.console();

String name = console.readLine("Enter name: ");

System.out.println("Hello, " + name);

## ****12. Exception Handling****

Handles runtime errors to prevent program crashes.

### ****Syntax:****

try {

// Risky code

} catch (ExceptionType e) {

// Handling code

} finally {

// Code that runs always

}

### ****Example:****

try {

int result = 10 / 0; // Division by zero

} catch (ArithmeticException e) {

System.out.println("Cannot divide by zero.");

} finally {

System.out.println("Execution completed.");

}

## ****13. Scanner Class****

Used to read input from the console or a file.

### ****Reading from Console:****

Scanner sc = new Scanner(System.in);

System.out.print("Enter your name: ");

String name = sc.nextLine();

System.out.println("Hello, " + name);

## ****14. Reading Input from a File using Scanner****

We can read data from a file using Scanner.

### ****Example:****

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class ReadFileExample {

public static void main(String[] args) {

try {

File file = new File("data.txt");

Scanner scanner = new Scanner(file);

while (scanner.hasNextLine()) {

System.out.println(scanner.nextLine());

}

scanner.close();

} catch (FileNotFoundException e) {

System.out.println("File not found.");

}

}

}

## ****Common Interview Questions****

1. **What is the difference between** break **and** continue**?**
2. **Why can’t a** switch **statement work with** float **or** boolean **values?**
3. **What is the difference between** static **and** instance **variables?**
4. **How do you handle exceptions in Java?**
5. **How can you read input from a file in Java?**
6. **What is the difference between** parseInt() **and** valueOf() **methods?**
7. **Why is** finally **used in exception handling?**
8. **What is the difference between** System.console() **and** Scanner **for reading input?**

### 1. ****What is OOPs?****

Object-Oriented Programming is a programming paradigm that uses "objects" and "classes".

It focuses on **code reusability**, **modularity**, **abstraction**, **encapsulation**, **inheritance**, and **polymorphism**.

**Benefits:**

Modular structure

Reusable code

Easier maintenance

Scalable

**Interview Q:**

What are the 4 pillars of OOP?

Why is OOP better than procedural programming?

### 2. ****Class as Blueprint, Objects, Instances****

**Class**: Blueprint for creating objects

**Object**: Instance of a class

**Instance**: Specific object created using new

class Car {

String color;

}

Car obj = new Car(); // obj is an instance

**Tip:** You can have multiple instances of a class, each with different state.

**Tricky Q:** Difference between class and object?

### 3. ****null keyword and Primitive Types****

null is a special literal used to indicate no reference.

Primitive types **can’t** be null, only reference types can.

String str = null; // OK

int num = null; // Compile-time error

#### ****Primitive (Non-Reference) Types****

These are the basic built-in types in Java. They store **actual values** in memory. Primitive types are usually allocated on the stack, a region of memory that is managed efficiently by the programming language.

#### ****Reference Types****

#### These are any types that **refer to objects** in memory.Reference types are usually allocated on the heap, a larger and more flexible memory area that is managed by the garbage collector

**Interview Q:** Why primitive types can't be null?

Because null represents the **absence of an object reference**, and primitive types are **not objects**. They always contain an actual value, not a reference.

### 4. ****Encapsulation, Getters & Setters, and**** this ****keyword****

## ****What is Encapsulation?****

**Encapsulation** is the OOP principle of **bundling data (variables) and methods (functions) that operate on the data into a single unit — a class — andhiding the internal details of how an object works and exposing only the necessary parts via public methods, typically called getters and setters.**

**Example: ATM Machine**

You just **press buttons** to withdraw cash.

You don’t know (or need to know) the internal logic (how it connects to bank servers, how money is debited).

The internal system is **encapsulated**, and you're provided **safe access methods**.

## Access Modifiers in Encapsulation

Encapsulation heavily relies on **access modifiers** in Java to **control visibility**.

| **Modifier** | **Visibility** | **Use in Encapsulation** |
| --- | --- | --- |
| private | Accessible only within the same class | Used to **hide data members** (variables) |
| public | Accessible from anywhere | Used for **getter/setter methods** |
| protected | Accessible in the same package and subclasses | Sometimes used if inheritance is involved |
| default (no modifier) | Accessible within the same package | Package-level encapsulation |

**Why Getters and Setters?**

Control how values are set or retrieved

Only validated data is accepted via setters (e.g., no negative age)

· **Q:** How is encapsulation different from abstraction?  
**A:**

Encapsulation hides **data**;

Abstraction hides **implementation complexity**.

