🧱 1. Java Basics

☕ What is Java?

Java is a **platform-independent, high-level, object-oriented programming language**.
It's platform-independent because Java code is compiled into **bytecode**, which runs on any system that has a **Java Virtual Machine (JVM)**.

✅ *Write once, run anywhere.*

💡 Interesting Facts About Java

- Developed by **James Gosling** at **Sun Microsystems** in **1995**.

- Originally called **Oak**, then **Green**, and finally **Java**.

🧩 4 Editions of Java

1. **Java SE (Standard Edition)** – Core Java functionality
2. **Java EE (Enterprise Edition)** – Enterprise-level applications, servlets, web apps
3. **Java ME (Micro Edition)** – Mobile and embedded systems
4. **Java Card** – Smart cards and small devices

🧠 JVM vs JRE vs JDK

Feature	JVM (Java Virtual Machine)	JRE (Java Runtime Environment)	JDK (Java Development Kit)
Purpose	Runs Java bytecode	Provides runtime environment	Develops and runs apps
Contains	Interpreter, JIT, GC	JVM + Java libraries	JRE + compiler, debugger, tools
Used for	Executing programs	Running programs	Writing & compiling programs
Developer Needs?	Comes with JRE/JDK	Comes with JDK	Must install for development

🧩 **Summary:**

- **JVM** → executes Java bytecode (platform-specific).
- **JRE** → JVM + libraries (needed to run Java apps).
- **JDK** → JRE + compiler, debugger, tools (needed to write and compile Java apps).

💾 Java Types

🟦 Primitive Types

Type	Bytes	Range	Example
byte	1	-128 to 127	`byte b = 10;`
short	2	-32,768 to 32,767	`short s = 500;`
int	4	-2B to 2B	`int i = 100000;`
long	8	Very large numbers	`long l = 10000000000L;`
float	4	~6 decimal digits	`float f = 3.14f;`
double	8	~15 decimal digits	`double d = 3.1415926535;`
char	2	Unicode characters	`char c = 'A';`
boolean	1	true / false	`boolean flag = true;`

🟩 Reference Types

Objects, arrays, and classes.

Example:

```
String name = "Irina";
int[] numbers = {1, 2, 3};
```

🔄 Control Flow

🔹 Switch Statement

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
```

```
    break;
  default:
    // code block
}
```

⚠️ Without a `break`, execution continues to the next case (“fall-through”).

🔹 Loops

1. `for`
2. `while`
3. `do...while`
4. `for-each`

Example:

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

🧰 Refactoring

Changing the **structure** of code without altering its **behavior**.

Common refactorings:

- Extract related logic into a **new method**.
- Move **repetitive code** into reusable functions.

⚙️ 2. Object-Oriented Programming (OOP)

🧩 What is OOP?

OOP stands for **Object-Oriented Programming**, a **programming paradigm** that organizes software into **objects** — each combining **data** (fields) and **behavior** (methods).

Other programming paradigms include:

- Functional Programming
- Procedural Programming
- Event-driven Programming
- Logical Programming

Some languages support only one paradigm, while others (like Java) support multiple.

🧱 4 Core OOP Concepts

Java's four fundamental OOP principles are:

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

🔹 Encapsulation

Definition: Wrapping data (fields) and methods (functions) that operate on that data into a single unit — a class.

Purpose: Hiding internal data and implementation and providing controlled access via methods.

Mechanism: Using access modifiers (`private`, `public`) to hide internal data and implementation.

Example:

```
class Person {
    private String name;  // private field

    public String getName() {
        return name;
    }

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

🔹 Abstraction

Definition: Hiding complexity by showing only essential behavior.

Purpose: Simplifies usage and reduces coupling.

Mechanism: Interfaces and abstract classes in Java

Example:

```
interface Vehicle {
    void start();
}

class Car implements Vehicle {
    public void start() {
        System.out.println("Car starting");
    }
}
```

🔹 Inheritance

Definition: Mechanism where a new class (child/subclass) acquires properties and behaviors of an existing class (parent/superclass).

Purpose: Promotes code reuse and establishes relationships.

Example:

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

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

🔹 Polymorphism

Definition: The ability of one interface to represent different underlying forms (data types). The same operation behaves differently in different contexts.

Types:

- **Compile-time polymorphism** – via **method overloading** – same method name, different parameters
- **Runtime polymorphism** – via **method overriding** – subclass provides specific implementation of a superclass method

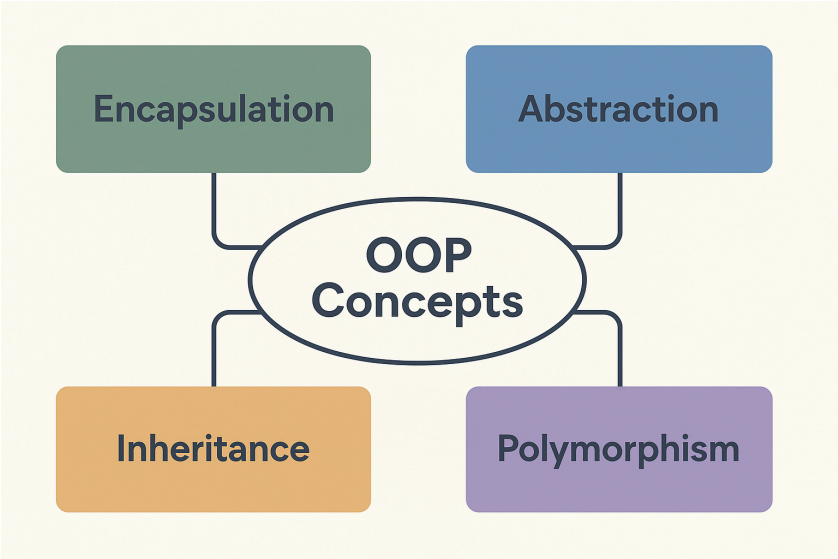
Example:

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

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

```
Animal a = new Dog();
a.sound();  // Output: Bark
```

✅ **In short:** Encapsulation, Abstraction, Inheritance, Polymorphism.



🧱 Class and Object

- **Class:** A blueprint for creating objects.
- **Object:** An instance of a class.

```
class Car {
    String color;
    void drive() { System.out.println("Driving"); }
}

Car myCar = new Car();
myCar.color = "Red";
myCar.drive();
```

🧠 Constructors in Java

Constructors are **special methods** used to initialize new objects.

Key points:

- Same name as the class
- No return type.
- Automatically called when an object is created.
- Can be overload
- A constructor without any parameters is a **default constructor**. If we don't create it, the Java compiler will automatically add one to our classes.

Example:

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

    // Default constructor
    Student() {
        System.out.println("Default constructor");
    }

    // Parameterized constructor
    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

🧠 Inheritance and Constructors

Constructors are **not inherited**!
When a subclass is instantiated:

- The **parent constructor** is called first (an instance of the parent class is created implicitly).

- You can explicitly call the parent constructor using `super()`, but it must be the **first line** in the subclass constructor.

```
class Parent {
    Parent() { System.out.println("Parent constructor"); }
}

class Child extends Parent {
    Child() {
        super();
        System.out.println("Child constructor");
    }
}
```

🔄 Method Overloading

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

Example:

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

⚠️ Overloading cannot be done **by changing only the return type**.

🔗 Coupling

Definition: Coupling represents the level of dependency between software entities (e.g. classes). The more our classes are dependent on each other, the harder it is to change them. Changing one class may result in several cascading and breaking changes.

- **Tight coupling:** Classes depend heavily on each other (harder to change).

- **Loose coupling:** Classes interact through interfaces or abstractions (easier to maintain).

Example: Using an interface reduces coupling.

```
interface PaymentProcessor {
    void process();
}

class PayPal implements PaymentProcessor {
    public void process() {
        System.out.println("Processing via PayPal");
    }
}
```

🧩 How Abstraction Reduces Coupling

By **hiding implementation details**, we prevent other parts of the code from depending on them.
Changes inside one class don't affect others.

