JAVA INTODUCTION:

💬 Definition:

> Java is a high-level, object-oriented, platform-independent, robust, and secure programming language developed by Sun Microsystems (now owned by Oracle).

It follows the principle of "Write Once, Run Anywhere."

🔑 Core Keywords / Features of Java:

Simple,Easy to learn and use,Syntax is simple compared to C++; no pointers, no header files, automatic memory management.

Object-Oriented: Everything is based on objects and classes.Uses class, object, inheritance, polymorphism, encapsulation, abstraction.

Platform-Independent: Works on any OS Java code compiles into bytecode, which runs on any JVM (Write Once, Run Anywhere).

Robust: Strong and reliable Has exception handling, garbage collection, and type checking at compile and runtime.

Secure :Safe from viruses and unauthorized access No direct memory access (no pointers), uses bytecode verification and security manager.

Portable: Can move programs across systems easily Bytecode works on all platforms that have JVM installed.

Multithreaded : Can run multiple tasks at the same time Java provides Thread class and Runnable interface to handle multitasking.

Distributed :Can work over networks easily Supports technologies like RMI, JDBC, and web services for distributed apps.

Dynamic :Adapts while running Supports runtime polymorphism, and classes can be loaded dynamically using ClassLoader.

Architecture-Neutral: Not tied to any hardware Bytecode is designed to be executed on any CPU with a JVM.

High Performance Faster than other interpreted languages Uses JIT (Just-In-Time) compiler inside the JVM.

Interpreted + Compiled Uses both compiler and interpreter javac compiles to bytecode, and JVM interprets or compiles it at runtime.

✅ Java Keywords Explained (Simple + Interview Level)

1. class

- Used to declare a class (blueprint of objects).
- Everything in Java lives inside a class.

👉 Interview line:
"A class is a template that defines state and behavior of objects."

---

2. public

- Access modifier → visible everywhere.
- JVM requires main() to be public so it can be accessed from outside the class.

👉 Interview line:
"public means anyone can access that class or method."

---

3. static

- Belongs to the class, not the object.
- No object creation needed.
- JVM calls the main method directly using the class name.

👉 Interview line:
"Static methods load into memory when the class is loaded."

Java ensures security at both compile time and runtime.

At compile time, the compiler checks for syntax and type errors.

At runtime, the JVM uses the ClassLoader, Bytecode Verifier, and Security Manager to verify that the code is safe, legal, and doesn't perform any unauthorized actions."

---

4. void

- Means method returns nothing.
- JVM does not expect any return value from main().

👉 Interview line:
"void indicates no value is returned by the method."

---

5. main

- The entry point of every Java program.
- Signature must be exactly:
- public static void main(String[] args)

👉 Interview line:
"main is the first method executed by JVM after class loading."

6. String

- Predefined class in java.lang.
- Represents text data.
- Immutable (value cannot change once created).
- args → command-line arguments.

👉 Interview line:
"String is a class in Java, not a primitive type."

Keyword	Meaning
class	Blueprint for objects
public	Accessible anywhere
static	Belongs to class, not object
void	No return value
main	Program entry point
String	Text data type/class

💬 Simple Overview of Variables in Java

👉 What:

A variable is a name given to a memory location where data is stored.

👉 Why Java introduced variables:

Because in real life, we work with data (like names, prices, marks, etc.).

To store, use, and process this data in programs, Java introduced variables.

Without variables, Java cannot remember or handle any value.

👉 Why we use variables:

We use variables to:

Store data (like numbers, text, etc.)

Reuse and update data later in the program

Make programs dynamic (accept user input, calculations, etc.)

✅ Simple Interview Line

> Variables in Java are used to store and process data. They make programs dynamic and help us handle information efficiently during runtime

👉 Where we use variables:

We use them anywhere data is needed —

inside methods, classes, loops, or conditions.

Example:

```
int age = 25;
if (age >= 18) {
   System.out.println("Eligible to vote");
```

☞ When we use variables:

Whenever we need to store or process any information in the program —

like during calculations, input/output, database storage, or displaying results.

 Example Summary

String name = "Harika"; // to store name

int age = 22; // to store number

double salary = 45000.50; // to store decimal

We can use these variables anytime in the program.

☞ How we use variables:

We first declare and then assign a value.

Example:

int marks; // Declaration

marks = 90; // Assignment

or in one line:

int marks = 90;

💬 Overview of Data Types in Java

In Java, data types define the type of data a variable can hold.

They tell the compiler what kind of value (number, text, true/false, etc.) will be stored

🔹 Two main categories:

1. Primitive Data Types
2. Non-Primitive (Reference) Data Types

🔹 1. Primitive Data Types

Primitive types are the basic building blocks of Java.

They directly store values in memory and work on bytes.

A byte consists of 8 bits, and each bit represents either 0 or 1.

Type	Description	Size	Example
byte	Small integer	1 byte	byte b = 10;
short	Medium integer	2 bytes	short s = 200;
int	Whole number (most used)	4 bytes	int age = 25;
long	Large integer	8 bytes	long distance = 123456L;
float	Decimal (single precision)	4 bytes	float price = 99.9f;
double	Decimal (high precision)	8 bytes	double salary = 45000.50;
char	Single character	2 bytes	char grade = 'A';
boolean	True or False	1 bit (logical)	boolean isPassed = true;

 Note: Primitive types store actual values, not refeerences

2. Non-Primitive Data Types

These store references (memory addresses), not direct values.

Examples:

String → String name = "Harika";

Arrays → int[] marks = {90, 85, 70};

Classes, Objects, Interfacees

✅ Interview Line Example

> Java provides 8 primitive data types that define the size and type of variable values. These types make Java memory-efficient and strongly typed.

Non-primitive types like String and arrays are created by users and store object references.

💬 Type Conversion vs Type Casting in Java

Both are used to convert one data type into another,

but the way they happen is different.

1️⃣ Type Conversion (Automatic / Implicit Casting)

✅ Definition:

When Java automatically converts one data type to another without the programmer's

🔹 Also called Widening Conversion — small → large data type.

This is safe, so Java does it automatically.

📌 Example:

```
int num = 10;

double result = num; // int automatically converted to double

System.out.println(result); // Output: 10.0
```

💡 Here, Java converts int → double automatically.

Because double has more memory than int.

✅ Happens automatically

✅ No data loss

✅ Safe conversion

✅ Simple Interview Answer

> Type Conversion happens automatically when Java converts smaller data types to larger ones (like int to double).

Type Casting is manual done by the programmer to convert larger data types into smaller ones (like double to int), which may cause data loss."

```java
package Basics;

public class TypeConversion {

    public static void main(String[] args) {

        byte b = 10;
        int i = b;          // byte → int (widening)

        int x = 20;
        double d = x;     // int → double

        char c = 'A';
        int ascii = c;    // char → int (65)


        System.out.print(i);
    }
}
```

2️⃣ Type Casting (Manual / Explicit Casting)

✅ Definition:

When the programmer manually forces a conversion from one type to another.

🔹 Also called Narrowing Conversion — large → small data type.

This can cause data loss or precision loss, so we must do it manually using (type).

📌 Example:

double salary = 99.99;

int value = (int) salary; // double manually casted to int

System.out.println(value); // Output: 99

💡 Here, the decimal part .99 is lost — so data loss can occur.

- ➤ ❌ Not automatic
- ➤ ⚠️ Possible data loss
- ➤ ✋ Done by developer

Quick Difference Table

Feature	Type Conversion	Type Casting
Other Name	Widening	Narrowing
Done By	Java (Automatic)	Programmer (Manual)
Conversion	Small → Large	Large → Small
Data Loss	No	Possible
Syntax	Automatic	(type)variable
Example	int x = 10; double y = x;	double x = 10.5; int y = (int)x;

```java
package Basics;

public class TypeCasting {

    public static void main(String[] args) {

        double d = 10.8;
        int i = (int) d;    // double → int
        // output: 10

        int x = 130;
        byte b = (byte) x; // overflow
        // output: -126

        long l = 1000;
        short s = (short) l;

        System.out.println(x);

    }

}
```

💬 What are Conditions in Java?:

👉 Definition:

Conditions (or decision-making statements) help Java programs make decisions based on certain situations or data.

They decide what to do next — just like how we decide things in real life.

💡 Why we use Conditions in real time

In real projects, we often need to check situations and take actions —

for example:

If user login is correct → show dashboard

Else → show error message

If stock is less → send alert

Else → continue billing

So, conditions help the program think logically and make decisions.

⚙ Types of Conditional Statements in Java

1️⃣ If Statement

Used when we want to execute a block only if a condition is true.

```
int age = 18;
if (age >= 18) {
   System.out.println("Eligible to vote");
}
```

2️⃣ If-else Statement

Used when there are two possible outcomes.

```
int age = 16;
if (age >= 18) {
   System.out.println("Eligible to vote");
} else {
   System.out.println("Not eligible to vote");
}
```

3️⃣ else if Ladder

Used when there are multiple conditions.

```
int marks = 85;
if (marks >= 90) {
    System.out.println("Excellent");
} else if (marks >= 75) {
    System.out.println("Good");
} else {
    System.out.println("Needs Improvement");
}
```

4️⃣ switch Statement

Used when we have many choices for a single value.

```
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");
}
```

✅ Simple Interview Line

> Conditions in Java help us control the flow of the program by making decisions based on situations.

In real time, they're used whenever we need to check user inputs, compare values, or decide which action to take next.

💬 What is a Ternary Condition in Java?

👉 The ternary operator is a shortcut for writing simple if-else conditions in one single line.

It is used for decision-making just like if-else, but it makes code shorter and cleaner.

🧠 Syntax:

condition ? expression1 : expression2;

If the condition is true, → expression1 executes.

If the condition is false, → expression2 executes.

💡 Example:

```
int age = 20;
String result = (age >= 18) ? "Eligible to vote" : "Not eligible to vote";
System.out.println(result);
```

✅ Output: Eligible to vote

🧠 Here, instead of writing:

```
if (age >= 18)
  result = "Eligible to vote";
else
  result = "Not eligible to vote";
```

we used just one line!

📘 Why We Use Ternary in Java

✅ To reduce code size

✅ To make conditions simple and readable

✅ To assign values quickly based on a condition

✅ Mostly used in short decisions, like:

- Checking login success/failure
- Assigning grades
- Displaying status (active/inactive)

🔧 Real-Time Example:

```
double salary = 50000;

String taxStatus = (salary > 30000) ? "Tax Applicable" : "No Tax";

System.out.println(taxStatus);
```

✅ Output: Tax Applicable

✅ Simple Interview Answer:

> “The ternary operator in Java is a shorthand version of the if-else statement.

It helps in writing simple conditional logic in a single line, making the code cleaner code

💬 Why We Use Loops in Java

👉 In real life, we often repeat the same task many times —

like sending daily emails, checking every item in a list, or counting from 1 to 100.

If we do this manually, it's time-consuming.

So, in Java, we use loops to automate repetitive tasks —

the computer repeats the work automatically until a condition is false

□ Simple Definition:

A loop in Java is used to execute a block of code multiple times

until a given condition becomes false.

💡 Example:

```
for (int i = 1; i <= 5; i++) {
   System.out.println("Hello Java");
}
```

✅ Output:

Hello Java

Hello Java

Hello Java

Hello Java

Hello Java

Instead of writing 5 print statements manually,

the loop repeats it automatically.

🔁 Why Loops Are Used (Real-Time Use)

✅ To automate repetitive work

✅ To process large data (like lists, arrays, or files)

✅ To save time and reduce manual errors

✅ To iterate over user inputs, database records, or API data

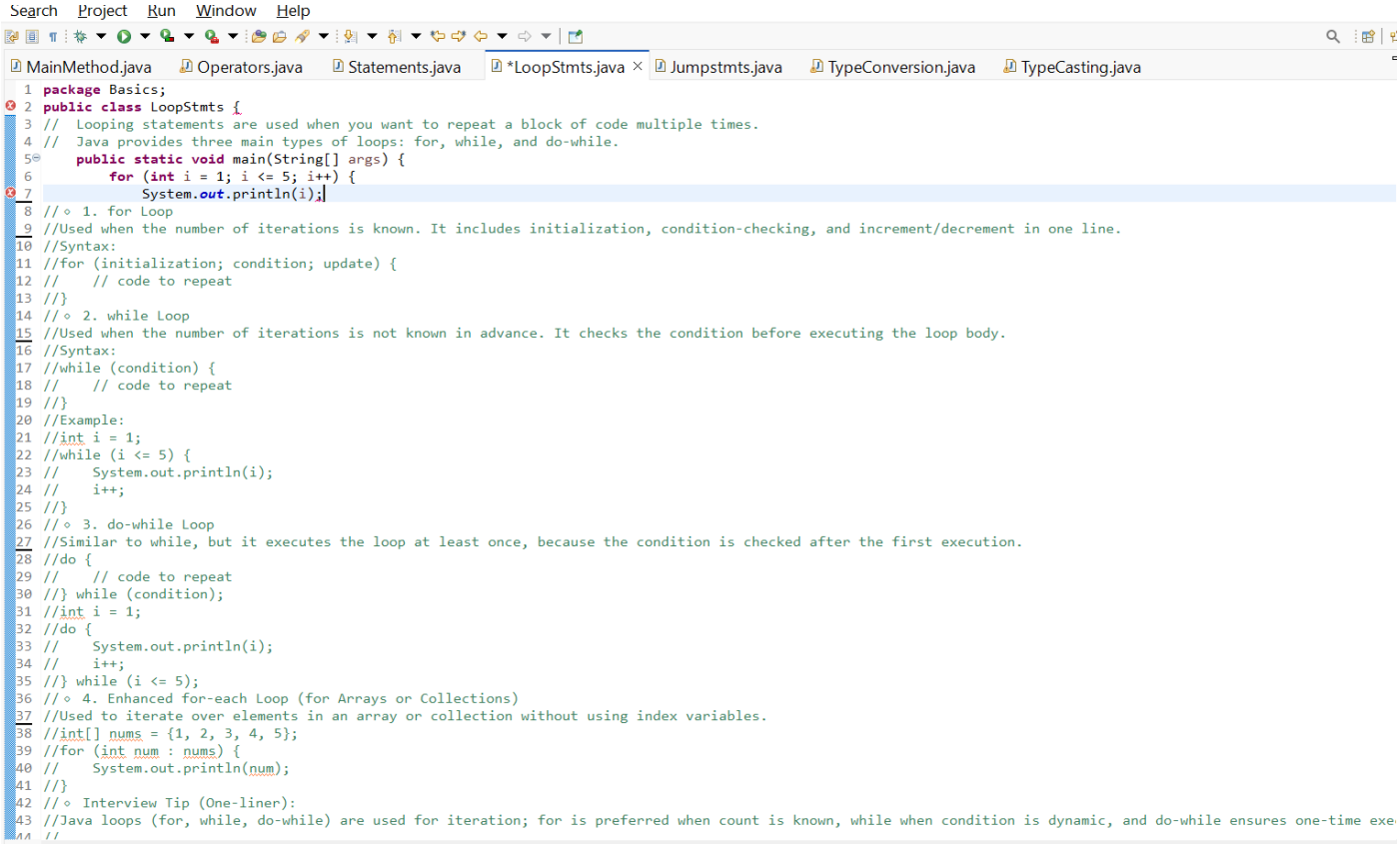
⚙ Types of Loops in Java:

Loop Type	Use Case	Example
for loop	When you know how many times to repeat	for(int i=0; i<5; i++)
while loop	When you don't know how many times, while(condition)	depends on condition
do-while loop	Runs at least once, even if condition is false	do { } while(condition);
for-each loop	Used to loop through arrays/collections	for(int num : numbers)

✅ Simple Interview Line

> "Loops in Java are used to perform repetitive tasks automatically instead of doing them manually.