· **Q:** Can you implement encapsulation without using private members?  
**A:** No — private (or at least restricted access) is necessary to hide internal state.

· **Q:** Is Java a fully encapsulated language?  
**A:** Not entirely. Java allows public and default members. But with good practices, classes can be fully encapsulated.

## ****What is**** this ****in Java?****

In Java, the this keyword is a **reference variable** that refers to the **current object** — the object whose method or constructor is being called.

## ****Main Roles of**** this ****Keyword****

| **Use Case** | **Description** | **Example** |
| --- | --- | --- |
| ✅ Refer to **current object’s fields** | Used to differentiate between instance variables and parameters with the same name | this.name = name; |
| ✅ Call another **constructor** in the same class | Used for constructor chaining | this(10, "Java"); |
| ✅ Pass the **current object as an argument** | Often used when passing the object to another method or constructor | someMethod(this); |
| ✅ Return the **current object** from a method | Useful for method chaining | return this; |

· **What is the use of** this **keyword in Java?**

To refer to the current object’s members (variables/methods).

· **Can** this() **be used in a method?**

❌ No, this() can **only be used in constructors**, and it must be the **first line**.

· **Can you return** this **in a method?**

✅ Yes, and it's often used in method chaining.

### 5. ****Constructors: Default, Parameterized, Overloading, Chaining****

## What is a Constructor in Java?

A **constructor** is a special method **used to initialize objects** in Java. It **has the same name as the class** and **no return type**, not even void.

It runs automatically when an object is created using the new keyword.

1. **Default Constructor**

A constructor **with no parameters**.

### ✅ Java provides it automatically ****if you don’t define any constructor****.

## 2. ****Parameterized Constructor****

A constructor that **takes arguments** to initialize values.

class Car {

String brand;

Car(String b) {

brand = b;

}

}

3. **Constructor Overloading**

Multiple constructors in the same class with **different parameter lists**.

Java picks the correct one based on arguments passed.

### ✅ Benefit: Flexibility to create objects in different ways.

4. **Constructor Chaining**

Using this() to **call one constructor from another** within the same class.

class Student {

String name;

int age;

Student() {

this("Unknown", 18); // Chaining to parameterized constructor

}

Student(String name, int age) {

this.name = name;

this.age = age;

}

}

### ⚠️ Rule:

this() **must be the first line** in the constructor.

## 🧬 Types of Constructor Chaining

| **Type** | **Example** |
| --- | --- |
| 🔁 **Within same class** | Use this() to call another constructor |
| 🧬 **Superclass constructor** | Use super() to call parent constructor (explained below) |

class Person {

Person() {

System.out.println("Person constructor");

}

}

class Student extends Person {

Student() {

super(); // calls Person()

System.out.println("Student constructor");

}

}

## ❗Important Notes & Rules

If **no constructor** is defined, Java adds a **default constructor**.

If **any constructor** is defined, Java **does not** add the default one.

Constructors **can’t be** static**,** final**, or** abstract.

❌ **Static**

Why?  
 static means the member belongs to the class, not any object.

But a constructor's whole purpose is to create an object.

❌ **Final**

Why?  
 final means cannot be overridden.

But constructors are not inherited and cannot be overridden anyway.

❌ **Abstract**

Why?  
 **abstract means:**

The method has no body

It must be implemented in a subclass

**But constructors:**

Always have a body

Are never inherited or overridden by a subclass

You **can’t call** this() **and** super() **together** — only one can be the first line.

## 💡 Interview Questions on Constructors

| **Question** | **Tip to Answer** |
| --- | --- |
| What is constructor overloading? | Multiple constructors with different signatures |
| Can you return a value from a constructor? | No, constructors have no return type |
| Can a constructor be private? | Yes — used in Singleton patterns - |
| Difference between constructor and method? | Constructor is called automatically; method is called explicitly |

**Important Note:** this() or super() must be the **first** statement in constructor.

**Interview Q:** Can we call super() and this() together? (No)

### 6. ****Reference vs Object vs Instance vs Class****

class A {}

A **ref**(ObjectName) = new A();