🧩 Upcasting and Downcasting

Upcasting

Casting an object to one of its supertypes. This is done **implicitly** - wherever a reference to an object of a super class is expected, reference to a subclass object can be passed.

```
UIControl control = new TextBox();
control.render();  // calls overridden method
```

⚠️ If someMethod() is overridden in a subclass (TextBox), implementation from the subclass will be called!

Downcasting

Casting an object to one of its subtypes. Java compiler will allow this **explicit** casting, but **java.lang.ClassCastException** can be thrown during the runtime.

```
UIControl control = new UIControl();
TextBox textBox = (TextBox) control;  // ClassCastException
```

or

```
@Override
public boolean equals(Object obj) {
    var other = (Point) obj;
    //ClassCastException if obj is not instance of Point class
    return other.x == x && other.y == y;
}
```

🌀 Polymorphism — In Depth

1️⃣ Compile-time Polymorphism (Static)

Achieved by **method overloading** or **operator overloading** (like + for strings).

The method to be executed is decided at compile time.

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

2️⃣ Runtime Polymorphism (Dynamic)

Achieved by **method overriding** + **inheritance**.

The method to be executed is decided at runtime based on the actual object type.

```
class UIControl {
    void render() { System.out.println("Rendering UIControl"); }
}
```

```
class TextBox extends UIControl {
    @Override
    void render() { System.out.println("Rendering TextBox"); }
}
class CheckBox extends UIControl {
    @Override
    void render() {
        System.out.println("Rendering CheckBox");
    }
}

public class Main {
    public static void main(String[] args) {
        UIControl[] controls = { new TextBox(), new CheckBox() };
        for (UIControl control : controls) {
            control.render(); // Calls the overridden method of the
actual object
        }
    }
}
```

Output:

```
Rendering TextBox
Rendering CheckBox
```

Here, the compiler knows the type is `UIControl`, but at runtime it calls the appropriate overridden `render()` method based on the actual object (`TextBox` or `CheckBox`).

✅ Summary Table

Type	Achieved By	Decision Time
Compile-time	Method overloading	Compile time
Runtime	Method overriding	Runtime

🧱 Abstract Classes

An **abstract class** is a class that **cannot be instantiated on its own**.
It can have all:

- Abstract methods (methods without a body, just a signature)
- Concrete methods (methods with a body)
- Fields/variables (non-static too)
- Constructors

⚠️ Note:

- **Abstract class cannot be instantiated!**
- Its purpose is to provide some common fields and methods for subclasses.
- We can define an abstract class without any abstract methods.

Example:

```
abstract class Shape {
    abstract void draw();
    void info() { System.out.println("This is a shape"); }
}
```

🧩 Interfaces

- Define a **contract (specification)** without implementation.
- Methods are implicitly `public` and `abstract` (before Java 8).
    - From Java 8+, can include **default** and **static** methods.
    - From Java 9+, can include **private** methods.
- No constructors
- Fields are **implicitly public, static, and final**

Example:

```
interface Drawable {
    void draw();
}

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

```
}
```

🧱 Interface vs Abstract Class

Feature	Abstract Class	Interface
Fields	Yes	Only `public static final`
Methods	Abstract + Concrete	Abstract + Default + Static + Private
Constructors	Yes	No
Inheritance	Single (extends)	Multiple (implements)
Use Case	Shared behavior	Contract / capability

🧠 Functional, Marker, and Nested Interfaces

🧩 Functional Interface

A functional interface is an interface that contains **exactly one abstract method**.

✅ Key Points

- Can have **default, static, and private methods** (they don't count toward the "one abstract method" rule).
- Can have **methods from Object class** (like toString(), equals(), etc.) - Methods in java.lang.Object do not count as abstract methods for functional interfaces.
- Functional interface can be implemented using:
    1. lambda expressions
    2. method references
    3. anonymous classes (old way)
- Marked with the annotation @FunctionalInterface (optional, but recommended — compiler enforces the rule):

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

```
    public static void main(String[] args) {
        // Using lambda expression
        Calculator sum = (a, b) -> a + b;
        System.out.println(sum.add(5, 3)); // Output: 8
    }
}
```

👉 Lambda expressions and method references (implemented in Java 8) are syntax shortcuts for providing implementations of functional interfaces.

Common built-ins:

- `Predicate<T>` — returns `boolean`
- `Function<T, R>` — transforms input to output
- `Consumer<T>` — consumes value, no return
- `Supplier<T>` — returns a value

Core Java;s functional interfaces:

- `Runnable`
- `Callable<V>`
- `Comparable<T>`
- `Comparator<T>`

🚫 Marker Interface

An interface with **no methods or fields**. Used to mark or **tag** a class to provide metadata to the JVM or other code — it indicates that the class has some special behavior.

Examples:

- `java.io.Serializable`
- `java.lang.Cloneable`
- `java.util.RandomAccess`

🧠 Java 5 introduced **annotations**, which largely replaced marker interfaces for tagging classes.

🏗️ Nested Interface

An interface defined **inside a class or another interface**.
Used for logical grouping or encapsulation.

Example:

```
class Outer {
    interface Nested {
        void showMessage();
    }
}

class InnerImpl implements Outer.Nested {
    public void showMessage() {
        System.out.println("Hello from nested interface!");
    }
}
```

💎 What is a Diamond Problem

The diamond problem happens in languages that support **multiple inheritance**. If two classes (B, C) derive from A and are also the parents of another class (D), we see a diamond. If the top class (A) declares a method (e.g. toString) and its children (B and C) override this method, it's not clear which implementation will be inherited by D.