They help save time and make programs more efficient by running the same logic multiple times under a condition.



💬 What are Literals in Java?

👉 A literal is a fixed value that you directly use in your program —

it represents constant data that does not change while the program runs.

📓 Example:

int num = 10; // 10 is a literal

char letter = 'A'; // 'A' is a literal

boolean flag = true; // true is a literal

So, whenever we assign a value to a variable,

that value is called a literal.

🔹 Types of Literals in Java

Type	Example	Description
Integer Literal	`10, -25, 0`	Whole numbers
Floating Literal	`12.5, -3.14, 5.0f`	Decimal numbers
Character Literal	`'A', '1', '#'`	Single character (uses ASCII/Unicode values)
String Literal	`"Hello", "Java"`	Sequence of characters
Boolean Literal	`true, false`	Logical values
Null Literal	`null`	Represents no value / empty reference

Character Literals and ASCII Values

characters in Java are internally stored using Unicode values (which include ASCII values).

For example:

```java
package Basics;

public class AsciiConvertChar {

    public static void main(String[] args) {
        char ch = 'A';
        int ascii = ch; // Converts char to its ASCII value
        System.out.println(ascii);


    }

}
```

✅ Output:

65

💡 So, 'A' is a character literal,

and its ASCII/Unicode value is 65.

🔹 Real-Time Example

Let’s say we are checking if a user pressed the correct letter:

```
char key = 'Y';
if (key == 'Y' || key == 'y') {
   System.out.println("You agreed!");
} else {
   System.out.println("You declined!");
}
```

Here 'Y' and 'y' are character literals.

✅ Simple Interview Line

> Literals in Java are constant values assigned to variables.

They represent fixed data like numbers, characters, or strings.

Character literals are stored using Unicode (which includes ASCII values).

This one is very useful, especially if the interviewer asks about your coding setup or tools you use for Java development.

💬 What is an IDE in Java?

👉 IDE stands for Integrated Development Environment.

It's a software tool that helps developers write, run, debug, and test Java programs easily — all in one place.

It combines:

Code editor ✍️

Compiler / Interpreter ⚙️

Debugger 🐞

Error checking tools ✅

Build & Run options ▶️

🧠 Why We Use IDEs

✅ To write code faster

✅ To automatically check for syntax errors

✅ To suggest code completions (intellisense)

✅ To debug programs easily

✅ To manage multiple Java files and projects

💻 Popular IDEs for Java

IDE	Description	Best For
Eclipse	Widely used open-source IDE	Beginners to advanced developers
IntelliJ IDEA	Fast, smart, and user-friendly	Professional developers
NetBeans	Simple setup with an easy UI	Learning and small projects
VS Code (Java Extension)	Lightweight, supports multiple languages	Simple coding and practice

⚙ Steps to Set Up a Java IDE (Example: Eclipse)

1. Install Java JDK (Java Development Kit)

◈ Download from Oracle

◈ Set JAVA_HOME and add to PATH

2. Install Eclipse IDE

◈ Download from eclipse.org

◈ Choose “Eclipse IDE for Java Developers”

3. Open Eclipse → Create a New Java Project

File → New → Java Project

Give project name (e.g., MyFirstJavaProject)

4. Create a New Class

Right-click src → New → Class → name it HelloWorld

Write your code:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, Java!");
    }
}
```

5. Run the Program

Click ▶ (Run Button)

Output shows: Hello, Java

✅ Simple Interview Line

> An IDE (Integrated Development Environment) helps developers write, run, and debug Java code easily.

I mostly use Eclipse/IntelliJ because it automatically checks syntax errors, provides suggestions, and helps execute programs quickly.

Source Code (.java) // Developer writes code

↓

[Compiled by javac in JDK] // CTRL + C (compile)

↓

Bytecode (.class) // Platform-independent

↓

[Executed by JVM in JRE] // Run

↓

Machine Code (Output) // Final program result

💬 1️⃣JVM (Java Virtual Machine)//platform dependent

👉 Definition:

JVM is the engine that runs Java programs.

It converts bytecode into machine code that your computer can understand.

💡 Think of JVM as the runner — it executes your Java program.

📌 Main work:

Loads code

Verifies code

Executes code

Handles memory & garbage collection

Platform-dependent (different for Windows, Mac, Linux)

But makes Java platform-independent (because the same bytecode runs everywhere)

💬 2️⃣ JRE (Java Runtime Environment)

👉 Definition:

JRE provides the environment in which the JVM runs.

It includes everything needed to run a Java program — but not to develop one.

📄 JRE = JVM + Libraries + Class files + Other runtime tools

💡 It's for users who just run Java applications, not write them.

💬 3️⃣ JDK (Java Development Kit)

👉 Definition:

JDK is the complete package for Java developers.

It includes everything in JRE, plus tools for development (like compiler).

📄 JDK = JRE + Compiler (javac) + Debugger + Other development tools

💡 Used by developers to write, compile, and run Java programs.

🧩 Simple Example:

When you write and run:

```
class Hello {
    public static void main(String[] args) {
        System.out.println("Hello Java");
    }
}
```

- You write code → using JDK (because it includes compiler javac)
- You compile it → compiler converts it into bytecode (Hello.class)
- You run it → JVM executes it inside the JRE

□ Simple Diagram

JDK → contains JRE → contains JVM

Or in words:

> JDK = JRE + Development Tools

JRE = JVM + Libraries

✅ Simple Interview Line

> JDK is used to develop Java applications, JRE is used to run them,

and JVM is the engine that actually executes the bytecode.

Together, they make it

💡 Class in Java – Simple Meaning:

A class in Java is like a blueprint or plan for creating objects.

It defines how an object will look (its variables) and what it can do (its methods).

🏠 Real-Time Example:

Think of a class as a blueprint of a house 🏠

Blueprint → Class

Actual houses built from it → Objects

Without the blueprint, you can't build houses correctly —

just like without a class, you can't create objects in Java.

🧠 Interview-style short answer:

> A class in Java is like a blueprint — it defines properties and behavior of objects.

Without a class, we can't create objects, just like we can't build a house without a plan.

 Java Example:

```
// Class = blueprint
class Car {
   String color;//properties
   int speed;//properties
   void drive() {//non static methods//behaviours
      System.out.println("Car is driving...");
   }
}
// Object = actual car
public class Main {
   public static void main(String[] args) {
      Car myCar = new Car(); // creating object
      myCar.color = "Red";
      myCar.speed = 100;
myCar.drive();
```

✅ Car → blueprint

✅ myCar → real car built from blueprint

Object in Real Time

An object is something that has:

1. Properties (What it has / What it knows)

2. Behaviors (What it can do / How it acts)

Real-Life Example:

Let's take an example — a Car

Concept	Example	Meaning
Property (What it knows)	Color, Brand, Speed	Information about the car
Behavior (What it does)	Drive, Stop, Horn	Actions the car performs

So —

> The car (object) knows its color, brand, and speed,

and it can perform actions like driving or stopping.

In Java Terms:

```
class Car {
    // Properties (What it knows)
    String color;
    int speed;
    // Behaviors (What it does)
    void drive() {
        System.out.println("Car is driving...");
    }
    void stop() {
        System.out.println("Car stopped.");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();  // creating object
        myCar.color = "Red";    // setting property
        myCar.speed = 100;      // setting property
        myCar.drive();          // calling behavior
    }
}
```

✅ Properties: color, speed

✅ Behaviors: drive(), stop()

Interview Explanation (Simple):

> “An object in Java represents a real-world entity.

It has properties (data) — what it knows,

and behaviors (methods) — what it can do.”

Example for Human Object:

Property Behavior

name, age speak(), walk(), eat(

Short Answer for Interview:

> Objects have properties that define their data and behaviors that define their actions.

For example, a car knows its color and speed (properties) and can drive or store

💡 Class and Object Relationship (Simple Meaning)

A class is just a blueprint or plan — it doesn't do anything by itself.

An object is the real thing created from that plan — it actually does the work.

🏠 Real-life Example:

Imagine you have a blueprint of a house 🏘️

The blueprint shows how the house should look (rooms, doors, windows).

But you can't live in the blueprint — right? 😄

You must build a real house (object) from that blueprint to actually use it.

✅ Blueprint → Class

✅ Built house → Object

💻 In Java Terms:

```
class Car {
    String color;
    void drive() {
        System.out.println("Car is driving...");
    }
}
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();  // Object created from class
        myCar.color = "Red";
        myCar.drive();          // Now the plan is in action
    }
}
```

Car → plan/blueprint (class)

myCar → real car that actually drives (object)

 Interview Explanation:

> “A class in Java is just a plan or blueprint that defines how objects will look and behave.

Without creating objects, the class has no use — just like a building plan without a real building.”

🗣️ Short Answer (for Interview):

> “Class is a plan; object is the real thing.

Without an object, a class is only a design —

💡 1️⃣ String in Java

String in Java is a class, not a primitive data type.

It is used to store and handle text (sequence of characters).

Example:

String name = "Harika";

🔄 2️⃣ Mutable vs Immutable

Term	Meaning
Immutable	Once created, the value cannot be changed.
Mutable	The value can be changed or modified later.

🔒 3️⃣ String is Immutable

When you change a string, a new object is created in memory instead of modifying the old one.

Example:

String s1 = "Java";

s1 = s1 + " Developer";

System.out.println(s1);

Here:

"Java" → created first

"Java Developer" → new object created

old "Java" is not changed

✅ This is why String is immutable — once created, it can’t be modifity

StringMutable.java

```java
package Basics;

public class StringMutable {

    public static void main(String[] args) {
        String s1 = "john cena";
        String s2 = s1; // both point to same object
        s1 = "tiger"; // reassignment
        System.out.println(s1 == s2);
    }

}
```

⚙️ 4️⃣ Mutable Strings — StringBuffer & StringBuilder

Class	Mutability	Thread Safety	Speed
String	Immutable	Thread-safe (because immutable)	Slower
StringBuffer	Mutable	Thread-safe (synchronized)	Slower
StringBuilder	Mutable	Not thread-safe	Faster

StringBuffer Example:

```java
        StringBuffer sb1 = new StringBuffer("Hello")
        StringBuffer sb2 = sb1;

        sb1.append(" Harika");

        System.out.println(sb1 == sb2);
        System.out.println(sb2);
    }
```

✅ It modifies the same object → no new object created.

💬 StringBuilder Example:

StringBuilder sb = new StringBuilder("Hello");

sb.append(" Java");

System.out.println(sb); // Output: Hello Java

✅ Similar to StringBuffer, but faster because it's not synchronized.

5️⃣ Why String Immutable?

1. Security – String used in URLs, file paths, etc. shouldn't be changed accidentally.

2. Caching – JVM stores common strings in String Pool (saves memory).

3. Thread Safety – Immutable means safe to use across threads.

🎯 6️⃣ Interview Summary Answer:

> "In Java, String is a class used to handle text data.

Strings are immutable, meaning once created, they cannot be changed — every modification creates a new object.

For mutable strings, we use StringBuffer (thread-safe) or StringBuilder (faster, non-thread-safe).

Both support the append() method to add or modify content."

🧠 How String is stored in JVM memory

Java stores Strings in **TWO places**:

1️⃣ String Constant Pool (SCP)

- Special memory area inside Heap
- Stores String literals
- Avoids duplicate objects → saves memory

Example

String s1 = "Hari";

String s2 = "Hari";

Memory:

String Constant Pool

"Hari" ← s1, s2

✓ Only **one object** created

✓ Both references point to same object

2️⃣ Heap Memory (Using new keyword)

String s3 = new String("Hari");

Memory:

Heap Memory

"Hari" ← s3 (new object)

❌ Not reused

❌ Separate object created

🔍 Proof using code

String s1 = "Hari";

String s2 = "Hari";

String s3 = new String("Hari");

System.out.println(s1 == s2); // true

System.out.println(s1 == s3); // false

🔁 intern() method (Important)

String s4 = s3.intern();

System.out.println(s1 == s4); // true

✓ Moves string reference to **String Constant Pool**

□ Full Memory Diagram (Easy)

Stack

s1 → "Hari"

s2 → "Hari"

s3 → heap object

Heap

String Object ("Hari")

String Constant Pool (inside heap)

"Hari"

🔑 Interview-ready answer

String literals are stored in the String Constant Pool inside heap memory. When the same literal is used multiple times, JVM reuses the same object. Strings created using new keyword are stored as separate objects in heap memory. This improves memory efficiency and performance.

🤔 Common confusion (You are not alone)

❌ String is **NOT stored in stack**

✓ Reference variable is stored in stack

✓ Actual String object is in heap / SCP

> In Java, memory is divided into two parts — Stack and Heap.

The Stack memory stores:

Method calls (main, other methods)

Local variables

Object references

The Heap memory stores:

Actual objects

Instance variables of those objects

> When a method (like main()) is called, a stack frame is created in the Stack.

If you create an object inside it, the object is stored in the Heap, and only the reference to that object is stored in the Stack

 Example:

```
public class Demo {
   int x = 10;
   public static void main(String[] args) {
      Demo d = new Demo();
      System.out.println(d.x);
   }
}
```

 Explanation:

main() → stored in Stack.

d → reference variable in Stack.

new Demo() → object created in Heap (with instance variable x = 10).

When program ends or d = null, object in Heap becomes eligible for Garbage Collection.

🗑️ Garbage Collection concept:

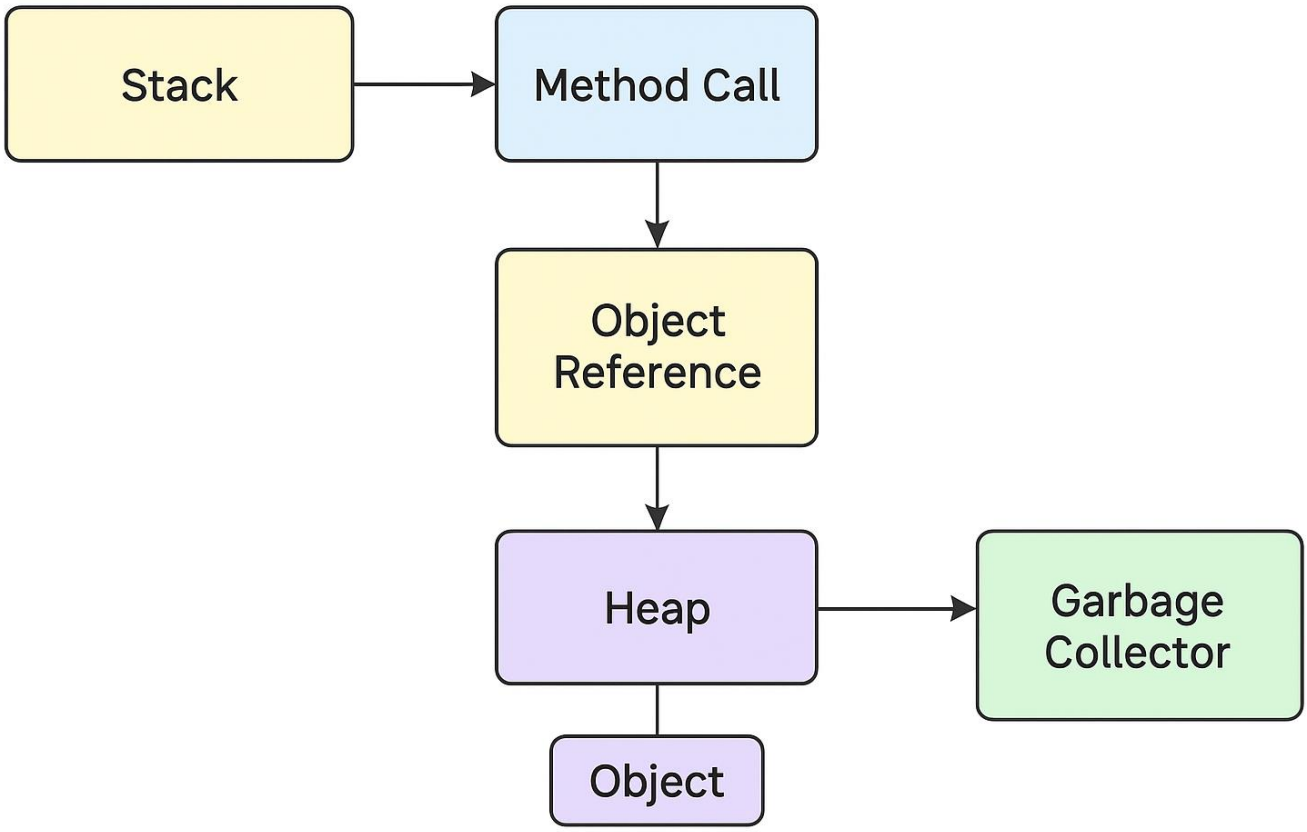
> If the object in Heap has no reference in Stack,

Java's Garbage Collector automatically removes it.

✅ In short (Interview Line):

> "Stack stores method calls and references, Heap stores actual objects.

When reference is gone, the object in Heap is garbage collected."



🔹 Step-by-step Flow (according to the diagram)

STEP 1: Stack → Method Call

- When program starts, JVM calls main()
- A **stack frame** is created for main()

STEP 2: Stack → Object Reference

- Inside main(), you write:
- Demo d = new Demo();

- d (reference) is stored in the **Stack**

STEP 3: Heap → Object

- new Demo() creates an object in the **Heap**
- That object contains:
    - int x = 10;

STEP 4: Reference connects Stack → Heap

- The reference d in Stack **points to** the object in Heap.

So the flow is:
main() → d → Heap Object

☆ Interview One-Liner

Stack stores method calls and reference variables; Heap stores real objects.
If no reference points to a Heap object, it is removed by Garbage Collector.

□ Your understanding (simplified from your message):

1. The main() method and other methods are stored in the stack

2. When a method runs, operations happen inside the stack frame

3. When we create an object, it is stored in the heap memory

4. The reference of that object is stored in the stack

5. The instance variables and data of that object are stored in the heap

6. If the object is removed from heap (or reference becomes null), the reference in stack is useless (and object will be garbage collected)

✅ In short — your concept is perfectly right:

> Stack → methods + local variables + object references

Heap → actual objects + instance variables

If the reference in stack is gone → object in heap becomes eligible for Garbage Collection

1. Why static keyword is used in Java?

> The static keyword means — “belongs to the class, not to any object.”

So you don’t need to create an object to use it.

✅ Use:

To save memory and to make things common/shared for all objects.

⚙️ 2. Where we can use static:

You can apply static to:

Variables → static variable

Methods → static method

Blocks → static block

Inner classes → static nested class

3. Example:

```
class Student {
   static String college = "ABC College"; // static variable
   int id;
   String name;
   static void displayCollege() {        // static method
      System.out.println("College: " + college);
   }
   void showDetails() {
      System.out.println(id + " " + name);
   }
}
```

💬 4. Explanation for interview:

Static variable:

Shared by all objects. Memory created only once (class-level).

Student.college

Static method:

Can be called without creating object.

Student.displayCollege();

Static block:

Runs once when the class is loaded — before main() or object creation.

⚡ 5. Why main() must be static?

> Because when Java starts your program, no object is created yet.

The JVM needs a starting point — it directly calls main() without creating an object.

So:

public static void main(String[] args)

public → JVM can access it

static → no object needed to call it

void → doesn't return anything

main → entry point

✅ 6. In simple words (for quick answer):

> static means class-level.

It loads first before any object.

It helps share common data and call methods without creating objects.

That's why main() is static — so JVM can run program directly.

1. What is a method in Java?

> A method is a block of code that performs a specific task.

It helps to reuse code, organize logic, and avoid repetition.

Example:

```
void display() {
   System.out.println("Hello Harika!");
}
```

So yes — without methods, we can't organize or reuse code properly.

That's why methods are core part of Java programming.

⚙️ 2. Two types of methods:

Type	Belongs To	How to Call	Need Object?	Example
Static Method	Class	ClassName.methodName()	❌ No	Test.show()
Non-Static Method	Object	objectName.methodName()	✅ Yes	obj.display()

🧠 3. Example:

```
class Example {
   static void staticMethod() {
      System.out.println("I am static method");
   }
   void nonStaticMethod() {
      System.out.println("I am non-static method");
   }
   public static void main(String[] args) {
      staticMethod(); // ✅ Works directly (no object)

      Example ex = new Example();
      ex.nonStaticMethod(); // ✅ Needs object
```

```
    }
}
```

4. Main difference (Interview Point):

Feature	Static Method	Non-Static Method
Belongs To	Class	Object
Memory	Created once when the class is loaded	Created for each object
Access	Cannot access instance variables directly	Can access both static and instance variables
Call	ClassName.method()	object.method()
Example	Math.sqrt()	String.length()

💬 5. Why both are needed?

Static methods → for common tasks (utility work, like Math.max() or main()).

Non-static methods → for object-specific behavior (each object's data may differ).

Example:

Every student has different marks → showMarks() must be non-static.

But calculation utility → can be static.

✅ In short:

> Methods = block of code to perform task.

Static → belongs to class, no object needed.

Non-static → belongs to object, needs object to call.

We need both: static for general work, non-static for object behavior.

> **Method Overloading** means defining multiple methods with the same name in the same class but with different parameter lists (number, type, or order of parameters).

⚙ Example:

```
class Calculator {
    // 1. add with two integers
    void add(int a, int b) {
        System.out.println("Sum: " + (a + b));
    }
    // 2. add with three integers
    void add(int a, int b, int c) {
        System.out.println("Sum: " + (a + b + c));
    }
    // 3. add with double values
    void add(double a, double b) {
        System.out.println("Sum: " + (a + b));
    }
    public static void main(String[] args) {
        Calculator c = new Calculator();
        c.add(10, 20);       // calls first
        c.add(10, 20, 30);   // calls second
        c.add(2.5, 3.5);     // calls third
    }
}
```

✅ Output:

```
Sum: 30
Sum: 60
Sum: 6.0
```

💬 Key Points for Interview:

Same method name

Different parameter number or type or order

Return type doesn't matter (it alone can't overload a method)

Decided at compile time → so it's a compile-time polymorphism

❓ Example of invalid case:

void display(int a) {}

int display(int a) { return a; } // ❌ Invalid - only return type differs

🗣️ Short interview line:

> "Method overloading is compile-time polymorphism where the same method name is used with different parameter lists to perform similar tasks in different ways."

OOPS in Java is a programming approach based on objects that follows four principles: Encapsulation, Inheritance, Polymorphism, and Abstraction to make code reusable, secure, and maintainable.

Encapsulation — Simple Definition

> Encapsulation is binding data (variables) and methods that operate on that data into a single unit (class).

We make variables private and access them using getter and setter methods.

What is the main purpose of Encapsulation?

Encapsulation means *hiding internal data and implementation* and allowing access only through controlled methods.

ATM Example (Very Easy to Understand)

When you use an **ATM**:

- You **don't see**:
  - Your balance variable
  - PIN verification logic
  - Cash deduction logic
- You **only use buttons**:
  - Withdraw
  - Check Balance
  - Deposit

👉 This is **encapsulation**.

🔑 How it works in Java

- **Variables** → private (hidden)
- **Methods** → public (controlled access)

```
class ATM {
   private int balance = 10000;   // hidden data
   public void withdraw(int amount) {
      if (amount <= balance) {
         balance -= amount;
         System.out.println("Withdraw successful");
      } else {
         System.out.println("Insufficient balance");
      }
   }
}
```

👉 **User cannot directly change balance**
👉 **User must use withdraw() method**

🎯 Main Purposes of Encapsulation

1. Data Hiding
   Prevents direct access to variables (like ATM balance)
2. Security
   No one can misuse or change data directly
3. Control Over Data
   Rules can be applied (e.g., minimum balance)
4. Code Maintainability
   Internal logic can change without affecting users

💻 How to Generate Getters and Setters Automatically

🔷 In Eclipse:

1. Right-click inside your class file.
2. Choose Source → Generate Getters and Setters..
3. Select the variables → click Generate ✅

👉 Shortcut:

Press Alt + Shift + S, then choose R (Generate Getters and Setters).

🔷 In IntelliJ IDEA:

1. Place your cursor inside the class.
2. Press Alt + Insert (Windows/Linux) or Cmd + N (Mac).
3. Choose Getter and Setter.
4. Select variables → click OK

🔒 Why We Use Encapsulation (Real-Time Use):

To protect data (no direct access to variables).

To control how values are set or retrieved.

To maintain consistency in large applications.

It's the base for JavaBeans used in frameworks like Spring and AEM models.

✅ In short:

> Encapsulation = Data hiding + Control access.

Private variables + public getters & setters.

In IDEs, you can auto-generate them (Eclipse → Alt+Shift+S, IntelliJ → Alt+Insert).

🔷 WHAT is a Constructor?

A constructor is a special method in Java that is used to initialize an object.

Key points:

- Name is **same as class name**
- **No return type** (not even void)
- Automatically called when **new keyword** is used
- Used to initialize **instance variables**

✔ Example:

```
Constructor e = new Constructor(101, "Harika");
```

🔷 WHY do we need a Constructor?

We need constructors to:

✔ Initialize object data at creation time
✔ Ensure object is created in a **valid state**
✔ Avoid multiple setter calls
✔ Improve **readability & safety**

❌ Without constructor:

```
Student s = new Student();

s.name = "Harika";

s.age = 22;
```

✔ With constructor:

```
Student s = new Student("Harika", 22);
```

👉 Less code, safer object creation

🔷 WHEN is a Constructor called?

A constructor is called automatically when an object is created using the new keyword.

✔ Example:

```
Student s = new Student();
```

→ Constructor runs immediately.

🔷 HOW does a Constructor work?

1. JVM allocates memory for the object
2. Constructor is called automatically
3. Instance variables are initialized
4. Object reference is returned

🔷 TYPES of Constructors

Type	Meaning	Example
Default Constructor	Provided by JVM if no constructor is written	Student()
No-arg Constructor	Manually written constructor with no parameters	Student(){}
Parameterized Constructor	Accepts parameters to initialize values	Student(String n, int a)

📌 Important rule:

If **any constructor is written**, JVM **will not** provide default constructor.

◆ COMPLETE EXAMPLE (All types)

Constructor.java

```java
package Basics;

public class Constructor {
    int id;
    String name;
    // Default constructor
    Constructor() {
    }
    // Parameterized constructor
    Constructor(int id, String name) {
        this.id = id;
        this.name = name;
    }
    void display() {
        System.out.println(id + " " + name);
    }
    @Override
    public String toString() {
        return "Constructor{id=" + id + ", name='" + name + "'}";
    }
    public static void main(String[] args) {
        Constructor e1 = new Constructor();//value passing
        e1.id = 101;
        e1.name = "Harika";
        Constructor e2 = new Constructor(102, "John");//constrtor all at once value passing
        e1.display();
        e2.display();
    }
```

◆ BONUS POINTS (Interview)

- ✔ Constructors **cannot be inherited**
- ✔ Constructors **can be overloaded**
- ✔ this keyword refers to current object
- ✔ Constructor is **not a method**

□ INTERVIEW SUMMARY (Memorize this)

A constructor is a special method in Java used to initialize objects. It has the same name as the class and no return type. Constructors are automatically called when the new keyword is used. Java provides a default constructor if none is defined, and we can also create no-arg and parameterized constructors to initialize values efficiently.

□ What is this keyword in Java?

> The this keyword is a reference variable in Java that refers to the current object of the class.

💬 Interview Definition:

> this keyword is used to refer to the current class instance variable, method, or constructor.

It helps Java understand which variable or method we are referring to inside the same class.

⚙ Why we use this keyword? (Main reason)

Sometimes local variables (method parameters) have the same name as instance variables.

Without this, Java gets confused which one to use — local or instance variable.

To avoid this naming conflict, we use this to refer to the instance variable.

📓 Example (Without this)

```
class Student {
   private String name;
   public void setName(String name) {
      name = name; // ✕ assigns parameter to itself, not instance variable
   }
   public void show() {
      System.out.println(name); // null
   }
}
```

→ Here, name = name; doesn't work as expected — both refer to the local variable.

✅ Correct version (Using this)

```
class Student {
    private String name;
    public void setName(String name) {
        this.name = name;  // ✅ this refers to instance variable
    }
    public void show() {
        System.out.println(name); // prints actual value
    }
    public static void main(String[] args) {
        Student s = new Student();
        s.setName("Hari");
        s.show(); // Output: Hari
    }
}
```

💬 So in interview you can say:

> When local and instance variables have the same name, Java gives priority to local variables.

To tell Java that we're referring to the instance variable, we use the this keyword.

⚡ Other uses of this:

1. To call current class methods

→ this.show();

2. To call current class constructors

→ this();

3. To pass current object as a parameter

→ method(this);

4. To return the current object

→ return this;

✅ In short (for interview):

> this = current object reference

Solves conflict between instance and local variables

Used to access current class members

1. What is Inheritance in Java?

> Inheritance means one class (child) can reuse the properties and methods of another class (parent).

✅ It allows **code reusability** and helps us follow OOP principles.

📓 Example:

```
class Parent {
    void display() {
        System.out.println("I am parent class");
    }
}
class Child extends Parent {//extends keyword to call parent class.
    void show() {
        System.out.println("I am child class");
    }
    public static void main(String[] args) {
        Child c = new Child();
        c.display(); // from Parent
        c.show();    // from Child
    }
}
```

✅ Output:

```
I am parent class
I am child class
```

⚙ 2. Why We Need Inheritance

To reuse code (no need to rewrite methods again)

To reduce duplication

To build a hierarchy (Parent → Child → GrandChild)

To make code easier to maintain

◈ 3. Types of Inheritance in Java

Type	Description	Example
Single Inheritance	One parent → one child	A → B
Multilevel Inheritance	Parent → Child → Grandchild	A → B → C
Hierarchical Inheritance	One parent → multiple children	A → B, A → C
Multiple Inheritance	Two parents → one child	✕ Not allowed with classes (diamond problem)

📖 Example: Single & Multilevel