A is a **class**

new A() creates an **object**

Ref(ObjectName) is a **reference** pointing to an **instance** of A

### 7. ****Object Creation: With vs Without Reference****

new Car("red"); // No reference

Car c = new Car("blue"); // With reference

**Tip:** Without reference, you can't access object later.

### 8. ****Static Variables and Use Cases****

A static variable in Java is a **class-level** variable — it belongs to the **class** rather than to any individual instance (object) of that class.

class Car {

static int wheels = 4;

}

✔ Shared among all objects  
✔ Memory is allocated only **once** when the class is loaded

**Use Cases:**

Counting instances

Unique ID generation

#### Shared resources/constants - Used to define constants that are common to all instances of a class.

public static final double PI = 3.14159;

🧠 Note: final makes the variable constant (unchangeable).

Only one copy of a static variable is maintained, no matter how many instances are created — saves memory.

**Interview Q:** Why static methods can't access non-static variables directly?

Because static methods belong to the class, not to any specific object, while non-static variables belong to an instance (object).  
And you can’t access instance-specific data without creating or referring to an instance

### 9. ****Static vs Instance Methods****

| **Feature** | **Static Method** | **Instance Method** |
| --- | --- | --- |
| Belongs to | Class | Object (instance of a class) |
| Called using | Class name (or object, but not recommended) | Object reference only |
| Access to static vars | ✅ Yes | ✅ Yes |
| Access to instance vars | ❌ No (unless object is created) | ✅ Yes |
| When created | Loaded once when class is loaded | Created each time an object is created |
|  |  |  |

### Use Cases of static method:

Utility/helper methods (e.g., Math.pow())

Main method (public static void main)

Factory methods (static Foo create())

❓ Can instance methods call static methods?

✅ Yes, because instance methods have access to both static and non-static members.

❓ Why is main() method static?

So that JVM can call it **without creating an object**.

❓ Can you override static methods?

❌ No, they are hidden (method hiding), not overridden.

❓ Can this be used in a static method?

❌ No, because this refers to the current object, and static methods don’t belong to objects.

### ****Inheritance with Real-Life Example****

**Inheritance** is an **OOP concept** where one class (child) **inherits** fields and methods from another class (parent).

### 🔹 Inheritance allows you to ****reuse code****, ****extend functionality****, and achieve ****hierarchical relationships****.

| **Inheritance Type** | **Supported in Java?** | **Description** |
| --- | --- | --- |
| **Single** | ✅ Yes | One child class inherits one parent |
| **Multilevel** | ✅ Yes | Child inherits a class which also inherits another |
| **Hierarchical** | ✅ Yes | Multiple child classes inherit one parent class |
| **Multiple** | ❌ Not through classes (only via interfaces) | Java avoids ambiguity caused by diamond problem |
| **Hybrid** | ✅ Partially (via interfaces) | Combination of above types |

### Extends Used for class inheritance

**class Child extends Parent**

**super** keyword is used to call:

· Access parent class's methods or variables

· Call parent class constructor

**Tricky Q:** What happens if you don’t explicitly call super()?

**A** - Java automatically inserts a call to the no-argument constructor of the parent class (super()), but only if the parent class has a no-arg constructor.

· ❓ Can constructors be inherited?

👉 No, but super() can be used to call the parent’s constructor.

· ❓ Does child class inherit private members of the parent?

👉 No, but they can be accessed via **getters/setters**.

· ❓ Can static methods be inherited?

👉 Yes, but they **cannot be overridden**, only hidden.

### 11. ****Method Overriding and Polymorphism****

**Overriding**: Redefining parent method in child

**Polymorphism**: Same interface, different implementation

Think of a **universal remote control**:

The remote is the **reference**.

Different devices (TV, AC, DVD player) are different **object types**.

Pressing "Power" turns on the **device corresponding to the actual object**, not the remote itself.

Same command (method) behaves differently depending on the device (object).

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Compile-time Polymorphism** (Static) | Method overloading or operator overloading resolved at compile time | Multiple methods named add(int a, int b) and add(double a, double b) |
| **Runtime Polymorphism** (Dynamic) | Method overriding, Method call is decided at runtime based on actual object type.  **static, private, and final methods cannot be overridden** → no runtime polymorphism for these. | Child class overrides a parent method |

**Interview Q:** Can private,static, final methods be overridden? (No, as static belong to that class only, and polymorphism is for instances.)