```
   A
  / \
 B   C
  \ /
   D
```

Diamond problem in Java (Java 8+): If a class implements **two interfaces that have the same default method**, Java **forces the class to override that method**. Otherwise, the compiler will throw an **ambiguity error.**

💾 3. Java Memory and Performance

💡 JVM Architecture Overview

JVM (Java Virtual Machine) consists of several key components:

1️⃣ Runtime Data Areas (Memory Areas)

Component	Description
Method Area / Metaspace	Stores class-level data: static variables, constants, bytecode, class metadata. **PermGen** - old implementation of method area, **Metaspace** - new (Java 8) implementation of method area.
Heap	Runtime memory for all objects and their **instance variables**; managed by **GC**
Stack	Stores **local variables** and method call frames; thread-local
PC Register	Keeps track of the **current instruction** being executed for each thread
Native Method Stack	Stores state and data for **native (non-Java) methods** called from Java

2️⃣ JVM Sybsystems/Components

Component / Subsystem	Description

Class Loader Subsystem	Dynamically loads `.class` files into the JVM. Supports hierarchy: **Bootstrap** → **Extension** → **Application**.
Execution Engine	Executes bytecode. Consists of: • **Interpreter** – executes bytecode line by line • **JIT Compiler** – converts hot bytecode to native machine code for performance.
Garbage Collector (GC)	Automatically frees memory of unreachable heap objects. Handles **young/old generation** and **metaspace** memory.
Native Interface	Enables Java code to interact with **native (C/C++) code**.

✅ **Notes**

- **Memory Areas**: Stack → thread-local, Heap → shared objects, Method Area → class metadata.
- **Subsystems**: JIT compiler optimizes runtime, GC manages memory, Class Loaders handle dynamic loading, Native Interface allows C/C++ calls.

🧠 Two Types of Java Memory

Java divides memory into different regions, primarily **Heap** and **Stack**, along with a **Method Area** used internally by the JVM.

Memory Area	What it Stores	Lifetime	Accessible Across Methods?
Stack	**Local variables:** primitives and references, **method calls**	Until the method ends	❌ No

Heap	**Objects** (new keyword) and their **instance variables** (both primitive and references)	Until garbage collected	✅ Yes
Method Area	**Class metadata, static variables**	Lifetime of JVM	✅ Yes

📘 Notes:

- **Stack memory** is fast — it uses **LIFO (Last-In-First-Out)** order.
- **Heap memory** is managed by the **Garbage Collector (GC)**.
- Objects created with `new` live in the **heap**.

♻️ Garbage Collection and References

🔹 1. What is Garbage Collection

Garbage Collection (GC) is an automatic memory management process of the JVM.
Its job: **detect unreachable objects and reclaim their memory**.

Key GC algorithms used by the JVM:

- **Mark-Sweep**
    - **Mark**: Traverse from GC roots and mark reachable objects
    - **Sweep**: Free unmarked objects
- **Mark-Sweep-Compact**
    - **Compaction:** Move live objects to eliminate fragmentation
- **Copying collectors**
    - Copies live objects from one region to another (Eden -> Survivor 0 -> Survivor 1)
- **Generational GC**

    - Heap is split by object “age”:
        1. **Young Gen:** frequent, fast collections (copying)
        2. **Old Gen:** less frequent, slower collections (mark–sweep–compact or region-based)
- **Region-based GC**
    - Heap is divided into many small regions (1–32MB).
    - GC works on a subset of regions instead of whole generations.
- **Concurrent Marking** (modern low-pause collectors)
    - Marking of reachable objects happens **while application is running**
    - Minimizes stop-the-world pauses
- **Concurrent Relocation (Concurrent Compaction)**
    - Move live objects **without stopping the world**
- **Incremental/Pause Prediction Algorithms**
    - Break large GC work into small chunks
    - Try to stay within a target pause time

🔹 2. The key idea: Reachability & GC Roots

All HotSpot GCs internally rely on reachability-based marking.

Java does NOT use reference counting.

An object is “alive” if it is reachable from **GC Roots**, which include:

- Stack frames of active threads (local variables)
- Static fields
- Active threads themselves
- JNI references

🧩 **Concept of reachability**:

- If an object is reachable → **must stay alive**
- If unreachable → **eligible for GC**

GC can collect cyclic graphs because it uses reachability, not reference counts.

🔹 3. Young vs Old Generation

Java heap is typically **generational**:

Young Generation

- Eden (where new objects are created) + S0 + S1 (two equally sized survivor spaces, used to store objects that survive GC in Eden; only one is in use at a time)
- Frequent, fast collections
- Uses **copying**: live objects moved between survivor spaces
- Surviving objects get aged and eventually promoted to Old Gen

Old Generation

- Larger memory
- Collections less frequent but more expensive
- Uses **marking + sweeping + (optionally) compaction or region-based** GC

Why generational?

Because *most objects die young* → collecting the young Gen is very efficient.

🔹 4. Major Garbage Collectors in the JVM (Actual Implementations)

Interviewers often ask for differences.

✔ **Serial GC**
- Single-threaded
- Suitable for small heaps, embedded JVMs
- **Enable:** `-XX:+UseSerialGC`

✔ **Parallel GC (Throughput Collector)**
- Multi-threaded, focuses on throughput
- Longer pauses than G1 but higher raw throughput
- **Enable:** `-XX:+UseParallelGC`

✔ **G1 GC (Default since Java 9+)**
- Region-based (heap split into 1–32 MB regions)
- Predictable pause times
- Concurrent marking
- Mostly compacts incrementally
- Replaces old **CMS**
- **Enable:** `-XX:+UseG1GC`

✔ **ZGC (Java 15+)**
- Ultra-low pause collector (< 1ms)
- Huge heap support (TB-sized)
- Fully concurrent (no long pauses)
- Compacts concurrently
- **Enable:** `-XX:+UseZGC`

✔ **Shenandoah**
- Similar goals to ZGC (low-pause)
- Compacts concurrently
- **Enable:** `-XX:+UseShenandoahGC`

🧩 If asked:
G1 = balanced default
ZGC = low pause
Parallel = maximum throughput
Serial = tiny environments

Serial and Parallel GC use classic generational copying + mark–sweep–compact.
G1, ZGC, and Shenandoah use region-based heaps with concurrent marking;
ZGC and Shenandoah additionally support fully concurrent relocation.
G1 uses pause prediction algorithms.

🔹 5. Stop-the-world (STW) events

Every GC — even concurrent ones — has small STW phases - all the threads are paused until GC finishes its work..

Examples:

- root snapshot (initial mark)
- reference processing
- final cleanup

Why it matters:
Latency-sensitive systems trade throughput for low pauses → choose G1/ZGC/Shenandoah.

🔹 6. Reference Types (Crucial for Interviews)

Java has 4 reference types:

✔ Strong reference
Default reference type
→ GC never collects it while reachable.

✔ SoftReference
- Collected **only when JVM is low on memory**
- Used for **caches**

✔ WeakReference
- Collected on **next GC cycle -** weak references **never prevent GC**
- Used in:
  - WeakHashMap
  - ClassLoader graph cleanup
  - Metadata caches

✔ PhantomReference
- The PhantomReference itself **does NOT hold the object** — `get()` always returns `null`.
- GC first runs finalization (if any), then enqueues the PhantomReference into a **ReferenceQueue**, and then the object becomes eligible for actual memory reclamation.

- Phantom references **never prevent GC**; they are used only as a notification mechanism that the object is dead.
- Used for:
    - Monitoring object cleanup
    - Building custom resource deallocators
- Example:

```
public class Test {
    public static void main(String[] args) {
        Object o = new Object();
        ReferenceQueue<Object> rq = new ReferenceQueue<>();
        PhantomReference<Object> pr = new PhantomReference<>(o,
rq);
        o = null;
        System.gc();  // not guarantied
        Reference<?> ref = rq.poll();  // null or pr
        System.out.println(ref == pr); // true or false
        System.out.println(pr.get());  // Always null
    }
}
```

Note: Weak and phantom references do not prevent referenced objects from being garbage collected.

🔹 7. Finalization (Deprecated & Dangerous)

`finalize()` is **deprecated** since Java 9 and will be removed. It was used to execute code (cleanup) before the object was removed by GC.

Why?

- Unpredictable timing
- Can resurrect objects
- Causes performance issues
- Difficult to debug

Replacement: **Cleaner API**, **try-with-resources.**

🔹 8. Memory Leaks in Java

Even with GC, memory leaks can happen when objects remain reachable unintentionally.

Common leak sources:

- Static collections holding references
- Caches without eviction (LRU, LFU, TTL, FIFO)
- Listener not deregistered
- Threads (ThreadLocal leaks)
- Unbounded queues (ExecutorServices)
- Poorly designed singletons

Interview-friendly example:

```
static List<Object> cache = new ArrayList<>();
public void add(Object o) {
    cache.add(o); // never removed → memory leak
}
```

🔹 9. GC Tuning Basics

Often asked: "How do you tune GC?"

Key parameters:

```
-Xms   (initial heap)
-Xmx   (max heap)
-Xmn   (young gen size – older collectors)
-XX:+UseG1GC
-XX:+UseZGC
-XX:+UseParallelGC
```

Most important metrics:

- GC pause time
- Promotion failures
- Allocation rate
- Survivor space occupancy

Typical goals:

☑ Reduce pauses
☑ Improve throughput
☑ Avoid premature old-gen promotion
☑ Avoid "stop-the-world full GC"

🔹 10. What triggers a Full GC?

This question appears frequently.

Triggers:

- Old gen is full
- G1 concurrent cycle failed
- Metaspace is full
- `System.gc()` call (unless disabled via `-XX:+DisableExplicitGC`; it is only a suggestion to JVM)
- Allocation failure
- Promotion failure from Young → Old

🔹 11. Escape Analysis (JIT Optimization)

Escape analysis decides if an object:

- **escapes the method**
- **escapes the thread**
- **never escapes**

If it **does not escape**, the JVM can:

- Allocate it on the stack (scalar replacement)
- Remove synchronization

Interview takeaway:

> Escape analysis reduces heap allocation → reduces GC pressure.

🔹 12. GC Logging (How to observe GC)

```
java -Xlog:gc* MyApp
```

Or older:

```
-verbose:gc
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps
```

🔹 13. ClassLoader Leaks

A classic senior-level question:

> Why do app servers leak memory after redeployment (redeployment does not include application server restart)?

Core reason:
Application servers deploy each application using a **dedicated ClassLoader**.

On redeploy, a **new ClassLoader** is created and the old one must become **unreachable** to be garbage-collected.

A memory leak occurs when the **old ClassLoader is still referenced**, preventing it (and everything it loaded) from being reclaimed.

Key mechanics:

- Every object → references its `Class` (java.lang.Class object)
- Every `Class` → references its defining `ClassLoader`
- A `ClassLoader` → holds references to all classes it loaded (java.lang.Class objects)

➡ If any object remains reachable, the entire ClassLoader graph stays alive

---

Typical leak causes:

- **Threads** started by the application that are not stopped on undeploy
  (threads are GC roots and retain the app's context ClassLoader)
- **Static fields** in shared or parent-loaded classes holding references to app classes or instances
- **ThreadLocal values** not removed when using thread pools
- **Global registries / caches** (JMX, JDBC drivers, logging frameworks) retaining app objects

---

Leak effect:

```
GC Root
 └── Thread / static field
      └── App object
```

```
└── Class
    └── Old WebAppClassLoader
        └── All app classes + metadata
```

➡ Old application **cannot be unloaded**
➡ **Metaspace grows** (class metadata retained)
➡ Heap may grow as well (static fields, cached objects)

Why GC cannot fix it:
GC works on **reachability**, not deployment boundaries.
As long as the old ClassLoader is reachable, **nothing it loaded can be collected**.

Interview one-liner:

> **ClassLoader leaks occur when objects loaded by a parent or shared component retain references to classes or instances loaded by an application's ClassLoader, preventing the old ClassLoader and its Metaspace from being garbage-collected after redeploy.**

✅ Checklist summary for preventing ClassLoader leaks

Cause	Prevention
Threads	Stop them / use container-managed pools
ThreadLocals	Always call `remove()`
JDBC / Drivers	Deregister
Static fields	Avoid holding app classes in shared libraries
Caches / Singletons	Clear on undeploy
Listeners / MBeans	Unregister on undeploy

🔹 14. Metaspace (Post-Java 8)

Before Java 8 → **PermGen**

After Java 8 → **Metaspace** (native memory - not heap)

Metaspace stores:

- Class metadata (including static fields)
- Constant pool
- Method-related data

Metaspace exhaustion → Full GC → `OutOfMemoryError: Metaspace`

🔹 15. Common GC-related Interview Q&A (Quick Answers)

❓ Can GC collect a running thread?

No — threads are GC roots.

❓ Can soft references cause memory leak?

Yes — because they survive until memory pressure occurs.

❓ Does GC collect objects in use by JNI?

Not while referenced via JNI handles.

❓ What is a "promotion failure"?

Young GC wants to promote object to Old Gen → no space → triggers Full GC.

❓ What is "compaction"?

Reordering memory to eliminate fragmentation.

✔ Final: One-Screen Summary Table

Concept	Explanation
Reachability	Objects reachable from GC roots are live
Generations	Young = frequent, cheap; Old = expensive
Major GCs	G1 (default, balanced), ZGC (low pause), Parallel (throughput)
Reference types	Strong, Soft, Weak, Phantom
Common leaks	Statics, listeners, ThreadLocals, caches
Full GC triggers	Old gen full, metaspace full, System.gc(), promotion failure, allocation failure
Escape analysis	Allows stack allocation, reduces GC work
STW events	GC pauses all threads during certain phases

🧩 The Two Java Compilers

When you run a Java program, two compilers play a role:

1. **Java Compiler (`javac`)** — converts source code to bytecode.
2. **JIT Compiler (Just-In-Time)** — converts bytecode to machine code at runtime.

⚙️ Stage 1: `javac` Compiler (Before Execution)

Input: `.java` source file
Output: `.class` bytecode file

```
javac MyProgram.java
```

Purpose:

- Translates human-readable Java code into **platform-independent bytecode**.
- The output `.class` file runs on any system with a JVM.

Key Points:

- Happens **once before execution**.
- Produces portable `.class` files.

⚙️ Stage 2: JIT Compiler (During Execution)

Definition:
The **Just-In-Time (JIT)** compiler is part of the JVM. It converts **bytecode** into **native machine code** *at runtime* for faster execution.

How It Works:

1. JVM interprets bytecode line by line.
2. Detects “hot spots” (frequently executed code).
3. Compiles these to **native (machine) code**.
4. Caches the native code for reuse.

Purpose: Speed up execution by reducing interpretation overhead.

🧮 Analogy

Process	Analogy
`javac` compiler	Translating a book into a universal language (bytecode)

JIT compiler	A translator who memorizes and speeds up repeated sentences

🧾 Comparison Table

Feature	Java Compiler (`javac`)	JIT Compiler
When It Runs	Before execution	During execution
Input	`.java` source	`.class` bytecode
Output	`.class` bytecode	Native machine code
Runs On	Developer's machine (JDK)	Inside JVM
Purpose	Portability	Performance
Type	Static compilation	Dynamic (runtime) compilation

🚀 Summary

- `javac` → Converts Java source → **Bytecode (platform-independent)**
- JIT → Converts Bytecode → **Machine code (optimized for current OS/CPU)**

Together, they make Java both **portable and fast**.

🧩 JIT Compiler Recap

- JIT is **enabled by default**.
- Improves performance by **caching native code** for frequently used methods.
- Part of the **HotSpot JVM**.
- Allows runtime optimizations like **method inlining**, **dead-code elimination**, and **loop unrolling**.

🧩 What is Escape Analysis?

Escape analysis tracks whether an object:

- **escapes the method**, or
- **escapes the thread**, or
- **does not escape at all**.

Depending on this, the JVM can apply powerful optimizations.

🔍 Levels of Escape

1. No Escape (Non-escaping object)

The object is used **only within the method** and **never returns or is passed elsewhere**.

→ **JVM can allocate object on the stack**
→ **Object may be completely optimized away**

2. Method Escape

The object escapes the method but stays within the **same thread**.
Example: returning an object.

→ **Must be allocated on the heap**
→ Can still be optimized (e.g., scalar replacement)

3. Thread Escape

The object is shared between threads (e.g., stored in a global variable, passed to another thread).

→ **Must be heap allocated**

🚀 What Optimizations Does Escape Analysis Enable?

1. Stack Allocation

If object does **not escape**, JVM allocates on the **stack**, not the heap.

👉 This avoids GC pressure and speeds up allocation.

2. Scalar Replacement

If the JVM sees that the object is never used as a whole, only its fields are accessed:

→ The JVM **removes the object entirely**
→ Replaces fields with separate variables ("scalars")

```
class Point {
    int x, y;
}

void test() {
    Point p = new Point(); // might be eliminated
    p.x = 10;
    p.y = 20;
    int z = p.x + p.y;
}
```

The JVM may optimize this to:

```
int x = 10;
int y = 20;
int z = x + y;
```

No object allocation!

3. Elimination of Synchronization

If the JVM can prove that a synchronized object is **never accessed by multiple threads**, it removes the synchronization:

```
public void test() {
    synchronized(new Object()) {   // lock can be removed
```

```
        ...
    }
}
```

🔧 How to View Escape Analysis in Action

Run with JVM flags:

```
-XX:+UnlockDiagnosticVMOptions -XX:+PrintEscapeAnalysis
-XX:+PrintEliminateAllocations
```

You will see logs like:

```
Eliminated: allocation eliminated
```

or:

```
Escape: scalar replaceable
```

📌 When Escape Analysis Helps Most

- Short-lived temporary objects
- Loops that create objects each iteration
- Functional style code with many small objects
- Builder- or DSL-style code
- Small value-like objects (e.g., pairs, points, events)

❗ Important Notes

✔ **Escape analysis is performed at runtime**

It is not a compiler feature; it's a JVM optimization in tiered compilation (C2 JIT).

✔ **Escape analysis is not guaranteed**

The JVM may choose not to optimize depending on complexity.

✔ **Does not help with very large objects**

Large objects usually stay on the heap.

🧪 Example: Object Does Not Escape

```
public int compute() {
    Foo f = new Foo(); // may be stack allocated
    f.x = 10;
    return f.x * 2;
}
```

→ No escape → stack alloc or scalar replacement.

🧪 Example: Method Escape

```
public Foo createFoo() {
    Foo f = new Foo();
    return f; // escapes
}
```

→ Must be heap allocated.

🧪 Example: Thread Escape