```
// Single
class A { void msgA() { System.out.println("From A"); } }
class B extends A { void msgB() { System.out.println("From B"); } }
// Multilevel
class C extends B { void msgC() { System.out.println("From C"); } }
class Main {
   public static void main(String[] args) {
      C obj = new C();
      obj.msgA();
      obj.msgB();
      obj.msgC();
   }
```

}

✅ Output:

From A

From B

From C

⚠️ 4. Why Multiple Inheritance Not Allowed

If two parent classes have the same method, Java gets confused which one to call — this is known as the Diamond Problem.

That's why Java doesn't allow multiple inheritance with classes —

but it allows it with interfaces (since they only contain method declarations).

📘 Example (Diamond Problem)

```
class A { void show() { System.out.println("From A"); } }
class B { void show() { System.out.println("From B"); } }
// class C extends A, B ❌ Not allowed in Java
```

✅ Java solves this by using interfaces:

```
interface A { void show(); }
interface B { void show(); }
class C implements A, B {
    public void show() { System.out.println("Resolved!"); }
}
```

💬 5. Interview Answer (Perfect Summary):

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

It helps in code reusability and reduces duplication.

Java supports single, multilevel, and hierarchical inheritance, but not multiple inheritance with classes (to avoid the diamond problem).

Multiple inheritance is possible using interfaces.”

✅ In short:

> Inheritance = getting data/methods from parent class → child class

extends keyword is used

Multiple inheritance ❌ (to avoid confusion)

Through interface ✔ (allowed)

✳️ this vs super in Java — Interview Explanation

Both this and super are keywords in Java used to refer to objects —

but they refer to different objects.

Keyword	Refers To	Used In	Purpose
this	Current class object	Same class	Access current class variables, methods, or constructors
super	Parent class object	Subclass	Access parent class variables, methods, or constructors

1️⃣ this keyword

> Refers to the current class object — used when class variables and parameters have the same name.

📓 Example:

```
class Student {
   int id;
   String name;
   Student(int id, String name) {
      this.id = id;      // refers to current class variable
      this.name = name;  // avoids confusion
   }
   void display() {
      System.out.println(id + " " + name);
   }
   public static void main(String[] args) {
      Student s = new Student(101, "Harika");
      s.display();
   }
}
```

✅ Output:

101 Harika

👉 Why needed:

Without this, Java would get confused between local variable (id) and instance variable (id).

this clears the confusion.

2️⃣ super keyword

> Refers to the parent class object — used to access parent class methods, variables, or constructors.

📓 Example:

```
class Parent {
   int a = 10;
   void show() {
      System.out.println("Parent show method");
   }
}
class Child extends Parent {
   int a = 20;
   void show() {
      System.out.println("Child show method");
   }
   void display() {
      System.out.println(a);       // Child variable
      System.out.println(super.a);  // Parent variable
      show();                    // Child method
      super.show();              // Parent method
   }
```

```
    public static void main(String[] args) {
        Child c = new Child();
        c.display();
    }
}
```

✅ Output:

```
20
10
Child show method
Parent show method
```

3. super() and this() in constructor.

this() → calls current class constructor

super() → calls parent class constructor

📓 Example:

```
class A {
    A() {
        System.out.println("Parent Constructor");
    }
}
class B extends A {
    B() {
        super(); // calls A's constructor
        System.out.println("Child Constructor");
    }
    public static void main(String[] args) {
        new B();
    }
}
```

✅ Output:

Parent Constructor

Child Constructor

🧠💬 **Interview-ready Answer:**

> this refers to the current class object, while super refers to the parent class object.

We use this to access current class variables, methods, or constructors.

We use super to access parent class variables, methods, or constructors.

Example — when child and parent have the same variable or method name, super helps to differentiate them.

⚡ Quick Difference Table

Feature	this	super
Refers to	Current class	Parent class
Used for	Access current class variables/methods	Access parent class variables/methods
Constructor call	Calls current class constructor	Calls parent class constructor
Used in	Same class	Subclass only
Common use	Avoid variable shadowing	Access overridden members

✳ Casting in Java (Simple Overview)

◈ What is Casting?

Casting means converting one type into another type.

In Java, we have:

1. Primitive casting (int → double)
2. Reference casting (Object → Object)
   ⊡ This includes Upcasting & Downcasting

▲ 1 ☐Upcasting (Simple)

◈ WHAT is Upcasting?

Upcasting means converting a child object into a parent class reference.

✔ Happens automatically
✔ Safe
✔ Used for runtime polymorphism

UpCasting.java | Downcasting.java

```java
package Basics;

// Parent class
class Parent {
    void show() {
        System.out.println("Parent show");
    }
}
// Child class
class Child extends Parent {
    void show() {
        System.out.println("Child show");
    }

    void childMethod() {
        System.out.println("Child specific method");
    }
}
// Upcasting example
public class UpCasting{
    public static void main(String[] args) {
        // Upcasting: Parent reference pointing to Child object
        Parent p = new Child();
        // Calls overridden method in Child class (runtime polymorphism)
        p.show();   // Output: Child show
        // p.childMethod();
        // ❌ Compile-time error: Parent reference cannot call Child-specific methods
    }
}
```

🔷 WHY Upcasting?

- ✔ To achieve runtime polymorphism
- ✔ To write flexible code
- ✔ Used in frameworks (Spring, AEM, Hibernate)

🔷 KEY POINTS (Remember)

- Done **implicitly**
- Parent reference → Child object
- Only parent methods accessible
- Overridden child methods are executed

▼ 2 ☐Downcasting (Simple)

◈ WHAT is Downcasting?

Downcasting means converting a parent reference back into a child reference.

✔ Done **manually**
✔ Used to access **child-specific methods**

◈ HOW (Example)

```
Parent p = new Child();   // Upcasting
Child c = (Child) p;      // Downcasting
c.onlyChildMethod();
```

✕ Dangerous Case

```
Parent p = new Parent();
Child c = (Child) p;   // Runtime error
```

✕ **ClassCastException**

☑ SAFE Downcasting (Best Practice)

```
if (p instanceof Child) {
    Child c = (Child) p;
    c.onlyChildMethod();
}
```

✔ Prevents runtime error

◈ WHY Downcasting?

✔ To access **child-specific features**
✔ When framework returns **parent reference**

UpCasting.java | Downcasting.java

```java
package Basics;

// Parent class
class Parent1 {
    void show() {
        System.out.println("Parent show");
    }
}
// Child class
class Child1 extends Parent1{
    void show() {
        System.out.println("Child show");
    }

    void childMethod() {
        System.out.println("Child specific method");
    }
}
// Downcasting example
public class Downcasting {
    public static void main(String[] args) {
        // Upcasting first (Parent reference pointing to Child object)
        Parent p = new Child();
        // Downcasting: Convert Parent reference back to Child reference
        if (p instanceof Child) {  // Safety check
            Child c = (Child) p;
            c.show();          // Calls Child's show()
            c.childMethod();   // Access Child-specific method
        }
```

🔄 Summary Table (Easy)

Casting	Meaning	Automatic?	Safe?
Upcasting	Child → Parent	Yes	✅
Downcasting	Parent → Child	No	⚠ use instanceof

🧠 Interview One-Line Answer

Upcasting converts a child object to a parent reference and is used for runtime polymorphism. Downcasting converts a parent reference to a child reference and is used to access child-specific methods, and it must be done carefully using instanceof.

> Method Overriding occurs when a child class provides its own implementation of a method already defined in the parent class with same name, same parameters, and same return type.

 Example:

```
class Parent {
    void show() {
        System.out.println("Parent show()");
    }
}
class Child extends Parent {
    @Override
    void show() {
        System.out.println("Child show()");
    }
}
public class Test {
    public static void main(String[] args) {
        Parent obj = new Child();  // upcasting
        obj.show();  // calls Child's version — runtime polymorphism
    }
}
```

✅ Output:

```
Child show()
```

 Key Points (Interview Notes)

Feature	Description
Purpose	Change or redefine the behavior of a parent class method in the child class
Rules	Same method name, same parameters, and same return type
Access Modifier	Cannot be more restrictive than the parent method
Annotation	Use @Override to avoid mistakes
Type	Happens at runtime (Runtime Polymorphism)
Object Used	Parent reference → Child object
Dynamic Dispatch	Java decides at runtime which method to call based on the object type, not the reference type

⚙ Real-time Example:

```
class Bank {
    double getRateOfInterest() {
        return 5.0;
    }
}
class SBI extends Bank {
    @Override
    double getRateOfInterest() {
        return 7.0;
    }
}
```

```
class HDFC extends Bank {
    @Override
    double getRateOfInterest() {
        return 6.5;
    }
}
public class Main {
    public static void main(String[] args) {
        Bank b1 = new SBI();
        Bank b2 = new HDFC();
        System.out.println("SBI Rate: " + b1.getRateOfInterest());
        System.out.println("HDFC Rate: " + b2.getRateOfInterest());
    }
}
```

✅ Output:

SBI Rate: 7.0

HDFC Rate: 6.5

👉 Both classes override the parent method to give their own behavior.

⚡ Rules for Method Overriding:

1️⃣ Method name, return type, and parameters must be exactly same.

2️⃣ Access modifier cannot be more restrictive (e.g., parent public → child can't make it private).

3️⃣ The parent method must not be final or static (final = cannot override).

4️⃣ Constructors cannot be overridden.

5️⃣ Must be inherited class (inheritance required).

⚔️ Method Overloading vs Method Overriding

Feature	Method Overloading	Method Overriding
Definition	Same method name, different parameters	Same method name and parameters in different classes (inheritance)
Type of Polymorphism	Compile-time (Static)	Runtime (Dynamic)
Class	Same class	Parent–child relationship
Parameters	Must be different	Must be same
Return Type	Can be different	Must be same or covariant
Access Modifier	Can be anything	Cannot reduce visibility
Static / Final Methods	Can overload static/final methods	Cannot override static/final methods
When Resolved	At compile time	At runtime
Example	add(int, int) and add(double, double)	show() in parent and overridden show() in child

---

□ Example Showing Both

```
class Parent {
    void display() {
        System.out.println("Parent display()");
    }
}
class Child extends Parent {
    // Overloading
    void display(String name) {
        System.out.println("Hello " + name);
    }
    // Overriding
    @Override
    void display() {
        System.out.println("Child display()");
    }
}
public class Test {
    public static void main(String[] args) {
        Child c = new Child();
        c.display();          // overridden
        c.display("Hari");  // overloaded
```

✅ Output:

```
Child display()
Hello Hari
```

💬 Interview-Ready Answer:

> Method Overloading means having multiple methods with the same name but different parameters in the same class — it's compile-time polymorphism.

Method Overriding means redefining a parent class method in a child class with the same name and parameters — it's runtime polymorphism.

We use overriding to change or extend the behavior of a parent method in the child class.

🌟 Polymorphism in Java (Interview Explanation)

Meaning:

> “Poly” means many, and “morph” means forms.

So, Polymorphism means one thing showing many forms.

In Java — it allows one method, class, or object to behave differently based on the context.

💬 Simple Interview Definition:

> Polymorphism in Java means performing a single action in different ways

⚙️ Why we use it (Purpose):

To reuse code.

To write flexible and maintainable programs.

To achieve dynamic behavior — so the correct method runs automatically at runtime.

🔹 Types of Polymorphism in Java

Type	When it happens	Also called	Achieved by
Compile-time Polymorphism	During compilation	Static Polymorphism	Method Overloading
Runtime Polymorphism	During execution	Dynamic Polymorphism	Method Overriding

🔹 1️⃣ Compile-Time Polymorphism (Method Overloading)

Meaning:

Same method name but different parameter lists.

📓 Example:

```
class MathOperation {
    void add(int a, int b) {
        System.out.println("Sum: " + (a + b));
    }
    void add(double a, double b) {
        System.out.println("Sum: " + (a + b));
    }
}

public class Demo {
    public static void main(String[] args) {
        MathOperation obj = new MathOperation();
        obj.add(10, 20);      // calls int version
        obj.add(2.5, 3.5);    // calls double version
    }
}
```

✅ Output:

```
Sum: 30
Sum: 6.0
```

👉 Compile-time decides which method to call based on arguments.

2️⃣ Runtime Polymorphism (Method Overriding)

Meaning:

When a child class provides a specific implementation of a method already defined in the parent class.

📓 Example:

```
class Animal {
   void sound() {
      System.out.println("Animal makes sound");
   }
}
class Dog extends Animal {
   void sound() {
      System.out.println("Dog barks");
   }
}
class Cat extends Animal {
   void sound() {
      System.out.println("Cat meows");
   }
}
public class Test {
   public static void main(String[] args) {
      Animal a;      // reference variable of parent class
      a = new Dog();  // Dog object
      a.sound();     // Dog's sound()
      a = new Cat();  // Cat object
```

```
        a.sound();      // Cat's sound()
    }
}
```

✅ Output:

Dog barks

Cat meows

👉 Here, which method runs is decided at runtime, not compile time.

This is called Dynamic Method Dispatch.

⚡ Dynamic Method Dispatch

> Dynamic Method Dispatch is the process where a call to an overridden method is resolved at runtime rather than at compile time.

□ In short:

It allows Java to select the right method based on the actual object type, not the reference type.

🧠💬 Interview-ready Answer:

> Polymorphism in Java allows one action to be performed in different ways.

It helps achieve flexibility and dynamic behavior in programs.

There are two types — compile-time (method overloading) and runtime (method overriding).

Runtime polymorphism is achieved using method overriding and dynamic method dispatch.

Why Wrapper Classes in Java?

💡 **Reason:**

Java is not 100% object-oriented, because it has primitive data types (int, char, boolean, etc.) that are not objects.

To make these primitives work like objects, Java provides Wrapper Classes — so that even basic values can be used as objects.

 Example:

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

 Simple Example:

```
int a = 10;                 // primitive
Integer num = Integer.valueOf(a);  // wrapping primitive into object
System.out.println(num);
```

✅ Output:

```
10
```

Now num is an object, not a primitive — this helps Java behave more object-oriented.

⚙️ AutoBoxing and UnBoxing (Java 5 feature)

AutoBoxing: Automatic conversion of primitive → Wrapper

UnBoxing: Automatic conversion of Wrapper → primitive

🔷 Example:

```
int a = 5;        // primitive
Integer b = a;    // AutoBoxing
int c = b;        // UnBoxing
System.out.println(b + c); // Works fine
```

✅ Output:

```
10
```

💬 In Interview — How to Answer

> Java is not 100% object-oriented because it supports primitive data types which are not objects.

To treat primitives as objects, Java provides Wrapper Classes (like Integer, Double, Boolean).

Wrapper classes help when we need to store data in collections (like ArrayList, HashMap) because collections can only store objects — not primitives.

🧠 Example with Collections:

```
ArrayList<Integer> list = new ArrayList<>();
list.add(10);  // AutoBoxing — primitive int to Integer
list.add(20);
System.out.println(list);
```

✅ Output:

```
[10, 20]
```

🎯 In short (Interview Quick Answer):

> Java provides Wrapper Classes to convert primitive types into objects.

This is needed because Java aims to be object-oriented, and collections or frameworks (like JDBC, Hibernate) work with objects, not primitives.

Wrapper classes also help in autoboxing, unboxing, and using utility methods (like Integer.parseInt()).

💥 Object Class in Java — Overview

In Java,

> Every class implicitly inherits from the Object class, which is the parent of all classes in Java.

That means — even if you don't write extends Object, Java automatically adds it.

 Example:

```
class Student {
   int id;
   String name;
}
```

👉 Internally:

```
class Student extends Object {
   int id;
   String name;
}
```

✅ So every class in Java gets Object class methods like:

equals()

hashCode()

toString()

getClass()

clone()

finalize()

wait(), notify(), notifyAll() (used in threads)

---

1. equals() method

📓 Definition:

> Used to compare two objects for equality.

By default (from Object class), it compares memory addresses (references), not values.

🔹 Example 1: Default behavior

```
class Test {
    public static void main(String[] args) {
        String s1 = new String("Harika");
        String s2 = new String("Harika");
        System.out.println(s1 == s2);       // false (different memory)
        System.out.println(s1.equals(s2));  // true (String class overrides equals())
    }
}
```

✅ Output:

```
false
true
```

👉 String class overrides equals() to compare content, not memory.

👉 In your own classes, if you want to compare object values, you must override equals().

 Example 2: Custom class override

```
class Student {
    int id;
    String name;
    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null || getClass() != obj.getClass())
            return false;
        Student s = (Student) obj;
        return id == s.id && name.equals(s.name);
    }
}
```

✅ Now two students with same id and name are treated as equal.

2. hashCode() method

📓 Definition:

> Returns an integer hash value representing the object — used for hash-based collections like HashMap, HashSet, Hashtable.

If you override equals(), you must also override hashCode()

→ otherwise the object may behave wrongly in collections.

🔹 Example:

```
@Override
public int hashCode() {
    return Objects.hash(id, name);
}
```

✅ Both methods must be consistent:

If two objects are equal, their hashCodes must be same.

If hashCodes are same, they may or may not be equal.

⚠️ Common Interview Question

> ❓ Why should we override both equals() and hashCode() together?

✅ Answer: Because HashMap and HashSet use hashCode to find a bucket and equals to compare actual objects.

If only equals() is overridden, the collection may fail to locate the correct object.

3️⃣ toString() method

📓 Definition

> Returns a string representation of the object.

By default:

System.out.println(obj);

gives something like:

Student@5a39699c

If you override toString():

```
@Override
public String toString() {
    return "Student{id=" + id + ", name=" + name + "}";
}
```

✅ Output becomes readable:

Student{id=101, name=Harika}

4️⃣ getClass()

Returns the runtime class of the object.