**What is Dynamic Method Dispatch?**

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at runtime, not compile time based on the actual object's type referred to by the reference variable.

**What is Method Hiding?**

Method hiding occurs when a static method in a subclass has the same signature as a static method in the parent class.

Unlike overriding, method hiding means the subclass’s static method hides the parent’s static method.

Method call is resolved at compile time based on the reference type, not runtime object.

### 12. ****this vs super keyword****

### What is super?

super is a **reference variable** that refers to the **immediate parent class object**.

It is used to **access parent class members** (variables, methods, constructors) from a subclass.

### Uses of super:

**Access parent class instance variables** (when variable names are hidden by subclass variables).

**Call parent class methods** (overridden methods).

**Invoke parent class constructor** (to initialize parent class part when subclass is constructed).

this = refers to current class object

super = refers to parent class object

**this() vs super()**: Constructor chaining vs Parent constructor

### ****Text Block and Escape Sequences****

**multi-line string literal**

String text = """

Hello, this is a

multiline string

""";

### Features of Text Blocks:

Reduces need for escape sequences (e.g., \n, \").

Maintains formatting like indentation and line breaks.

Easier to write **JSON, XML, HTML, or SQL** strings.

You can use \ (backslash) to avoid new line at the end of a line.

**Escape Characters : Escape sequences are special characters preceded by a backslash (\) used to represent non-printable or special formatting characters.**

\n (new line), \t(tab space), \", \\ etc.

### 14. ****printf Formatting (C-type formating )****

// Integer

System.out.printf("Value of x = %d%n", x); // Output: Value of x = 25

// Floating-point

System.out.printf("PI = %.2f%n", pi); // Output: PI = 3.14

// String

System.out.printf("Welcome, %s!%n", name); // Output: Welcome, Java!

// Multiple values

System.out.printf("x = %d, pi = %.3f%n", x, pi); // Output: x = 25, pi = 3.142al float

### 15. ****String, StringBuilder, StringBuffer****

**1. String in Java**

**A String is a sequence of characters enclosed in double quotes.**

**In Java, String is a class, not a primitive type.**

**It is immutable, meaning once created, it cannot be changed.**

| **Method** | **Description** |
| --- | --- |
| length() | Returns the length of the string |
| charAt(index) | Returns char at given index |
| substring(begin, end) | Returns substring |
| indexOf(char) | First index of given char |
| equals(str) | Checks content equality |
| equalsIgnoreCase() | Ignores case during comparison |
| toUpperCase() | Converts to upper case |
| toLowerCase() | Converts to lower case |
| replace(a, b) | Replaces characters |
| split(delim) | Splits based on regex/delimiter |
| trim() | Removes leading/trailing spaces |

****Why is String Immutable?****

**For security (e.g., in networking, passwords).**

**For thread safety.**

**Because of String pooling in JVM memory optimization.**

**It prevents data from being changed once it's assigned.**

## 2. StringBuilder in Java

A **mutable** sequence of characters.

Can change its value (append, insert, delete) without creating new objects.

**Not thread-safe**.

StringBuilder sb = new StringBuilder("Hello");

sb.append(" World");

System.out.println(sb);

| **Method** | **Description** |
| --- | --- |
| append(str) | Adds string to the end |
| insert(index, str) | Inserts at given index |
| delete(start, end) | Deletes characters from range |
| reverse() | Reverses the content |
| replace(start, end, str) | Replaces in given range |
| capacity() | Returns current buffer capacity |
| ensureCapacity(n) | Sets min capacity |

## 3. StringBuffer in Java

**Mutable** and **thread-safe** (synchronized).

Slower than StringBuilder due to synchronization.

StringBuffer sb = new StringBuffer("Java");

### When to Use Which?

| **Type** | **Mutability** | **Thread Safe** | **Speed** | **Use Case** |
| --- | --- | --- | --- | --- |
| String | ❌ Immutable | ✅ (because it can't change) | Slow | Constants, configuration, keys, etc. |
| StringBuilder | ✅ Mutable | ❌ Not thread safe | Fast | In single-threaded environments |
| StringBuffer | ✅ Mutable | ✅ Thread safe | Slower | Multi-threaded environment (rarely used) |

## Difference: String vs StringBuilder vs StringBuffer

| **Feature** | **String** | **StringBuilder** | **StringBuffer** |
| --- | --- | --- | --- |
| Mutability | Immutable | Mutable | Mutable |
| Thread Safety | Yes (due to immutability) | No | Yes (synchronized) |
| Performance | Slower | Fastest (no lock) | Slower (with lock) |
| Memory Allocation | New object on change | Same object modified | Same object modified |
| Use Case | Read-only strings | Dynamic content (single-thread) | Dynamic (multi-thread) |

· **Why is String immutable in Java?**

To allow string pooling, enhance security, and enable thread safety.