```
Foo f = new Foo();
```

```
new Thread(() -> {
    System.out.println(f.x); // escapes to another thread
}).start();
```

→ Heap allocated

📘 Summary

Escape analysis allows Java to:

- Allocate objects on **stack - reduce GC pressure**
- Avoid actual object creation (**scalar replacement**)
- Remove unneeded **synchronization**
- Improve performance of object-heavy code

It's one of the reasons Java can run allocation-heavy programs efficiently.

🧩 Class Loaders

In Java, a **Class Loader** is a fundamental part of the**Java Virtual Machine (JVM)** that **dynamically loads classes (.class files) into memory** when they are required by a program.

🧠 What Is a Class Loader?

When you run a Java program, the **JVM doesn't load all classes at once**. Instead, it loads them **on demand** — when a class is first referenced.
This is the job of the **Class Loader subsystem**.

⚙️ Responsibilities of Class Loader

1. **Loading** — Reads `.class` files (bytecode) and brings them into JVM memory.
2. **Linking** — Verifies, prepares, and resolves class dependencies.
    - **Verification:** Ensures the bytecode is valid and safe.
    - **Preparation:** Allocates memory for static fields and sets default values.
    - **Resolution:** Converts symbolic references to actual memory references.
3. **Initialization** — Executes static initializers and assignments.

🧩 How It Works

When you use `new`, or reference a class for the first time:

1. JVM asks the **ClassLoader** to locate and load the `.class` file.
2. The ClassLoader reads the bytecode and creates a `Class` object.
3. The `Class` object is stored in JVM memory

🏗️ Types of Class Loaders

Java uses a **hierarchical delegation model** for class loading.

Class Loader	Description	Loads Classes From
Bootstrap Class Loader	The root loader, part of the JVM. It loads core Java classes.	`<JAVA_HOME>/lib` (e.g., `rt.jar`, `java.base` module)
Extension (Platform) Class Loader	Loads classes from the extensions directory.	`<JAVA_HOME>/lib/ext` or modules in `jmods`
Application (System) Class Loader	Loads classes from the classpath (your app's `.class` or `.jar` files).	`CLASSPATH` or `java -cp` option

Custom Class Loader	User-defined loader that extends `ClassLoader`.	User-defined sources (e.g., network, database, encrypted jar)

🔁 Delegation Model

When a class loader receives a request to load a class:

1. It first **delegates** the request to its **parent** loader.
2. If the parent **cannot find** the class, only then it **tries to load** it itself.

Example Diagram

```
       [Bootstrap ClassLoader]
                  ↓
     [Extension/Platform ClassLoader]
                  ↓
    [Application/System ClassLoader]
                  ↓
          [Custom ClassLoader]
```

This ensures:

- Core Java classes aren't overridden by user code (security).
- Classes are loaded only once (efficiency).

🧩 Example: Custom Class Loader

```
public class MyClassLoader extends ClassLoader {

    @Override
    protected Class<?> findClass(String name) throws
ClassNotFoundException {
        // Load bytecode from custom location
        byte[] bytes = loadClassData(name);
        return defineClass(name, bytes, 0, bytes.length);
    }

    private byte[] loadClassData(String name) {
        // Read bytes from file, network, etc.
        return new byte[0]; // placeholder
    }
```

```
}
```

You can then use it like:

```
MyClassLoader loader = new MyClassLoader();
Class<?> cls = loader.loadClass("com.example.MyClass");
```

🔒 Why Class Loaders Matter

- **Security:** Prevents untrusted code from interfering with core classes.
- **Flexibility:** Enables dynamic module loading (e.g., in web servers, plugins).
- **Isolation:** Each class loader can load separate versions of the same class — used in application servers like Tomcat or OSGi.

🧰 Real-World Examples

- **Web containers (Tomcat, Jetty)** use different class loaders for each web app to isolate dependencies.
- **Frameworks (Spring Boot)** use class loaders to scan and load classes dynamically.
- **Java Reflection and dynamic proxies** rely on class loading mechanisms internally.

🔑 4. Java Keywords and Modifiers

🧩 Keywords Overview

In Java, **keywords** are reserved words that have predefined meanings in the language. Here we'll focus on the important ones that directly affect class behavior and inheritance relationships:

- `this`

- `super`
- `static`
- `final`
- Access modifiers (`private`, `protected`, `public`, `default`)
- Non-access modifiers (`abstract`, `synchronized`, `volatile`, etc.)

🧠 Keyword `this`

`this` represents a reference to the **current object** of the class.

🔹 Use cases:

1. **Accessing current object's fields or methods:**

```
  class Student {
    private String name;
    public void setName(String name) {
        this.name = name;  // distinguish field from parameter
    }
}
```

2. **Invoking current class constructor:**

```
  public Student() {
    this("Unnamed"); // calls another constructor in the same class
}
```

3. **Returning the current object:**

```
  public Student getThis() {
    return this;
}
```

4. Passing as an argument to methods or constructors:

```
someMethod(this);
```

🧩 Keyword `super`

`super` refers to the **immediate parent class object**.

Whenever you create a subclass instance, an instance of the parent class is created implicitly — accessible via `super`.

🔹 Use cases:

1. **Access parent class fields or methods:**

```
  class Animal {
void sound() { System.out.println("Animal sound"); }
}
class Dog extends Animal {
    void sound() {
        super.sound();  // call parent method
        System.out.println("Bark");
    }
}
```

2. **Invoke parent class constructor:**

```
  class Child extends Parent {
Child() {
    super();  // must be first line in constructor
}
}
```

3. **Pass parent reference as argument:**

```
someMethod(super);
```

4. **Returning the parent object: ❌ NOT ALLOWED**

```
public Parent getParent() {
```

```
        return super;   // ❌ Compile-time error
    }
```

🧩 Keyword `static`

- Belongs to the **class**, not to specific objects.
- Shared among all instances.
- Can be used with **variables, methods, blocks, and nested classes**.

Example:

```
class Counter {
    static int count = 0;  // shared variable
    Counter() { count++; }
}
```

Key Notes:

- Static methods **cannot access non-static members** directly.
- Static blocks execute **once** when the class is loaded.
- Static variables are stored in the **Method Area**.

🔒 Access Modifiers

Access modifiers define **visibility and accessibility** of classes, methods, and variables.

Modifier	Accessible Within Class	Same Package	Subclass	Other Packages

private	✅	❌	❌	❌
default (no keyword)	✅	✅	❌	❌
protected	✅	✅	✅	❌ (except through inheritance)
public	✅	✅	✅	✅

🧩 Important Notes:

- **Classes** can only be declared as `public` or **default** (package-private).
  Inner classes, however, can be declared with `private` or `protected` access.
- If no modifier is specified, it defaults to **package-private** (visible only in the same package).

⚙️ Non-Access Modifiers

Non-access modifiers change the **behavior** of classes, methods, and variables rather than their visibility.

Modifier	Description
static	Belongs to the class rather than instance
final	Cannot be reassigned, overridden, or inherited (depending on use)
abstract	Declares methods or classes that must be implemented/extended
synchronized	Ensures thread-safe execution of a method or block
volatile	Marks variable value as always read from main memory
transient	Prevents variable from being serialized
native	Indicates the method is implemented in native (non-Java) code

---

🧩 Default Modifier — Two Meanings

1️⃣ Default Methods in Interfaces (Java 8+)

- Applies to **methods in interfaces**.
- Provides a **default implementation** that implementing classes can override.

```
interface Vehicle {
    default void honk() {
        System.out.println("Honking...");
    }
}
```

✅ **Note:**
If two interfaces contain the same default method and a class implements both, the class **must override** the method to resolve the conflict (diamond problem).

2️⃣ Default Access (Package-Private)

If no modifier is specified on a class, method, or variable, it is **accessible only within the same package**.