```
Student s = new Student(1, "Harika");
System.out.println(s.getClass().getName());
```

✅ Output:

Student

💬 Interview-ready Summary Answer:

> Every class in Java extends the Object class directly or indirectly.

The Object class provides basic methods like equals(), hashCode(), toString(), and getClass().

equals() → compares objects for equality

hashCode() → returns hash value for object

toString() → returns string representation

getClass() → returns runtime class

We override these methods to compare values, print readable output, and ensure correct behavior in collections.

□ Bonus Tip: Dynamic Dispatch Relation

When we override equals(), toString(), or hashCode() in a subclass and call them using a parent reference — Java uses runtime polymorphism (dynamic method dispatch) to call the subclass version.

Overview

> The final keyword in Java is used to restrict modification.

It can be applied to variables, methods, and classes — and each has a different meaning.

1️⃣ final variable

📓 Meaning:

> A final variable's value cannot be changed once assigned.

In other words — constant value.

✅ Example:

```
final int speedLimit = 90;

speedLimit = 100;   // ❌ Error – cannot change value
```

⚠️ If the variable is declared final:

It must be initialized only once.

If it's not initialized during declaration, it must be done in the constructor.

✅ Example:

```
class Car {
   final int wheels;
   Car() {
      wheels = 4; // allowed, only once
   }
}
```

👉 Interview Tip:

> Use final when you want a variable to act as a constant (fixed value).

Usually combined with static → public static final (like PI value in Math class).

2️⃣ final method

📓 Meaning:

> A final method cannot be overridden by subclasses.

✅ Example:

```
class Parent {
   final void show() {
      System.out.println("This is a final method");
   }
}
class Child extends Parent {
   // void show() { } ❌ Not allowed — cannot override final method
}
```

👉 Interview Tip:

> We use final on methods when we want to prevent child classes from changing important logic in the parent class.

3️⃣ final class

📓 Meaning:

> A final class cannot be extended (inherited) by any other class.

✅ Example:

```
final class Vehicle {
   void run() {
      System.out.println("Running...");
   }
}

class Car extends Vehicle { // ❌ Error – cannot inherit final class
```

}

👉 Interview Tip:

> Use final class when you want to secure your class from modification.

Example: The String class in Java is final — no class can extend it.

□ Summary Table

Use with	Meaning	Effect
Variable	Constant	Value cannot be changed
Method	Locked	Cannot be overridden
Class	Sealed	Cannot be extended

💬 Interview-ready Answer:

> The final keyword in Java is used to restrict modification.

If we use final with a variable, its value becomes constant.

If we use it with a method, it cannot be overridden.

If we use it with a class, it cannot be inherited.

Example: The String class in Java is final, so no one can extend it.

⚡ Common Interview Questions

1□Can we declare a constructor as final?

→ ✕ No, because constructors are never inherited or overridden.

2□What is the difference between finally and finalize()?

final → keyword to restrict modification

finally → block in exception handling (always executes)

finalize() → method used before object destruction by GC (rarely used)

Naming conventions are standard rules that help Java developers write clean, readable, and consistent code.

They make it easier to understand the flow, purpose, and structure of the program.

1. Packages

Style: all lowercase

Purpose: organize classes logically by feature or layer

✅ Example:

com.companyname.projectname.service

com.aem.workflow.utils

2. Classes

Style: PascalCase (each word starts with capital)

Purpose: represent objects, entities, or main logic units

✅ Example:

public class StudentDetails { }

public class EmailNotificationService { }

3. Interfaces

Style: PascalCase

Purpose: represent capabilities or contracts

✅ Example:

public interface Printable { }

public interface Exportable { }

4. Methods

Style: camelCase (start lowercase, next word capitalized)

Purpose: represent actions or behaviors (usually verbs

✅ Example:

```
public void sendEmail() { }
public int calculateTotal() { }
```

5. Variables

Style: camelCase

Purpose: store data values, describe what the value represents

✅ Example:

```
String studentName;
int pageCount;
boolean isValid;
```

6. Constants

Style: ALL_UPPERCASE with underscores

Purpose: represent fixed values that do not change

✅ Example:

```
public static final String ADMIN_ROLE = "admin";
public static final int MAX_RETRIES = 3;
```

7. Enums

Style: PascalCase for enum type, UPPERCASE for values

Purpose: represent fixed set of constants

✅ Example:

```
public enum Status {
    ACTIVE,
    INACTIVE,
    PENDING
```

}

□ 8. Type Parameters (Generics)

Style: single capital letter

Purpose: represent generic types

✅ Example:

```
class Box<T> {  // T stands for Type
   private T content;
}
```

🎯 Purpose Summary

Type	Example	Style	Purpose
Package	com.aem.workflow.service	lowercase	Organize code
Class	EmailService	PascalCase	Represent object or logic
Interface	Exportable	PascalCase	Define capability
Method	sendEmail()	camelCase	Perform action
Variable	userName	camelCase	Store data
Constant	MAX_LIMIT	UPPERCASE	Fixed value
Enum	ACTIVE	UPPERCASE	Fixed set of values
Generic	T	Single letter	Type placeholder

💬 In short:

> Naming conventions make code self-explanatory.

You can understand the flow without reading comments

🎯 Main Purpose of Java Packages (Domains)

To organize your code.

To separate logic (e.g., controllers, services, models, etc.).

To avoid name conflicts between classes.

To make the project easy to understand and maintain.

 Typical Java Project (or Spring Boot) Package Domains

Package / Domain	Purpose
model	Holds data or entity classes
repository	Database operations
service	Business logic
controller	Handles user/API requests
config	App configuration classes
exception	Custom exception classes
dto	Data transfer between layers
util	Helper methods used everywhere

💬 In simple words:

Java packages (domains) are like folders that divide your project into layers — each with its own responsibility.

This helps you understand flow easily:

👉 Controller → Service → Repository → Database

What Are Access Modifiers?

Access Modifiers in Java control the visibility (scope) of classes, methods, and variables.

They define who can access which part of your code.

There are 4 main access modifiers 👇

🗂️ Access Modifiers Cheat Sheet

Modifier	Within Same Class	Within Same Package	Subclass (Different Package)	Other Packages	Usage / Meaning
public	✔ Yes	✔ Yes	✔ Yes	✔ Yes	Accessible from anywhere
protected	✔ Yes	✔ Yes	✔ Yes (inheritance only)	✘ No (without inheritance)	Same package + subclasses
default (no keyword)	✔ Yes	✔ Yes	✘ No	✘ No	Same package only
private	✔ Yes	✘ No	✘ No	✘ No	Same class only

📓 Example Code

```
package com.example.access;

public class ExampleClass {
    public int publicVar = 10;
    protected int protectedVar = 20;
    int defaultVar = 30; // default access (no modifier)
    private int privateVar = 40;
```

```
    public void showValues() {
        System.out.println("Public: " + publicVar);
        System.out.println("Protected: " + protectedVar);
        System.out.println("Default: " + defaultVar);
        System.out.println("Private: " + privateVar);
    }
}
```

 Access from Another Class (Same Package)

✅ Works for: public, protected, default

❌ Doesn’t work for: private

 Access from Subclass (Different Package)

✅ Works for: public, protected (via inheritance)

❌ **Doesn’t work for: default, private**

 Access from Non-Subclass (Different Package)

✅ Works for: public

❌ Doesn’t work for: protected, default, private

🔑 Quick Tips for Remembering

Shortcut Meaning

 public → Anywhere

 protected → Same package + subclasses

 default → Only same package

🔴 private → Only same class

□ Where to Use Which

Modifier	When to Use
public	When method or class should be used by any other code
protected	When allowing access for child classes (inheritance)
default	When used only within same package
private	When hiding internal logic from outside classes

💬 Example Real-Time Usage

```
public class Student {
   private String name;       // only inside class
   protected int rollNumber;   // subclasses can access
   public void printInfo() {   // can be used anywhere
      System.out.println("Name: " + name);
   }
}
```

□ 1. What is an Anonymous Object?

> An anonymous object is an object that is created without a reference variable.

You create it using the new keyword, but you don't store it in any reference.

📓 Example:

new Student().display();

Here:

new Student() → creates an object

.display() → immediately calls the method

No reference variable (like Student s) is used

✅ The object is used only once and then eligible for garbage collection.

⚙ 2. Normal object vs Anonymous object

Type	Example	Usage
Normal Object	Student s = new Student();s.display();	Object can be reused multiple times
Anonymous Object	new Student().display();	Used only once, then discarded

💬 3. Interview definition:

> "An anonymous object in Java is an object created using the new keyword without assigning it to any reference variable.

It's mainly used for one-time use, like calling methods immediately."

⚡ 4. When to use anonymous objects?

✅ When the object is needed only once — like:

Passing to a method temporarily

Calling one-time utility methods

In anonymous inner classes (common in event handling, AEM or GUI frameworks)

📓 Example (simple use):

```
class Demo {
   void show() {
      System.out.println("Hello from anonymous object");
   }
   public static void main(String[] args) {
      new Demo().show(); // Anonymous object
   }
}
```

✅ Output:

Hello from anonymous object

 5. Why we need it (real reason):

If we don’t need the object again,

why waste memory by creating a named reference?

So — anonymous object = memory-saving + one-time use.

🔷 About this in this context:

When we use new Demo().show(),

inside show(), the keyword this still works —

because this refers to the current anonymous object created by new Demo().

Even though the object has no name,

Java still knows which object is running that method — so this points to it internally.

✅ In short (Interview summary):

>Anonymous object means object without a name, created with new keyword and used immediately.

It’s useful for one-time operations.

Even inside it, this can be used — it refers to the current anonymous object.

1. Need of Array

👉 When we want to store multiple values of the same type in a single variable, we use an array.

Without Array ❌

int mark1 = 50;

int mark2 = 60;

int mark3 = 70;

Here, we need many variables — hard to manage.

With Array ✅

int[] marks = {50, 60, 70};

All values stored in one variable (marks) — easy to access and loop.

2. Creating an Array

🔷 Declaration and Initialization

int[] numbers = new int[3]; // creates array of size 3

numbers[0] = 10;

numbers[1] = 20;

numbers[2] = 30;

or

int[] numbers = {10, 20, 30}; // direct initialization

🔷 Access Elements

System.out.println(numbers[1]); // prints 20

🔷 Length

System.out.println(numbers.length); // prints 3

---

3. Multidimensional Array (Matrix)

Used to store data in rows and columns form.

Example: 2D Array

```
int[][] matrix = {
   {1, 2, 3},
   {4, 5, 6}
};
```

✅ Access:

```
System.out.println(matrix[0][2]); // Output: 3
```

✅ Loop:

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

4. Jagged Array

👉 A jagged array is a 2D array with different column sizes for each row.

Example:

```
int[][] jagged = new int[3][];  // 3 rows
jagged[0] = new int[2];  // row 1 → 2 elements
jagged[1] = new int[4];  // row 2 → 4 elements
jagged[2] = new int[3];  // row 3 → 3 elements
```

✅ Assigning values:

```
jagged[0][0] = 10;
jagged[0][1] = 20;
```

✅ Printing:

```
for (int[] row : jagged) {
    for (int val : row) {
        System.out.print(val + " ");
    }
    System.out.println();
}
```

 In short:

Jagged array = Array of arrays with uneven lengths

🌐 5. 3D Array

Used for storing data in 3 levels (like boxes, rows, and columns).

Example:

```
int[][][] cube = {
    { {1, 2}, {3, 4} },
    { {5, 6}, {7, 8} }

};
```

✅ Access:

```
System.out.println(cube[1][0][1]); // Output: 6
```

⚠ 6. Drawbacks of Arrays

Drawback	Description
Fixed Size	Once an array is created, its size cannot be changed
Same Data Type Only	Can store only one type of data (int, String, etc.)
No Direct Insert/Delete	Insertion or deletion is difficult because shifting is required
Memory Wastage	Unused array positions waste memory
No Built-in Functions	Must use loops manually for searching and sorting

✅ To overcome these → we use ArrayList, LinkedList, etc.

👩‍💻 7. Array of Objects

We can also store objects in an array — not just primitive types.

Example:

```
class Student {
   String name;
   Student(String name) {
      this.name = name;
   }
}
public class Main {
   public static void main(String[] args) {
      Student[] students = new Student[3];
      students[0] = new Student("Harika");
      students[1] = new Student("Ravi");
      students[2] = new Student("Kiran");
      for (Student s : students) {
         System.out.println(s.name);
```

✅ Output:

Harika

Ravi

Kiran

🔁 8. Enhanced for Loop (for-each loop)

Used to simplify looping over arrays.

Example:

```
int[] numbers = {10, 20, 30, 40};
for (int num : numbers) {
   System.out.println(num);
}
```

💡 No need for index, it automatically goes through each element.

Topic Meaning Example

Topic	Meaning	Example
Array	Stores multiple values of the same type	int[] a = {1, 2, 3};
2D Array	Data in rows and columns	int[][] m = {{1, 2}, {3, 4}};
Jagged Array	Rows with uneven number of columns	int[][] j = {{1, 2}, {3, 4, 5}};
3D Array	Data in multiple layers (box-like)	int[][][] x = {{{1, 2}}, {{3, 4}}};
Array of Objects	Stores objects of a class	Student[] s = new Student[3];
Enhanced for loop	Simplified way to iterate arrays	for (int x : a) {}
Drawbacks	Fixed size, cannot store mixed data types	—

1. abstract Keyword

The abstract keyword is used with classes and methods.

✅ Abstract Class

It cannot be directly instantiated (you cannot create an object of it).

It can have both abstract methods (without body) and normal methods (with body).

Used when you want to provide a base class for others to extend.

Example:

```
abstract class Animal {
    abstract void sound(); // abstract method (no body)
    void eat() { // normal method
        System.out.println("Eating...");
    }
}
class Dog extends Animal {
    void sound() {
        System.out.println("Bark");
    }
}
public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound();
        d.eat();
    }
```

You define something half-done (abstract), expecting subclasses to complete it.

2. Inner Class

An inner class is a class declared inside another class.

Why we use it:

To group classes logically that are used only by one outer class.

To access private members of the outer class easily.

Example:

```
class Outer {
    private String message = "Hello from Outer";
    class Inner {
        void display() {
            System.out.println(message); // can access private member
        }
    }
}
public class Main {
    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.display();
    }
}
```

Simple meaning:

Inner class = a class inside another class.

3. Anonymous Inner Class

An anonymous inner class is a class without a name.

It's used when you need to create a one-time-use class — often to override a method immediately.

Example:

```
abstract class Animal {
    abstract void sound();
}
public class Main {
    public static void main(String[] args) {
        Animal a = new Animal() {
            void sound() {
                System.out.println("Roar");
            }
        }; // semicolon after closing brace
        a.sound();
    }
}
```

Simple meaning:

Anonymous inner class = temporary class without name, created on the spot to override a method or implement an interface.

🔁 4. Abstract + Anonymous Inner Class Together

You often use anonymous inner classes with abstract classes or interfaces, because they let you define the method body immediately.

Example:

```
abstract class Greeting {
    abstract void sayHello();
}
public class Main {
    public static void main(String[] args) {
        Greeting g = new Greeting() {
            void sayHello() {
                System.out.println("Hello from Anonymous class!");
            }
        };
        g.sayHello();
    }
}
```

 Simple meaning:

You take an abstract class → directly give body to its abstract method → without creating a named subclass.

⚠ Summary Table

Concept	Meaning	Can Instantiate?	Used For
Abstract Class	Class containing abstract methods	❌ No	Base design for subclasses
Inner Class	Class defined inside another class	✅ Yes	Grouping logic & accessing outer class data
Anonymous Inner Class	Inner class without a name	✅ Yes (immediate)	One-time method override
Abstract + Anonymous	Instant implementation of abstract methods	✅ Yes	Quick or temporary implementation

1. Abstract Class Flow

[Abstract Class] → [Concrete Subclass] → [Object]

Example:

```
abstract class Animal {
   abstract void sound();
}
class Dog extends Animal {
   void sound() { System.out.println("Bark"); }
}
public class Main {
   public static void main(String[] args) {
      Animal a = new Dog(); // ✅ Allowed
      a.sound();          // Output: Bark
   }
```

📓 Flow Explanation:

Animal defines a common structure.

Dog gives the real implementation.

We can’t create an object of Animal directly, but can use it as a reference type.

2. Inner Class Flow

OuterClass

└── InnerClass (can access private members of OuterClass)

Example:

```
class Outer {
    private String message = "Hello";
    class Inner {
        void display() {
            System.out.println(message);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        Outer outer = new Outer();           // Create Outer
        Outer.Inner inner = outer.new Inner(); // Create Inner through Outer
        inner.display(); // Output: Hello
    }
}
```

Flow Explanation:

Inner class is tied to an instance of the outer class.

It can see and use the outer class’s private variables.

🕵️‍♀️ 3. Anonymous Inner Class Flow

Interface / Abstract Class

↓

[Anonymous Inner Class]

↓

[Object]

Example:

```
abstract class Greeting {
    abstract void sayHello();
}
public class Main {
    public static void main(String[] args) {
        Greeting g = new Greeting() { // anonymous inner class
            void sayHello() {
                System.out.println("Hi from anonymous class!");
            }
        };
        g.sayHello();
    }
}
```

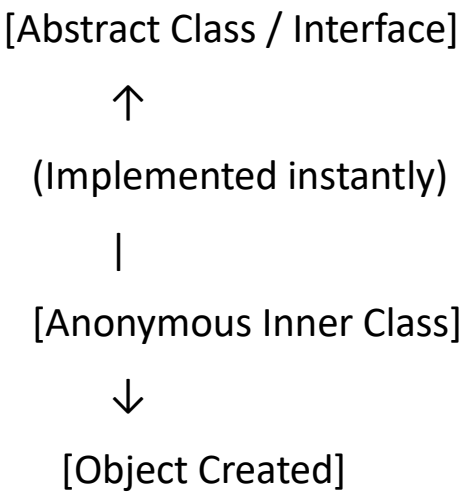
📓 Flow Explanation:

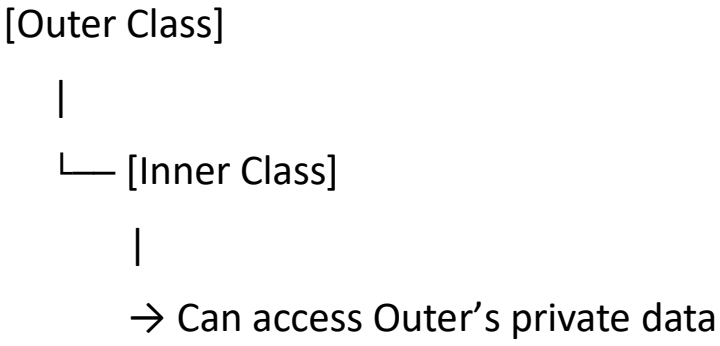
We don’t create a separate GreetingImpl class.

Instead, we create and define it immediately in one place.

This saves code when you need it only once.

Memory & Object Connection Visualization





When to Use What

Concept	Use When	Example Situation
Abstract Class	Subclasses must provide missing or specific behavior	Base class Vehicle → subclasses Car, Bike
Inner Class	Helper class is tightly coupled with the outer class	Button class → ActionListener as inner class
Anonymous Inner Class	Need a one-time implementation or method override	Quick event listener, one-time logic

1. What is an Interface?

👉 An interface in Java is like a **contract that says.**

Any class that implements me must provide these methods.

It is a completely abstract type — it only contains method declarations (no body) and constants.

🔷 Example:

```
interface Animal {
   void sound();  // abstract method
   void eat();
}
```

Now if a class implements this interface,

it must give body to all methods declared in the interface.

```
class Dog implements Animal {
   public void sound() {
      System.out.println("Bark");
   }
   public void eat() {
      System.out.println("Eating...");
   }
}


public class Main {
   public static void main(String[] args) {
      Animal a = new Dog(); // reference of interface
      a.sound();
      a.eat();
   }
}
```

✅ Output:

Bark

Eating...

💡 2. Need of Iterface (Why we use it?)

Interfaces are mainly used for abstraction and loose coupling.

🔹 Simple Meaning:

Interfaces allow you to write flexible, reusable code —

you can change implementations without changing the main logic.

🔹 Example for Need:

Imagine you have two classes:

```
class PaytmPayment { void pay() { System.out.println("Pay via Paytm"); } }
class GooglePayPayment { void pay() { System.out.println("Pay via Google Pay"); } }
```

Now your main class:

```
class PaymentService {
    void makePayment(PaytmPayment paytm) { paytm.pay(); }
}
```

⚠️ Problem:

If you want to use Google Pay tomorrow,

you have to change the PaymentService code — not good for flexibility.

✅ Better Solution: Use Interface

```
interface Payment {
    void pay();
}
class PaytmPayment implements Payment {
    public void pay() { System.out.println("Pay via Paytm"); }
}

class GooglePayPayment implements Payment {
    public void pay() { System.out.println("Pay via Google Pay"); }
}

class PaymentService {
    void makePayment(Payment payment) { // interface reference
        payment.pay();
    }
}

public class Main {
    public static void main(String[] args) {
        PaymentService ps = new PaymentService();
        ps.makePayment(new PaytmPayment());
        ps.makePayment(new GooglePayPayment());
```

✅ Output:

Pay via Paytm

Pay via Google Pay

 Main idea:

We can easily switch implementations without changing the logic —

this is called loose coupling and abstraction.

 3. Key Rules of Interfaces

Feature	Explanation
Methods	By default, public and abstract
Variables	By default, public, static, and final
Multiple Inheritance	A class can implement multiple interfaces
Cannot Instantiate	You cannot create an object of an interface directly
Default / Static Methods (Java 8+)	Interfaces can contain method bodies using default or static methods

 Example (Default & Static Methods)

```
interface Greeting {
    default void sayHello() {
        System.out.println("Hello from default method!");
    }

    static void sayHi() {
        System.out.println("Hi from static method!");
    }
}
class Welcome implements Greeting {}
public class Main {
    public static void main(String[] args) {
        Welcome w = new Welcome();
```

```
        w.sayHello();      // default method
        Greeting.sayHi();  // static method
    }
}
```

🎯 4. Interface vs Abstract Class (Quick Revision)

Feature	Interface	Abstract Class
Methods	Only abstract (till Java 7); can have default / static methods (Java 8+)	Can have both abstract and non-abstract methods
Variables	public static final only	Any type of variables
Constructor	❌ Not allowed	✅ Allowed
Multiple Inheritance	✅ Yes (implements multiple interfaces)	❌ No
Use Case	Define capability (what to do)	Define partial implementation (how partly to do)

🧠 Simple Real-World Analogy

Example	Interface Role	Implementer Role
Remote Control	Defines buttons (methods)	TV, AC, or Fan — provide actual behavior
Payment System	Defines pay() method	Paytm, Google Pay — implement payment differently
Vehicle	Defines drive() method	Car, Bike, Truck — each provides its own drive behavior

1. What is an enum?

👉 enum (short for enumeration) is a special class in Java that represents a group of constant values.

In simple words:

> Enum is used when you have a fixed set of related constants — like days, months, directions, colors, etc

🔷 Example:

```
enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

Now you can use it like this 👇

```
public class Main {
    public static void main(String[] args) {
        Day today = Day.MONDAY;
        System.out.println(today);
    }
}
```

✅ Output:

```
MONDAY
```

💡 2. Why we need Enum (Main Purpose)

Without enums, we might use:

```
int SUNDAY = 1;
int MONDAY = 2;
```

This can cause errors if we accidentally pass a wrong value (like 5 or 10).

✅ With enum, we restrict values to predefined constants only → safer and cleaner code.

🔷 3. Enum with if Statement

You can compare enums using == (not strings).

```
enum TrafficLight { RED, YELLOW, GREEN }
public class Main {
    public static void main(String[] args) {
        TrafficLight signal = TrafficLight.RED;

        if (signal == TrafficLight.RED) {
            System.out.println("STOP");
        } else if (signal == TrafficLight.YELLOW) {
            System.out.println("WAIT");
        } else {
            System.out.println("GO");
        }
    }
}
```

✅ Output:

```
STOP
```

🧠 Simple meaning:

You can use if conditions to check which enum constant it is.

🔷 4. Enum with switch Statement

Enums work very well with switch-case because each value is constant.

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }

public class Main {
    public static void main(String[] args) {
        Day today = Day.SATURDAY;

        switch (today) {
            case MONDAY:
                System.out.println("Start of the week!");
                break;
            case FRIDAY:
                System.out.println("Weekend is near!");
                break;
            case SATURDAY:
            case SUNDAY:
                System.out.println("It's weekend!");
                break;
            default:
                System.out.println("Midweek day");
        }
    }
}
```

✅ Output:

```
It's weekend!
```

□ Simple meaning:

Enums make switch statements clean and readable.

🔷 5. Enum as a Class (Enum with Variables & Methods)

Enums are actually special types of classes —
you can add fields, constructors, and methods inside them.

```
enum Color {
    RED("Stop"),
    YELLOW("Ready"),
    GREEN("Go");
    private String meaning; // variable
    // constructor
    Color(String meaning) {
        this.meaning = meaning;
    }
    public String getMeaning() {
        return meaning;
    }
}
public class Main {
    public static void main(String[] args) {
        for (Color c : Color.values()) {
            System.out.println(c + " means " + c.getMeaning());
        }
    }
}
```

✅ Output

```
RED means Stop
YELLOW means Ready
GREEN means Go
```

 Simple meaning:

Enums can behave like classes — you can store extra data and even write methods.

6. Summary Cheat Sheet

Concept	Description	Example
Enum	Group of related constants	enum Day { MON, TUE }
Use in if	Compare enum values	if (day == Day.MON)
Use in switch	Handle multiple cases easily	switch (day)
Enum class	Enum with fields and methods	Color.RED.getMeaning()
Method values()	Returns all enum constants	for (Day d : Day.values())
Method ordinal()	Returns position (index)	Day.MONDAY.ordinal()

Simple Real-Life Examples

Example	Enum Name	Values
Traffic Lights	TrafficLight	RED, YELLOW, GREEN
Payment Type	PaymentMode	CASH, CARD, UPI
Seasons	Season	SUMMER, WINTER, RAINY
Directions	Direction	NORTH, SOUTH, EAST, WEST

1. What is an Annotation?

👉 Annotation is like a note or label added to code — it gives information to the compiler or JVM about how to treat something.

It doesn't affect code logic but helps tools or frameworks understand your code.

🔷 Example:

```
@Override
void display() {
   System.out.println("Hello");
}
```

Meaning:

@Override tells the compiler:

"This method is overriding a parent class method."

If not, compiler will show an error.

💡 Common Java Annotations:

Annotation	Meaning
@Override	Indicates a method is overriding a superclass method
@Deprecated	Marks a method or class as outdated — avoid using it
@SuppressWarnings("unchecked")	Suppresses specific compiler warnings
@FunctionalInterface	Ensures the interface has exactly one abstract method

2. What is a Functional Interface?

👉 A functional interface is an interface with only one abstract method.

Simple meaning:

It defines only one behavior, and can be used with lambda expressions.

🔷 Example:

```
@FunctionalInterface
interface Calculator {
    int add(int a, int b);
}
```

Now we can use it in two ways 👇

✅ Normal Way:

```
class MyCalculator implements Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

✅ Lambda Way:

```
public class Main {
    public static void main(String[] args) {
        Calculator c = (a, b) -> a + b;
        System.out.println(c.add(5, 10));
    }
}
```

✅ Output:

```
15
```

⚡ 3. What is a Lambda Expression?

👉 Lambda expression is a short way of writing an anonymous inner class that implements a functional interface.

 Simple Meaning:

Lambda = shortcut for a single-method interface.

🔷 Syntax:

```
(parameters) -> expression or { statements }
```

🔷 Example:

```
@FunctionalInterface
interface Greeting {
    void sayHello(String name);
}
public class Main {
    public static void main(String[] args) {
        // Lambda expression
        Greeting g = (name) -> System.out.println("Hello " + name);
        g.sayHello("Hari");
    }
}
```

✅ Output:

```
Hello Hari
```

🎯 4. Lambda Expression with Return Type

If the interface method returns a value,

you can use lambda with or without return.