· **What is the difference between** StringBuilder **and** StringBuffer**?**

Both are mutable, but StringBuffer is thread-safe, StringBuilder is faster in single-threaded cases.

· **Which is more efficient for string concatenation in loops?**

StringBuilder is preferred.

· **Can you convert a** StringBuilder **to** String**?**

Yes: sb.toString();

· **What happens in memory when we concatenate strings with** + **inside a loop?**

It creates multiple temporary String objects (inefficient).

### What is the ****String Pool**** in Java?

The **String Pool** (also known as the **intern pool**) is a special area in Java memory (part of the **heap**) where **string literals** are stored.

### ✅ Key Concept

When you create a string **using literals** like:

String s1 = "Java";String s2 = "Java";

👉 Both s1 and s2 **point to the same object** in the string pool — no new object is created for s2.

### 🔧 How It Works

If a string **literal already exists** in the pool, Java **reuses** it.

If it doesn’t exist, Java **adds it to the pool**.

### ****Composition vs Inheritance****

## What is ****Composition**** in Java?

**Composition** is a design principle where one class contains a reference to another class (HAS-A relationship), meaning one class is **composed of one or more objects** of other classes.

### 🔹 Definition:

"Composition is when an object of one class is used in another class as a member field."

### 🎯 Real-Life Example

class Engine {

void start() {

System.out.println("Engine started.");

}

}

class Car {

private Engine engine = new Engine(); // Composition

void drive() {

engine.start();

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

}

}

**Explanation:**

A **Car** HAS-A **Engine**.

The Car class uses Engine without extending it.

**Composition:** HAS-A

**Inheritance:** IS-A

**Interview Q:**

| **❓ Question** | **✅ Answer** |
| --- | --- |
| What is composition in Java? | Using an object of one class inside another (HAS-A relationship). |
| Difference between composition and inheritance? | Composition uses another class; inheritance extends a class. |
| Why prefer composition over inheritance? | More flexible, modular, avoids tight coupling. |
| Real-world use of composition? | Car has Engine, Human has Heart, Student has Address. |
| Can we use both composition and inheritance? | Yes, often together in complex designs. |

**Abstraction**

Abstraction is the process of **hiding internal implementation details** and showing **only the essential features** of an object.

### 🧠 Real-Life Analogy

**Car**: When you drive a car, you use the **steering wheel**, **accelerator**, and **brakes** — but you don’t know (or need to know) **how the engine works internally**.

That’s **abstraction** — **only showing the necessary functionality**.

### 🔧 How to Achieve Abstraction in Java

| **Tool** | **Description** |
| --- | --- |
| abstract class | Partially abstract (can have both abstract and non-abstract methods) |
| interface | Fully abstract (until Java 8), used to define contract or capability |

### 🔹 Abstract Class Example

abstract class Animal {

abstract void makeSound(); // Abstract method

void sleep() {

System.out.println("Sleeping...");

}

}

class Dog **extends** Animal {

void makeSound() {

System.out.println("Bark");

}

}

### 🔹 Interface Example

interface Vehicle {

void start();

void stop();

}

class Car **implements** Vehicle {

public void start() {

System.out.println("Car started");

}

public void stop() {

System.out.println("Car stopped");

}

}

### 🔍 Use Cases of Abstraction

| **Use Case** | **Explanation** |
| --- | --- |
| Hiding Complexity | Users don’t see internal logic — only necessary operations (e.g., APIs) |
| Defining Contracts | Interfaces let classes commit to a common structure |
| Plug-and-Play Design | Allows changing internal implementation without changing calling code |
| Framework Development | Abstract classes/interfaces define reusable code skeletons |
| Payment Gateway Integration | Expose just pay(), internally support credit card, wallet, UPI, etc. |

### 🧠 Key Points

You **cannot instantiate** abstract classes.

Interfaces support **multiple inheritance** of type.

Abstraction focuses on **what to do**, not **how to do it**.