```
class MyClass { // package-private
    void greet() { // package-private
        System.out.println("Hello");
    }
}
```

✅ Summary Table: Default Keyword & Default Access

Keyword / Concept	Applies To	Meaning
`default` keyword	Interface methods	Provides a default implementation

(no modifier)	Classes, methods, fields	Default (package-private) access

---

🧱 `final` Keyword

Used to mark **classes, methods, and variables** as unchangeable.

Context	Meaning
final variable	Value cannot be reassigned after initialization
final method	Cannot be overridden in subclasses
final class	Cannot be extended

Example:

```
final class MathUtils { }  // cannot be subclassed

class Example {
    final int number = 10;  // cannot be changed
}
```

💡 ***Remember:***
`final` only protects the **reference**, not the **object's content** (see next section on immutability).

🧠 5. Immutability and Strings

🔹 What is Immutability?

Immutability means an object's **state cannot be changed** after it is created.
Instead of modifying an existing object, operations create **a new instance**.

In Java, the most well-known immutable class is **`String`**.

🧩 Why Are Strings Immutable?

Java `String` objects are immutable because of **security, performance, and memory** reasons:

1. **Security:**
   Strings are often used in sensitive contexts (e.g., URLs, file paths, network connections). Immutability prevents them from being altered after creation.

2. **Hashcode Caching:**
   Since `String` is immutable, its hashcode can be cached — this makes lookups in collections like `HashMap` faster.

3. **Thread Safety:**
   Multiple threads can share the same `String` without synchronization issues.

4. **String Pooling:**
   JVM stores all string literals in a **String pool** for reuse.
   If strings were mutable, this mechanism would be unsafe.

💡 Example: String Behavior

```
String s1 = "Java";
String s2 = s1.concat(" Rocks!");

System.out.println(s1);  // Output: Java
System.out.println(s2);  // Output: Java Rocks!
```

Even after concatenation, `s1` remains unchanged — a **new object** (`s2`) is created.

🔍 String Pool

The **String Constant Pool (SCP)** is a special memory region inside the **heap**, managed by the **JVM**.

There are **two main ways** to instantiate a String:

- Use **string literal** - When you create strings using literals, Java reuses them from the pool (memory-efficient).

```
String s1 = "Hello";
String s2 = "Hello";    // points to the same object in the pool
```

- Using **new** keyword - Creates a **new String object in heap memory** (outside the String Pool). Even if "Hello" exists in the String Pool, a new object is created in the heap. Slower, more memory used.

```
String s3 = new String("Hello"); // creates a new object outside
the pool
```

Expression	Location	New Object Created?
`"Hello"`	String Pool	Only once
`new String("Hello")`	Heap	Always yes

`intern()` method in java.lang.String:

- If the string **already exists** in the pool, `intern()` returns the **reference** from the pool.
- If the string **does not exist** in the pool, it is **added** to the pool and the reference is returned

---

🧩 `final` vs Immutable

`final` and immutability are related but **not the same**.

Concept	Affects	Meaning
final variable	Reference	Reference cannot point to a new object
immutable class	Object	Object's internal state cannot be modified

Example:

```
final StringBuilder sb = new StringBuilder("Hi");
sb.append(" there");     // allowed — object state changed
// sb = new StringBuilder("Hello");  // not allowed
```

✅ The variable `sb` is `final`, but the `StringBuilder` object is **mutable**.

🧱 How to Create an Immutable Class

To make a custom class immutable, follow these rules:

1. Declare the class as `final`.
2. Make all fields `private` and `final`.
3. Don’t provide any setters.
4. Initialize all fields through a constructor - this actually is not correct, final fields can also be initialized at declaration and in instance initialization blocks. What only is true, it is that they need to be initialized.
5. Return **defensive copies** of mutable objects - don’t expose mutable objects.

Example:

```
final class Person {
    private final String name;
    private final Date birthDate;

    public Person(String name, Date birthDate) {
        this.name = name;
        this.birthDate = new Date(birthDate.getTime()); // defensive
copy
    }

    public String getName() { return name; }

    public Date getBirthDate() {
        return new Date(birthDate.getTime()); // return copy
```

```
    }
}
```

🔁 String vs StringBuilder vs StringBuffer

Feature	String	StringBuilder	StringBuffer
Mutability	Immutable	Mutable	Mutable
Thread Safety	Safe (immutable)	❌ Not thread-safe	✅ Thread-safe
Performance	Slower (creates new objects)	Faster	Slower (due to sync)
Use Case	Fixed strings	Single-threaded concatenation	Multi-threaded concatenation

Example:

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");
System.out.println(sb); // Hello World
```

⚙️ Why Use `StringBuilder`?

- More efficient than string concatenation (`+`) in loops.
- Reduces object creation overhead.

Example:

```
String result = "";
for (int i = 0; i < 1000; i++) {
    result += i;  // inefficient – creates new String every time
}

StringBuilder sb = new StringBuilder();
```

```
for (int i = 0; i < 1000; i++) {
    sb.append(i);  // efficient – modifies same buffer
}
```

🧩 Common String Methods

Method	Description	Example
`length()`	Returns string length	`"Java".length()` → `4`
`charAt(i)`	Returns character at index	`"Java".charAt(2)` → `'v'`
`substring()`	Extracts part of string	`"Java".substring(1, 3)` → `"av"`
`equals()` / `equalsIgnoreCase()`	Compares content	`"Java".equals("JAVA")` → `false`
`compareTo()`	Lexicographic comparison	`"A".compareTo("B")` → `-1`
`toLowerCase()` / `toUpperCase()`	Changes case	`"java".toUpperCase()` → `"JAVA"`
`trim()`	Removes leading/trailing spaces	`" Java ".trim()`
`replace(a, b)`	Replaces chars	`"Java".replace('a', 'o')` → `"Jovo"`
`split()`	Splits by regex	`"a,b,c".split(",")`
`intern()`	Adds to string pool	`str.intern()` returns pooled reference

💾 `String.valueOf()` vs `toString()`

Method	Purpose	Works On
`String.valueOf()`	Converts any type to string (handles nulls safely)	All primitives and objects

`toString()`	Converts object to string (may throw NPE if null)	Objects only

Example:

```
Object obj = null;
System.out.println(String.valueOf(obj));  // prints "null"
System.out.println(obj.toString());     // throws NullPointerException
```

⚠️ 6. Exceptions and Error Handling

🧩 What Is an Exception?

An **exception** is an **event** that disrupts the normal flow of a program.
It is an object representing an **error or unexpected condition** that occurs during program execution.

When an exception occurs:

1. The method where it occurred creates an **Exception object**.
2. The object is passed to the **runtime system (JVM)**.
3. The JVM searches for **matching exception handlers** (`catch` blocks).
4. If none found → **program terminates**.

💥 Error vs Exception

Type	Description	Examples

Error	Serious problem beyond the program's control	`OutOfMemoryError`, `StackOverflowError`, `VirtualMachineError`
Exception	Problem that can be handled programmatically	`IOException`, `ArithmeticException`, `NullPointerException`

Errors are generally **unchecked** and **should not be caught**.
Exceptions can be either **checked** or **unchecked**.

⚖️ Checked vs Unchecked Exceptions

Type	Inheritance	Checked at	Must be handled?	Examples
Checked	Extends `Exception` (but not `RuntimeException`)	Compile time	✅ Yes	`IOException`, `SQLException`, `FileNotFoundException`
Unchecked	Extends `RuntimeException`	Runtime	❌ No	`NullPointerException`, `ArrayIndexOutOfBoundsException`

Example:

```
// Checked
try {
    FileReader file = new FileReader("nonexistent.txt");
} catch (IOException e) {
    e.printStackTrace();
}

// Unchecked
int x = 5 / 0;  // ArithmeticException
```

🔁 Exception Hierarchy

```
Throwable
 ├── Error
 │    ├── OutOfMemoryError
 │    ├── StackOverflowError
 │    └── ...
 └── Exception
      ├── IOException
      ├── SQLException
      ├── RuntimeException
      │     ├── NullPointerException
      │     ├── ArithmeticException
      │     └── ...
```

🧩 Try-Catch-Finally

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero");
} finally {
    System.out.println("This block always executes");
}
```

- The `finally` block **always executes**, even if an exception occurs or a `return` statement is hit.
- Used for **resource cleanup** (e.g., closing files or database connections).

🧰 Multi-Catch Block

Since Java 7, multiple exceptions can be handled in one block:

```
try {
```

```
    int[] arr = new int[5];
    arr[5] = 10;
} catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {
    System.out.println("Exception caught: " + e);
}
```

⚠️ Both exceptions must be **unrelated by inheritance**.

🧱 Nested Try Blocks

You can place a `try` block inside another `try`.