Example 1: Without return (single line)

```
@FunctionalInterface
interface Square {
    int calculate(int n);
}
public class Main {
    public static void main(String[] args) {
        Square s = (n) -> n * n;
        System.out.println(s.calculate(4)); // 16
    }
}
```

Example 2: With return (multi-line)

```
Square s = (n) -> {
    int result = n * n;
    return result;
};
```

✅ Both work fine!

---

5. Built-in Functional Interfaces (Java 8)

Java already gives us many ready-made functional interfaces inside java.util.function package.

Interface	Method	Example (Lambda)
Predicate<T>	boolean test(T t)	(x) -> x > 0
Function<T, R>	R apply(T t)	(x) -> x * 2
Consumer<T>	void accept(T t)	(x) -> System.out.println(x)
Supplier<T>	T get()	() -> "Hello"
BiFunction<T, U, R>	R apply(T t, U u)	(a, b) -> a + b

6. Types of Interfaces

Type	Description	Example
Normal Interface	Contains more than one abstract method	interface A { void m1(); void m2(); }
Functional Interface	Contains exactly one abstract method	@FunctionalInterface interface A { void show(); }
Marker Interface	Contains no methods; used to mark a class	Serializable, Cloneable
SAM Interface	Single Abstract Method (same as Functional Interface)	Runnable, Callable

7. Real-Time Analogy

Concept Example

Interface A contract: “You must implement this method.”

Functional Interface A simple one-method rule — “Just one behavior to define.”

Lambda Shortcut to write that one behavior quickly.

🔁 8. Summary Cheat Sheet

Concept	Description	Example
@FunctionalInterface	Ensures the interface has exactly one abstract method	@FunctionalInterface interface Test { void show(); }
Lambda Expression	Short form of an anonymous class	(a, b) -> a + b
Return Lambda	Lambda that returns a value	(n) -> n * n
Interface Types	Normal, Functional, Marker	Serializable, Runnable

👉 Collections Framework

Let’s go step-by-step in simple, clear words with examples, a hierarchy diagram, and a cheat sheet for quick revision.

🔹 1️⃣ What is the Collection API?

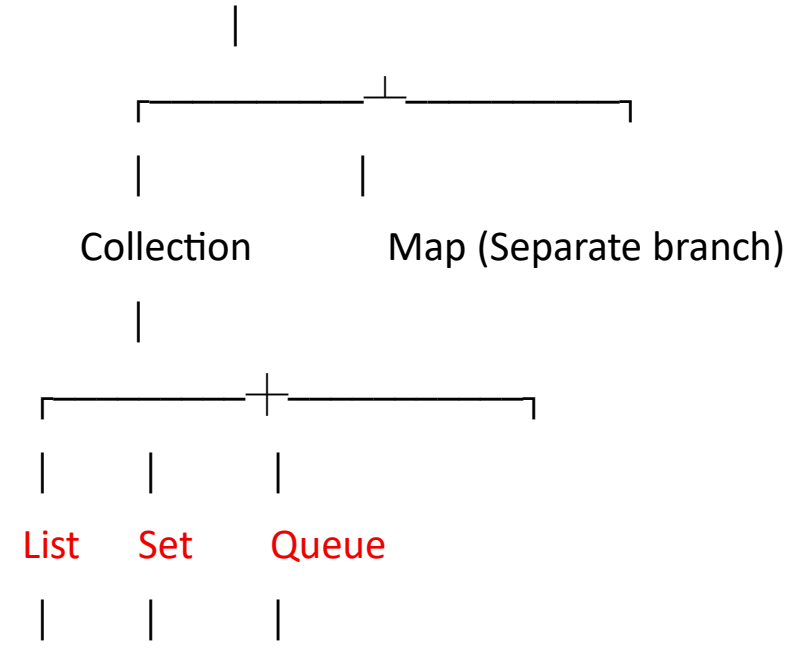
👉 The Collection Framework in Java is a set of classes and interfaces that store and manage groups of objects (like arrays but more powerful).

📄 Main goal:

To store, retrieve, and manipulate data easily.

To handle dynamic sizes (unlike arrays).

🔹 2️⃣ Collections Hierarchy (Simple Diagram)



🗺️ Map Branch

Map

|

├── HashMap

├── LinkedHashMap

└── TreeMap

🔑 3 Important Interfaces

Interface	Description	Duplicates	Order	Example
List	Stores elements in sequence	✅ Yes	✅ Maintains order	ArrayList, LinkedList
Set	Stores unique elements only	❌ No	❌ Unordered *(except LinkedHashSet)*	HashSet, TreeSet
Queue	Stores elements for processing	✅ Yes	✅ FIFO order	PriorityQueue, LinkedList
Map	Stores key–value pairs	❌ Duplicate keys	✅ Depends on type	HashMap, TreeMap

4. List Implementations

Class	Description	Example
ArrayList	Dynamic array	ArrayList<String> list = new ArrayList<>();
LinkedList	Doubly linked list	LinkedList<Integer> list = new LinkedList<>();
Vector	Synchronized ArrayList (thread-safe)	Vector<Integer> v = new Vector<>();
Stack	LIFO (Last In, First Out) stack	Stack<String> s = new Stack<>();

5. Set Implementations

Class	Description	Example
HashSet	Stores unique elements; no guaranteed order	HashSet<Integer> set = new HashSet<>();
LinkedHashSet	Stores unique elements; maintains insertion order	LinkedHashSet<String> set = new LinkedHashSet<>();
TreeSet	Stores unique elements in sorted order	TreeSet<Integer> set = new TreeSet<>();

🗺️ 6️⃣ Map Implementations

Class	Description	Example
HashMap	Stores key–value pairs; no guaranteed order	HashMap<Integer, String> map = new HashMap<>();
LinkedHashMap	Maintains insertion order	LinkedHashMap<Integer, String> map = new LinkedHashMap<>();
TreeMap	Stores entries sorted by key	TreeMap<Integer, String> map = new TreeMap<>();

⚖️ 7️⃣ Comparable vs Comparator

Feature	Comparable	Comparator
Package	java.lang	java.util
Method	compareTo()	compare()
Used for	Natural ordering	Custom ordering
Modify class?	✅ Yes	❌ No
Example	Sorting by name within the class	Sorting by marks externally

 Example:

Comparable

```
class Student implements Comparable<Student> {
    int id;
    String name;
    public int compareTo(Student s) {
        return this.id - s.id; // sort by ID
    }
}
```

Comparator

```
class NameComparator implements Comparator<Student> {
    public int compare(Student s1, Student s2) {
        return s1.name.compareTo(s2.name);
    }
}
```

Usage:

```
Collections.sort(list, new NameComparator());
```

8️⃣ Cheatsheet Summary

Keyword	Purpose	Example
extends	Class inheritance or interface extension	class A extends B
implements	Class implements an interface	class A implements Runnable
super	Refers to parent class members	super.method()
this	Refers to current object	this.var = var;

9️⃣ Collections Utility Class

👉 The Collections class provides useful static methods:

Collections.sort(list);

Collections.reverse(list);

Collections.max(list);

Collections.min(list);

Collections.frequency(list, "apple");

🏁 10️⃣ Enhanced for Loop

To loop over collections easily:

```
for(String name : list) {
   System.out.println(name);
}
```

🧠 Quick Summary Mind Map

```
Collection
 ├── List (Duplicates allowed)
|     ├── ArrayList
|     ├── LinkedList
|     ├── Vector → Stack
 ├── Set (Unique)
|     ├── HashSet
|     ├── LinkedHashSet
|     └── TreeSet (Sorted)
 ├── Queue
|     └── PriorityQueue
 └── Map (Key-Value)
       ├── HashMap
       ├── LinkedHashMap
      └── TreeMap
```

1️⃣ Collection Framework Overview

✅ Collection Framework = Interfaces + Classes + Methods

Used to store, access, and process data efficiently.

2️⃣ Hierarchy (Simple View)

Iterable

|

└── Collection

├── List

| ├── ArrayList

| ├── LinkedList

| └── Vector → Stack

├── Set

| ├── HashSet

| ├── LinkedHashSet

| └── TreeSet

└── Queue

└── PriorityQueue

Map (separate branch)

├── HashMap

├── LinkedHashMap

└── TreeMap

3️⃣ Common Methods in Collection Interface

Method	Purpose	Example
add(E e)	Adds an element	list.add("A");
addAll(Collection c)	Adds all elements from a collection	list.addAll(set);
remove(Object o)	Removes an element	list.remove("A");
clear()	Removes all elements	list.clear();
contains(Object o)	Checks if an element exists	list.contains("A");
size()	Returns the number of elements	list.size();
isEmpty()	Checks if the list is empty	list.isEmpty();
iterator()	Returns an iterator for traversal	Iterator it = list.iterator();

📜 4️⃣ List Interface (ArrayList, LinkedList, Vector, Stack)

✅ Ordered, allows duplicates.

🔷 Common Methods

Method	Description	Example
add(int index, E e)	Add element at a specific position	list.add(1, "Hello");
get(int index)	Retrieve element at an index	list.get(0);
set(int index, E e)	Replace element at an index	list.set(0, "Hi");
remove(int index)	Remove element by index	list.remove(2);
indexOf(Object o)	Returns first occurrence index	list.indexOf("A");
lastIndexOf(Object o)	Returns last occurrence index	list.lastIndexOf("A");
subList(from, to)	Returns a part of the list	list.subList(0, 3);

◆ ArrayList

Backed by a dynamic array

Fast for retrieval, slow for insert/delete in middle

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

◆ LinkedList

Backed by doubly linked list

Fast for insertion/deletion, slower for random access

LinkedList<Integer> list = new LinkedList<>();

Extra methods:

Method	Description
addFirst(E e)	Adds to beginning
addLast(E e)	Adds to end
getFirst()	Retrieves first
getLast()	Retrieves last

◆ Stack (LIFO)

Method	Description
push(E e)	Add item
pop()	Remove top item
peek()	Look at top
isEmpty()	Check empty

---

5. Set Interface

✅ No duplicates, unordered (mostly).

Method	Description
add(E e)	Adds element if not already present
remove(Object o)	Removes the specified element
contains(Object o)	Checks if element is present
clear()	Removes all elements from the set
size()	Returns the number of elements in the set

Implementations

HashSet → No order

LinkedHashSet → Maintains insertion order

TreeSet → Sorted order

Extra TreeSet methods:

Method	Description
first()	Returns the smallest element
last()	Returns the largest element
headSet(E e)	Returns elements less than e
tailSet(E e)	Returns elements greater than e

📦 6️⃣ Map Interface (Key–Value Pairs)

✅ Keys are unique, values can repeat.

Method	Description	Example
put(K, V)	Adds a key–value pair	map.put(1, "A");
get(K)	Retrieves the value for a key	map.get(1);
remove(K)	Removes the entry for a key	map.remove(1);
containsKey(K)	Checks if key exists	map.containsKey(2);
containsValue(V)	Checks if value exists	map.containsValue("A");
size()	Returns number of entries	map.size();
keySet()	Returns all keys as a set	map.keySet();
values()	Returns all values as a collection	map.values();
entrySet()	Returns key–value pairs as set	map.entrySet();

Map Implementations

Class	Order	Allows Null	Sorted?
HashMap	No order	✅ Yes	❌ No
LinkedHashMap	Maintains insertion order	✅ Yes	❌ No
TreeMap	Sorted by keys	❌ No	✅ Yes

⚖️ 7️⃣ Comparable vs Comparator

Feature	Comparable	Comparator
Package	java.lang	java.util
Method	compareTo()	compare()
Used for	Natural order	Custom order
Example	Collections.sort(list)	Collections.sort(list, comp)

🧰 8️⃣ Common Utility Methods (Collections class)

Method	Description
sort()	Sort list
reverse()	Reverse list
shuffle()	Random order
min() / max()	Smallest/largest element
frequency()	Occurrence count
unmodifiableList()	Read-only list

🔁 9️⃣ extends / implements — Quick Revision

Keyword	Used For	Example
extends	Class inherits another class **OR** interface inherits interface	class A extends B interface C extends D
implements	Class implements an interface	class A implements Runnable

-

🔟 Quick Example Mixing All Concepts

```
import java.util.*;
class Student implements Comparable<Student> {
    int id;
    String name;
    Student(int id, String name) { this.id = id; this.name = name; }
    public int compareTo(Student s) { return this.id - s.id; } // natural order
}

public class Main {
    public static void main(String[] args) {
        List<Student> list = new ArrayList<>();
        list.add(new Student(2, "Hari"));
        list.add(new Student(1, "Ravi"));

        Collections.sort(list); // uses Comparable
        for(Student s : list) System.out.println(s.id + " " + s.name);
    }
}
```

📓 11️⃣ Enhanced For Loop Example

```
for(String s : list) {
   System.out.println(s);
}
```

🎯 12️⃣ Summary — Mind Map

```
Collection
 ├── List → ArrayList, LinkedList, Vector, Stack
 |     ↳ Methods: add(), get(), set(), remove()
 ├── Set → HashSet, LinkedHashSet, TreeSet
 |     ↳ Methods: add(), contains(), size(), clear()
 ├── Queue → PriorityQueue
 |     ↳ Methods: offer(), poll(), peek()
 └── Map → HashMap, LinkedHashMap, TreeMap
       ↳ Methods: put(), get(), keySet(), entrySet()
```

🕓 Date and Time in Java

🔷 1. Old Date & Time API (Before Java 8)

Earlier, Java used:

java.util.Date

java.util.Calendar

java.text.SimpleDateFormat

✅ Example:

```
import java.util.Date;
import java.text.SimpleDateFormat;
public class OldDateExample {
    public static void main(String[] args) {
        Date date = new Date();
        SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy HH:mm:ss");
        System.out.println("Current Date: " + sdf.format(date));
    }
}
```

❌ Drawbacks:

Not thread-safe (SimpleDateFormat)

Difficult to handle time zones

Mutability (Date objects can be changed)

Confusing month indexing (0-based months)

□ Big Picture

In Java 8+, java.time is used because it is:

- ✅ Immutable
- ✅ Thread-safe
- ✅ Clear separation of date, time, timezone

👉 Interviewers mainly check:

1. Which class to use
2. Why old Date is bad
3. Basic conversions
4. Formatting

□ Main Classes You MUST Know (Core 6)

1□LocalDate – *Only Date*

📌 Use when time is NOT required

Examples in real projects

- Date of Birth
- Joining Date
- Order Date

```
LocalDate date = LocalDate.now();
LocalDate dob = LocalDate.of(2000, 5, 12);
```

🗣️ Interview line:

“LocalDate is used when only date is needed without time and timezone.”

2️⃣ LocalTime – *Only Time*

📌 Use when date is NOT required

Examples

- Store opening time
- Login time

```
LocalTime time = LocalTime.now();
```

🗣️ Interview line:

“LocalTime stores time without date.”

3️⃣ LocalDateTime – *Date + Time (No Timezone)*

📌 Most commonly used in Spring Boot applications

Examples

- CreatedAt
- UpdatedAt

```
LocalDateTime now = LocalDateTime.now();
```

🗣️ Interview line:

“LocalDateTime stores date and time but no timezone.”

4️⃣ ZonedDateTime – *Date + Time + Timezone*

📌 Use when global users are involved

```
ZonedDateTime indiaTime =
ZonedDateTime.now(ZoneId.of("Asia/Kolkata"));
```

🗣️ Interview line:

“ZonedDateTime is used when timezone is required.”