An abstract class can contain:

Constructors

Final, static, and non-static fields

Non-abstract methods

### 🤔 Abstraction vs Encapsulation

| **Aspect** | **Abstraction** | **Encapsulation** |
| --- | --- | --- |
| Focus | Hiding internal logic | Hiding data (variables) |
| Achieved Using | Abstract classes, Interfaces | Access modifiers (private, etc.) |
| Example | User knows start(), not logic | Data is private, accessed via getter/setter |

### 🧠 Important Interview Questions

| **❓ Question** | **✅ Answer** |
| --- | --- |
| What is abstraction? | Hiding internal details, showing only essential features. |
| How is abstraction achieved in Java? | Using abstract classes and interfaces. |
| Can an abstract class have a constructor? | Yes, but it can’t be used to create objects directly. |
| Difference between abstraction and encapsulation? | Abstraction hides implementation; encapsulation hides data. |
| When would you use an interface over an abstract class? | When you want to define a pure contract, or support multiple inheritance of types. |
| Can we have abstract and concrete methods in the same class? | Yes, in abstract classes. |

### var ****and Type Inference (Java 10+)****

var is a local variable type inference keyword.

Java lets the compiler infer the type of a variable from the right-hand side value, so you don't have to explicitly write it.

var name = "Java"; // inferred as String

**Tip:** Use Only for local variables, improves readability.

| **❓ Question** | **✅ Answer** |
| --- | --- |
| What is var in Java? | It enables local variable type inference (introduced in Java 10). |
| Can var be used for method parameters? | ❌ No. Only for local variables. |
| Is var dynamic typing like in Python or JavaScript? | ❌ No. Java is still statically typed. |
| Can you assign null to a var? | ❌ No. Type cannot be inferred from null. |
| Will var make Java dynamically typed? | No, types are still checked at compile-time. |

### ****Casting with Classes****

When it comes to classes, casting is about converting one object reference type into another within the same inheritance hierarchy.

| **Type** | **Description** |
| --- | --- |
| **Upcasting** | Converting a subclass object to a superclass reference |
| **Downcasting** | Converting a superclass reference back to a subclass |

class Animal {

void speak() {

System.out.println("Animal speaks");

}

}

class Dog extends Animal {

void bark() {

System.out.println("Dog barks");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

Animal a = dog; // Upcasting

a.speak(); // ✅ Allowed

// a.bark(); ❌ Not allowed – a is Animal reference

}

}

Animal a = new Dog(); // Upcasting

Dog d = (Dog) a; // Downcasting

d.bark(); // ✅ Allowed

**Object Casting:**

Object obj = "Hello";

String str = (String) obj;

### 22. ****.getClass() and .getSimpleName()****

obj.getClass().getSimpleName();

Returns the class name of the object

### ****Object Class and Its Methods****

Object class is the **root superclass** of all classes.

📌 Every class in Java **implicitly** extends java.lang.Object if no other superclass is specified.

This means all classes inherit the methods defined in the Object class.

| **Method** | **Purpose** | **Common Use Case** |
| --- | --- | --- |
| equals(Object obj) | Checks if two objects are **logically equal** | Comparing values of objects |
| hashCode() | Returns integer hash code of the object | Used in hash-based collections (like HashMap) |
| toString() | Returns a **String representation** of the object | Debugging, logging |
| getClass() | Returns **runtime class type** of the object | Reflection, logging |
| clone() | Creates and returns a copy of the object | Deep/shallow copy (requires Cloneable) |
| finalize() | Called by GC before object is destroyed (deprecated in Java 9) | Rarely used |
| wait(), notify(), notifyAll() | Used for **thread communication** | Multithreading (synchronized blocks) |

| **Topic** | **Explanation** |
| --- | --- |
| equals() vs == | equals() checks logical equality; == checks reference |
| Why override hashCode() with equals()? | Required by contract: equal objects must have same hashCode |
| When is finalize() called? | Before GC collects object (deprecated now) |
| Why use getClass()? | Useful in frameworks, dynamic object inspection |
| Is Object an interface? | ❌ No, it’s a concrete class |
| Can we create Object class instance? | ✅ Yes: Object obj = new Object(); |

**Interview Q:** Why all classes inherit from Object?

## 🔷 What is an Array in Java?

An **array** is a **container object** that holds a **fixed number of elements** of a **single data type**.