```
try {
    try {
        int num = 10 / 0;
    } catch (ArithmeticException e) {
        System.out.println("Inner catch");
    }
} catch (Exception e) {
    System.out.println("Outer catch");
}
```

💾 Try-with-Resources (Java 7+)

Automatically closes resources that implement **`AutoCloseable`**.

Example:

```
try (FileReader fr = new FileReader("data.txt");
     BufferedReader br = new BufferedReader(fr)) {
    System.out.println(br.readLine());
} catch (IOException e) {
    e.printStackTrace();
}
```

✅ Equivalent to using a `finally` block with manual `.close()` calls, but cleaner and safer.

🧩 Throw and Throws

Keyword	Purpose	Used In
`throw`	Actually throws an exception	Inside a method
`throws`	Declares exceptions a method may throw	In method signature

Example:

```
void divide(int a, int b) throws ArithmeticException {
    if (b == 0)
        throw new ArithmeticException("Cannot divide by zero");
}
```

🧠 Custom Exceptions

You can create your own checked or unchecked exceptions.

Example: Checked Exception

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

void validateAge(int age) throws InvalidAgeException {
    if (age < 18)
        throw new InvalidAgeException("Underage");
}
```

Example: Unchecked Exception

```
class CustomRuntimeException extends RuntimeException {
```

```
    public CustomRuntimeException(String message) {
        super(message);
    }
}
```

🚫 Common Runtime Exceptions

Exception	Cause
`NullPointerException`	Accessing a method/field on a `null` object
`ArrayIndexOutOfBoundsException`	Accessing invalid index in array
`ArithmeticException`	Division by zero
`NumberFormatException`	Invalid conversion (e.g., `"abc"` → int)
`ClassCastException`	Invalid type casting
`IllegalArgumentException`	Invalid argument passed to method

🧩 Re-throwing Exceptions

You can catch an exception and rethrow it — often to add context.

```
try {
    readFile();
} catch (IOException e) {
    throw new RuntimeException("Error reading configuration file", e);
}
```

🧰 Exception Chaining

Attaching the original cause to a new exception:

```
catch (SQLException e) {
    throw new IOException("Database failure", e);
```

```
}
```

Use `Throwable.getCause()` to access the original exception.

🧱 Best Practices

✅ DO:

- Catch only exceptions you can handle meaningfully.
- Use `try-with-resources` for automatic cleanup.
- Wrap low-level exceptions into meaningful custom ones.
- Log exceptions instead of swallowing them.

🚫 DON'T:

- Catch `Throwable` or `Exception` unless absolutely needed. That:
  - Catches things you shouldn't handle (OutOfMemoryError, StackOverflowError)
  - Hides real bugs
  - Makes debugging harder
- Leave catch blocks empty. Why is it bad?
  - Error disappears silently
  - Program continues in broken state
  - Impossible to debug in production
- Use exceptions for control flow logic.

  Example:

```
public boolean isEmailTaken(String email) {
    try {
        userRepository.findByEmail(email);
        return true;
    } catch (EntityNotFoundException e) {
        return false;
    }
}
```

Why this is bad:

- Exceptions are expensive (stack trace creation)
- Makes logic harder to read
- Hides real problems
- Breaks clean flow

Instead, do this:

```
public boolean isEmailTaken(String email) {
    return userRepository.existsByEmail(email);
}
```

- Ignore interrupted exceptions in multithreading code.

🔍 7. Advanced Java Topics

🧩 Reflection in Java

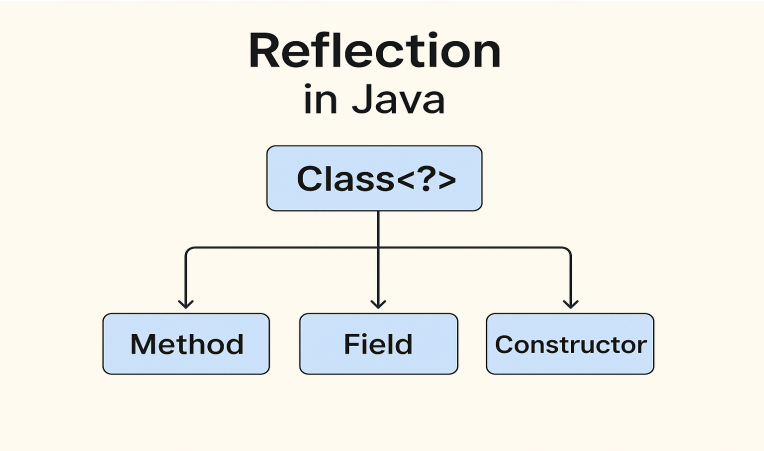
Reflection is a powerful mechanism that allows a Java program to **inspect and manipulate classes, methods, and fields** at runtime — even if you don't know their names at compile time.

⚙️ Key Uses of Reflection

- Inspecting class information (fields, methods, constructors).
- Accessing private members.
- Instantiating objects dynamically.
- Building frameworks (Spring, Hibernate, testing tools, etc.).
- Annotation processing.

🧠 Reflection API — Important Classes

Class	Purpose
`Class`	Represents metadata about a loaded class
`Method`	Represents a method in the class
`Field`	Represents a field (variable)
`Constructor`	Represents a constructor
`Modifier`	Provides utility methods to read access modifiers



💡 Example: Getting Class Info (inspecting class)

```
class Person {
    private String name;
    public void sayHello() { System.out.println("Hello!"); }
}

public class ReflectionExample {
    public static void main(String[] args) {
        Person p = new Person();
        Class<?> clazz = p.getClass();

        System.out.println("Class name: " + clazz.getName());
        Method[] methods = clazz.getDeclaredMethods();
        for (Method m : methods) {
```

```
            System.out.println(m.getName());
        }
    }
}
```

🔐 Accessing Private Fields

```
import java.lang.reflect.*;

class Secret {
    private String data = "hidden";
}

public class Reveal {
    public static void main(String[] args) throws Exception {
        Secret s = new Secret();
        Field field = s.getClass().getDeclaredField("data");
        field.setAccessible(true);
        System.out.println(field.get(s));  // prints "hidden"
    }
}
```

⚠️ Important notes:

- **Performance:** Reflection is slower than normal code because it bypasses compile-time optimizations.
- **Security:** Accessing private members via reflection can break encapsulation and security policies.
- **Use Cases:** Serialization, dependency injection (Spring), testing frameworks (JUnit),

---

🧩 Inner Classes in Java

An **inner class** is defined **inside another class**.
It helps logically group classes that are only used in one place.