5️⃣ Instant – *Machine Time (UTC)*

📌 Used internally for timestamps, DB, conversions

```
Instant instant = Instant.now();
```

🗣️ Interview line:

"Instant represents a timestamp in UTC."

6️⃣ DateTimeFormatter – *Formatting*

📌 Used for displaying date in required format

```
DateTimeFormatter formatter =
        DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");


System.out.println(now.format(formatter));
```

🗣️ Interview line:

"DateTimeFormatter is thread-safe unlike SimpleDateFormat."

🔄 Old API vs New API (VERY IMPORTANT)

❌ Old:

Date

Calendar

SimpleDateFormat

Problems:

- Not thread-safe
- Mutable
- Confusing

✅ New:

LocalDate

LocalDateTime

ZonedDateTime

🗣️ Interview line:

“Java 8 introduced java.time to fix issues in java.util.Date.”

🔁 Conversion (Most Asked Interview Question)

🔷 Date → LocalDateTime

```
Date date = new Date();
Instant instant = date.toInstant();
LocalDateTime ldt =
    instant.atZone(ZoneId.systemDefault()).toLocalDateTime();
```

🧠 Think like:

Date → Instant → LocalDateTime

🔷 LocalDateTime → Date

```
Instant instant =
    localDateTime.atZone(ZoneId.systemDefault()).toInstant();
Date date = Date.from(instant);
```

🧠 Think like:

LocalDateTime → Instant → Date

⏳ Period vs Duration (Easy Logic)

Class	Used For
Period	Date difference (years, months, days)
Duration	Time difference (hours, minutes, seconds)

```
Period p = Period.between(start, end);
```

```
Duration d = Duration.ofHours(5);
```

🎤 Interview line:

"Period is date-based, Duration is time-based."

🎯 How to Prepare for Interview (Step-by-Step)

✅ Step 1: Remember THIS flow

LocalDate → Date only

LocalTime → Time only

LocalDateTime → Date + Time

ZonedDateTime → Date + Time + Zone

Instant → Timestamp

✅ Step 2: Practice these 4 questions

1. Why java.time over Date?
2. Difference between LocalDate and LocalDateTime?
3. How to format date?
4. Convert Date ↔ LocalDateTime

✅ Step 3: One-liner answers (Confidence booster)

- "java.time is immutable and thread-safe."
- "LocalDateTime doesn't contain timezone."
- "Instant is UTC-based timestamp."

🔷 What does Local mean in Java Date & Time?

👉 Local = No Time Zone

📌 LocalDate / LocalTime / LocalDateTime

- Uses your system's local date & time
- Does NOT store timezone
- Human-friendly (used in business logic)

Examples

```
LocalDate date = LocalDate.now();          // 2025-12-16
LocalDateTime dt = LocalDateTime.now();    // 2025-12-16T15:30
```

🗣️ Interview line:

"Local classes represent date and time without timezone information."

🔷 What is Instant?

👉 Instant = Exact point in time (UTC)

- Machine-friendly
- Stores time in **UTC**
- Used for **timestamps, DB, logs, conversions**

Example

```
Instant instant = Instant.now();
```

🗣️ Interview line:

"Instant represents a timestamp in UTC, independent of timezone."

🔁 Key Difference (Very Important)

Feature	LocalDateTime	Instant
Time zone	❌ No	✅ Yes (UTC)
Human readable	✅ Yes	❌ Not directly
Used for	Business logic	System timestamps
Example	2025-12-16 15:30	2025-12-16T10:00:00Z

🔄 How They Work Together

```
Instant instant = Instant.now();

LocalDateTime ldt =

    instant.atZone(ZoneId.systemDefault()).toLocalDateTime();
```

□ Think like:

Instant = global time

Local = local view of that time

🎯 One-Line Interview Answer

“LocalDateTime is used for local business date-time without timezone, whereas Instant represents a UTC-based timestamp.”

1. What is an Exception?

👉 An exception is an unexpected event that stops the normal flow of a program.

Simple meaning:

When something goes wrong while running (like divide by 0, file not found, wrong input).

🔷 Example:

```
public class Main {
    public static void main(String[] args) {
        int a = 10;
        int b = 0;
        int result = a / b; // ❌ Error: Divide by zero
        System.out.println(result);
    }
}
```

💥 Output:

Exception in thread "main" java.lang.ArithmeticException: / by zero

2. Exception Handling

👉 Exception Handling means writing code to handle errors gracefully — so that the program doesn't crash suddenly.

We use:

try → catch → finally

🔷 Example (Basic try–catch)

```
public class Main {
    public static void main(String[] args) {
        try {
            int a = 10 / 0; // risky code
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        }
        System.out.println("Program continues...");
    }
}
```

✅ Output:

```
Cannot divide by zero!
Program continues...
```

Simple meaning:

try holds risky code.

catch handles the error.

Program continues safely.

🔷 3. Multiple Catch Blocks

You can handle different exceptions separately.

```
public class Main {
    public static void main(String[] args) {
        try {
            int[] arr = {1, 2, 3};
            System.out.println(arr[5]); // ArrayIndexOutOfBounds
        } catch (ArithmeticException e) {
            System.out.println("Math error!");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index issue!");
        } catch (Exception e) {
            System.out.println("Something went wrong!");
        }
    }
}
```

✅ Output:

```
Array index issue!
```

 Note: Always put catch (Exception e) last,
because it is the parent of all exceptions.

---

4. Exception Hierarchy (Simple Tree)

```
Object
└── Throwable
    ├── Error (JVM issues)
    └── Exception
        ├── RuntimeException (unchecked)
        │   ├── ArithmeticException
        │   ├── NullPointerException
        │   └── ArrayIndexOutOfBoundsException
        └── IOException (checked)
            └── FileNotFoundException
```

Remember:

Checked Exceptions: handled at compile-time (e.g., IOException, SQLException).

Unchecked Exceptions: occur at runtime (e.g., NullPointerException, ArithmeticException).

⚡ 5. throw Keyword

👉 Used to manually throw an exception.

```
public class Main {
   public static void main(String[] args) {
      int age = 15;
      if (age < 18) {
         throw new ArithmeticException("Not eligible to vote!");
      }
      System.out.println("You can vote!");
   }
}
```

✅ Output:

```
Exception in thread "main" java.lang.ArithmeticException: Not eligible to vote!
```

 Simple meaning:

You use throw when you want to create an exception yourself.

⚙ 6. throws Keyword

👉 Used in method signature to pass (duck) exception handling to the caller.

```
import java.io.*;
class Test {
    void readFile() throws IOException {
        FileReader f = new FileReader("abc.txt");
    }
}
public class Main {
    public static void main(String[] args) {
        Test t = new Test();
        try {
            t.readFile();
        } catch (IOException e) {
            System.out.println("File not found!");
        }
    }
}
```

 Simple meaning:

throws means — “I won’t handle it here, caller must handle it.”

This is called exception ducking.

🔗 7. Custom Exception

👉 You can create your own exception class for specific cases.

Example:

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String msg) {
        super(msg);
    }
}
public class Main {
    static void checkAge(int age) throws InvalidAgeException {
        if (age < 18)
            throw new InvalidAgeException("Age must be 18 or above!");
        else
            System.out.println("Eligible to vote");
    }
    public static void main(String[] args) {
        try {
            checkAge(15);
        } catch (InvalidAgeException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

✅ Output:

```
Age must be 18 or above!
```

□ Simple meaning:

Custom exceptions = your own error message for special situations.

 8. Using Input with BufferReader

```
import java.io.*;
public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        System.out.print("Enter number: ");
        int num = Integer.parseInt(br.readLine());
        System.out.println("You entered: " + num);
    }
}
```

✅ Handled using throws IOException

♻️ 9. try-with-resources (Java 7+)

👉 Used to automatically close resources (like files, readers) after use.

No need to manually close them in finally.

```
import java.io.*;
public class Main {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("abc.txt"))) {
            System.out.println(br.readLine());
        } catch (IOException e) {
            System.out.println("File problem!");
        }
    }
}
```

 Simple meaning:

When try block finishes, resource (br) closes automatically.

✅ 10. Summary Cheat Sheet

Concept	Keyword	Meaning	Example
Try-Catch	try / catch	Handle errors safely	try { } catch(Exception e) { }
Multiple Catch	several catch blocks	Handle different types of errors	catch(ArithmeticException e) { }
throw	throw	Create & throw an exception manually	throw new Exception("msg");
throws	throws	Declare exception in method	void read() throws IOException
Custom Exception	User-defined	Create your own exception class	class MyException extends Exception
Try-with-resources	try(...) {}	Auto-close resources	try(FileReader fr = ...) { }
Checked Exception	Compiler-checked	Must handle or declare	IOException
Unchecked Exception	Runtime exception	Happens at runtime	NullPointerException

1. What is a Thread?

👉 A thread is a lightweight sub-process — a small part of a program that runs independently.

In simple terms:

> A thread is like a small worker inside your program doing one task, while other threads do other tasks at the same time.

🔹 Example:

```
public class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running...");
    }
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start(); // starts the thread (calls run() in new thread)
    }
}
```

🗣️ Output:

```
Thread is running...
```

2. Why We Need Threads

✅ To perform multiple tasks simultaneously

✅ To make programs faster (useful in large apps, like file downloads or games)

✅ To avoid blocking one task when another is waiting

Example: You can listen to music and browse files at the same time — those are different threads running.

3. Creating Threads (Two Ways)

Method 1: Extending Thread class

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Running thread...");
    }
}
```

Start it:

```
MyThread t = new MyThread();
t.start(); // ✅ creates new thread
```

□ Method 2: Implementing Runnable interface

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable thread running...");
    }
}
public class Main {
    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable());
        t.start();
    }
}
```

📓 Use Runnable when your class already extends another class (since Java doesn’t allow multiple inheritance).

⚙️ 4️⃣ Multiple Threads Example

```
class A extends Thread {
    public void run() {
        for(int i=1; i<=3; i++) {
            System.out.println("From A: " + i);
        }
    }
}
class B extends Thread {
    public void run() {
        for(int i=1; i<=3; i++) {
            System.out.println("From B: " + i);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        new A().start();
        new B().start()
```

🤔 Output (order may change):

```
From A: 1
From B: 1
From A: 2
From B: 2
From A: 3
From B: 3
```

⚠️ Output order may vary — threads run concurrently.

5. Thread Sleep()

Used to pause thread for some time (milliseconds):

```
try {
    Thread.sleep(1000); // sleep for 1 second
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

6. Thread Priority

Each thread has a priority (1 to 10).

Default = 5.

```
t1.setPriority(Thread.MAX_PRIORITY); // 10
t2.setPriority(Thread.MIN_PRIORITY); // 1
```

⚠ The thread with higher priority may run first, but it's not guaranteed — depends on OS scheduler.

7. Race Condition

When two or more threads access shared data at the same time and produce wrong output.

Example: Two people withdrawing money from the same bank account at once.

✅ Fix: Use synchronized keyword.

```
synchronized void withdraw(int amount) {
    balance -= amount;
}
```

🔄 8. Thread States

State Meaning

State	Description
NEW	Thread created but not started
RUNNABLE	Thread is ready to run or currently running
BLOCKED	Waiting to acquire a monitor lock (e.g., synchronized)
WAITING	Waiting indefinitely for another thread's action
TIMED_WAITING	Waiting for a specified amount of time (e.g., sleep())
TERMINATED	Thread has finished execution

🧪 Example:

```
Thread t = new Thread(() -> System.out.println("Hello"));
System.out.println(t.getState()); // NEW
t.start();
System.out.println(t.getState()); // RUNNABLE or TERMINATED
```

---

9 Thread Join()

Used to wait for one thread to finish before another starts.

t1.join(); // Wait until t1 completes

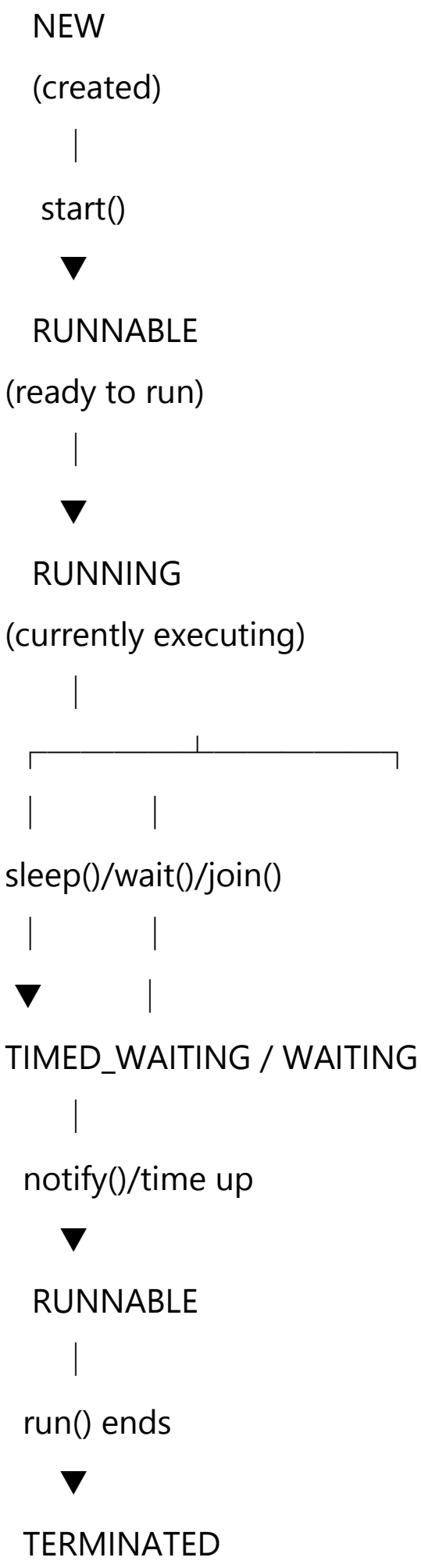
10 Try with Runnable Lambda

Java 8 simplified thread creation with lambdas:

```
Thread t = new Thread(() -> {
   System.out.println("Lambda thread running...");
});
t.start();
```

Summary Table

Concept	Meaning	Example
Thread class	Used to create threads	class MyThread extends Thread
Runnable	Interface for thread logic	implements Runnable
start()	Begins new thread execution	t.start()
run()	Contains thread code	public void run()
sleep()	Pause thread for specified time	Thread.sleep(1000)
join()	Wait for a thread to finish	t1.join()
priority	Set thread importance	setPriority(10)
synchronized	Avoid race condition / ensure thread safety	synchronized void method()


NEW
(created)
start()
RUNNABLE
(ready to run)
RUNNING
(currently executing)
sleep()/wait()/join()
TIMED_WAITING / WAITING
notify()/time up
RUNNABLE
run() ends
TERMINATED

State	When It Happens	Example / Method
NEW	Thread is created but not started	Thread t = new Thread();
RUNNABLE	After start() is called	t.start();
RUNNING	CPU is executing the thread	Managed internally by JVM
WAITING	Thread waiting for another thread	wait() or join()
TIMED_WAITING	Thread waiting for a fixed time	sleep(2000);
TERMINATED	After run() finishes	Thread completed

💡 In simple terms:

> Threads start from NEW, go to RUNNABLE, run for a while, may go to WAITING or SLEEP, and finally end in TERMINATED.