Arrays are **indexed**, **zero-based**, and **store elements contiguously in memory**.

✅ Arrays in Java are **objects** that **inherit from** java.lang.Object, regardless of the data type they store.

## 🛠️

## 1. ****Declaration of Arrays****

There are two ways:

int[] arr; // preferred style

int arr[]; // also valid

## 🔨 2. ****Instantiation of Arrays****

You must specify the size when you create an array:

arr = new int[5]; // creates array with 5 elements, default = 0

Or combine declaration and instantiation:

int[] arr = new int[5];

## 🧾 3. ****Array Initializer (Shortcut)****

int[] numbers = {1, 2, 3, 4, 5}; // size = 5, values assigned

Java internally creates and fills the array.

## 🧠 4. ****Arrays are Objects****

int[] arr = new int[3];

System.out.println(arr.getClass()); // class [I (I = int)

System.out.println(arr instanceof Object); // true

So, you can assign any array to an Object reference:

Object obj = new int[10];

## 🔹 5. ****Default Values in Arrays****

| **Data Type** | **Default Value** |
| --- | --- |
| int, byte, short | 0 |
| double, float | 0.0 |
| char | \u0000 (null char) |
| boolean | false |
| Objects (String, etc) | null |

## 🔁 6. ****Enhanced For Loop (For-each loop)****

Used to iterate through arrays cleanly.

int[] arr = {10, 20, 30};

for (int num : arr) {

System.out.println(num);

}

✅ No index needed

❌ Cannot modify array elements directly

## 📦 7. ****java.util.Arrays – Powerful Utility Class****

Import:

import java.util.Arrays;

### ✅ Commonly Used Methods:

| **Method** | **Use** |
| --- | --- |
| Arrays.toString(arr) | Convert array to readable String |
| Arrays.sort(arr) | Sort array in ascending order |
| Arrays.equals(a, b) | Check if arrays are equal |
| Arrays.fill(arr, val) | Fill array with a value |
| Arrays.copyOf(arr, len) | Create copy with specific length |
| Arrays.binarySearch(arr, key) | Search key in sorted array |

### 🔍 Examples:

int[] a = {4, 2, 1};

Arrays.sort(a); // [1, 2, 4]

System.out.println(Arrays.toString(a)); // [1, 2, 4]

int index = Arrays.binarySearch(a, 2); // returns index: 1

int[] b = Arrays.copyOf(a, 5); // [1, 2, 4, 0, 0]

## 🔢 8. ****Assigning Array to Object Reference****

int[] arr = {1, 2, 3};

Object obj = arr;

System.out.println(obj instanceof int[]); // true

✅ This helps in polymorphic use and generic handling.

## 🔍 9. ****Array Searching and Sorting Algorithms****

### 🔹 Searching Algorithms:

| **Algorithm** | **Description** | **Use Case** |
| --- | --- | --- |
| **Linear Search** | Search one-by-one | Small or unsorted arrays |
| **Binary Search** | Repeated half-split (requires sorted array) | Fast search in large sorted arrays |

🔹 Example – Binary Search:

int[] a = {1, 3, 5, 7};

int index = Arrays.binarySearch(a, 5); // returns 2

### 🔹 Sorting Algorithms (Java uses TimSort internally)

You can implement:

| **Algorithm** | **Time Complexity** | **Stable?** | **Best For** |
| --- | --- | --- | --- |
| Bubble Sort | O(n²) | ✅ | Learning purposes |
| Selection Sort | O(n²) | ❌ | Simple logic |
| Insertion Sort | O(n²) | ✅ | Small data |
| Merge Sort | O(n log n) | ✅ | Large stable sort |
| Quick Sort | O(n log n) avg | ❌ | Fastest general-purpose |
| Heap Sort | O(n log n) | ❌ | Balanced sort |

But usually, just use:

Arrays.sort(arr); // Optimized built-in sorting

## 🧠 Interview Tips and Tricky Points

| **Question** | **Concept** |
| --- | --- |
| What is default value of boolean array? | false |
| Can you assign an array to Object? | ✅ Yes |
| Can you change array size once created? | ❌ No, fixed size |
| Can you store multiple data types in array? | ❌ Only one type (but can use Object[]) |
| Is array an object? | ✅ Yes |
| Can array length be negative? | ❌ No, throws NegativeArraySizeException |