🧠 Types of Inner Classes

Type	Defined As	Can Access Outer Class Members?	Static?
Non-static inner class	Inside another class	✅ Yes	❌ No
Static nested class	Inside another class	❌ Only static members	✅ Yes
Local class	Inside a method	✅ Yes	❌ No
Anonymous class	Inline implementation	✅ Yes	❌ No

💡 Example: Non-Static Inner Class

```
class Outer {
    private int data = 10;
    class Inner {
        void show() {
            System.out.println("Data: " + data);
        }
    }
}

new Outer().new Inner().show();
```

💡 Example: Static Nested Class

```
class Outer {
    static int data = 30;
    static class Inner {
        void msg() {
            System.out.println("Data: " + data);
        }
    }
}
Outer.Inner obj = new Outer.Inner();
```

```
obj.msg();
```

💡 Example: Local and Anonymous Classes

```
class Outer {
    void display() {
        class Local {
            void show() { System.out.println("Local class method"); }
        }
        Local l = new Local();
        l.show();
    }
}
```

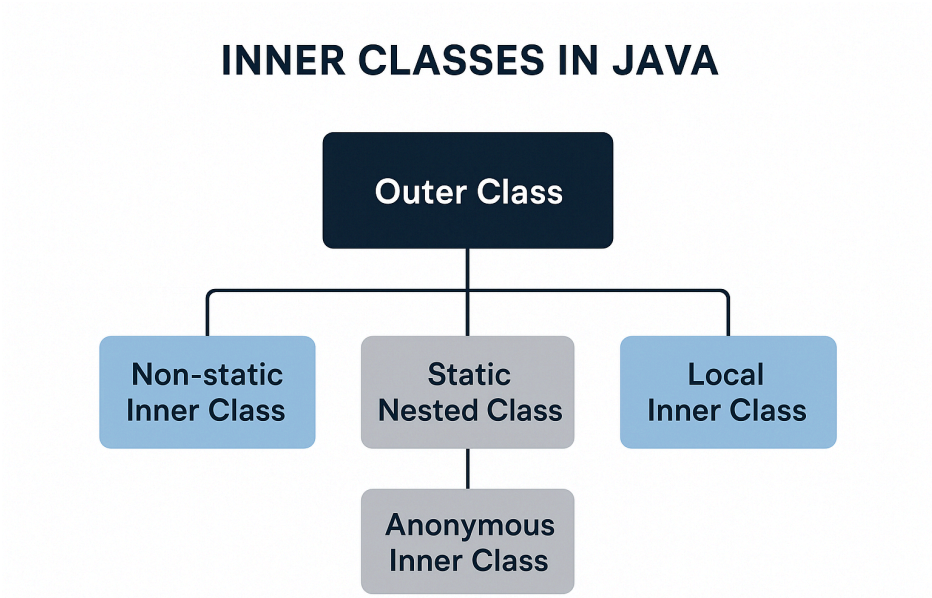
Anonymous Example:

```
interface Greeting {
    void say();
}

Greeting g = new Greeting() {
    public void say() { System.out.println("Hello!"); }
};
g.say();
```

🧩 Why Use Inner Classes?

- **Logical grouping**: Classes that are only useful within the context of another class.
- **Encapsulation**: Inner classes can be private, hiding implementation details.
- **Simplify code**: Useful for event handling (like in GUIs) or callbacks.



🧩 Annotations in Java

Annotations provide **metadata** about code that can be processed at compile time or runtime.

⚙️ Common Built-In Annotations

Annotation	Purpose
`@Override`	Ensures method overrides a superclass method
`@Deprecated`	Marks code as outdated
`@SuppressWarnings`	Hides compiler warnings
`@FunctionalInterface`	Ensures interface has one abstract method
`@SafeVarargs`	Prevents warnings for varargs with generics

🧠 Meta-Annotations (Annotations for Annotations)

Meta-Annotation	Purpose

`@Retention`	Specifies how long the annotation is retained
`@Target`	Specifies where the annotation can be applied (class, method, field, etc.)
`@Inherited`	Allows subclass to inherit parent annotation
`@Documented`	Includes annotation in Javadoc

💡 Example: Custom Annotation

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface RunMe {
    int times() default 1;
}

public class Demo {
    @RunMe(times = 3)
    public void test() {
        System.out.println("Running test");
    }
}
```

But note: Java **does not automatically execute it 3 times**.
Annotations themselves **do nothing** unless you write reflection code to process them.

🧩 Processing Annotations Using Reflection

```
Demo demo = new Demo();
for (Method m : Demo.class.getDeclaredMethods()) {
    RunMe ann = m.getAnnotation(RunMe.class);
    if (ann != null) {
        for (int i = 0; i < ann.times(); i++) {
            m.invoke(demo);
        }
    }
}
```

✅ Output:

```
Running test
Running test
Running test
```

🧱 Enums — Advanced Features

Definition:
An `enum` is a special Java type used to define a **fixed set of constants** (like days of the week, states, directions).

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

Key points:

- Implicitly `final` (cannot be extended).
- Implicitly `static` when defined inside another class.
- Extends `java.lang.Enum`.
- Enum constants are **instances** of the enum type (singleton).
- Enums can have **fields, constructors, and methods** (but constructors are always private or package-private).

⚡ 8. Functional Programming in Java

🧩 What Is Functional Programming?

Functional Programming (FP) is a paradigm where programs are constructed using **pure functions** and **immutable data**, emphasizing **declarative logic** instead of step-by-step instructions.

In Java, FP was introduced in **Java 8**, mainly through:

- **Lambda expressions**
- **Method references**
- **Functional interfaces**
- **Stream API**

🧠 Core Functional Programming Concepts

Concept	Description	Example
Pure Function	Same input → same output, no side effects	`int add(int a, int b)`
Immutability	Data cannot change after creation	`String`
Higher-Order Function	Accepts or returns a function	`Stream.map()`
Declarative Style	Express *what* to do, not *how*	`numbers.stream().filter(x -> x > 10)`

🧩 Lambda Expressions

Lambda expressions are **anonymous functions** — they enable you to pass behavior (not just data) as arguments.

Syntax:

```
(parameters) -> expression
```

or

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

💡 Example: Comparator Before and After Lambda

🕰️ Before Java 8

```
Collections.sort(list, new Comparator<String>() {
    public int compare(String a, String b) {
        return a.compareTo(b);
    }
});
```

⚡ After Java 8

```
Collections.sort(list, (a, b) -> a.compareTo(b));
```

🧱 Lambda with Functional Interface

```
@FunctionalInterface
interface Greeting {
    void say(String name);
}

public class Demo {
    public static void main(String[] args) {
        Greeting greet = (name) -> System.out.println("Hello, " +
name);
        greet.say("Irina");
    }
}
```

✅ Output:

```
Hello, Irina
```

🧠 Lambda Features

- Can be assigned to variables.
- Used to implement functional interfaces.
- Can access **effectively final** variables from enclosing scope.
- No explicit return type needed if it can be inferred.

🧱 Method References

A **method reference** is a shorter way to call an existing method by name instead of writing a lambda.

Syntax:

```
ClassName::methodName
```

💡 Types of Method References

Type	Syntax	Example
Static method	`Class::staticMethod`	`Math::max`
Instance method (specific object)	`obj::instanceMethod`	`System.out::println`
Instance method (any object of type)	`Class::instanceMethod`	`String::toUpperCase`
Constructor	`Class::new`	`ArrayList::new`

Example:

```
List<String> names = Arrays.asList("John", "Jane", "Irina");
names.forEach(System.out::println);
```

Equivalent to:

```
names.forEach(n -> System.out.println(n));
```

🧩 Functional Interfaces

Functional interfaces have **exactly one abstract method** — used as lambda targets.

Common Built-in Functional Interfaces (java.util.function):

Interface	Method	Description
`Predicate<T>`	`boolean test(T t)`	Evaluates a condition
`Function<T, R>`	`R apply(T t)`	Transforms input to output
`Consumer<T>`	`void accept(T t)`	Performs action, no return
`Supplier<T>`	`T get()`	Supplies a value
`UnaryOperator<T>`	`T apply(T t)`	Works on same input/output type
`BiFunction<T, U, R>`	`R apply(T t, U u)`	Takes two inputs

💡 Example: Predicate

```
Predicate<Integer> isEven = x -> x % 2 == 0;
System.out.println(isEven.test(4)); // true
```

💡 Example: Function

```
Function<String, Integer> lengthFn = s -> s.length();
System.out.println(lengthFn.apply("Lambda")); // 6
```

💡 Example: Consumer and Supplier

```
Consumer<String> printer = System.out::println;
printer.accept("Hello, FP!");

Supplier<Double> randomValue = Math::random;
System.out.println(randomValue.get());
```

🌊 Stream API

Introduced in **Java 8**, the **Stream API** allows processing collections in a **declarative** and **functional** way.

A **Stream** is **not a data structure** — it's a **sequence of elements** supporting aggregate operations (map, filter, reduce, etc.).

⚙️ Stream Pipeline

A stream pipeline has three parts:

1. **Source** → Collection, array, string, I/O channel, etc.
2. **Intermediate Operations** → Transform the data (`filter()`, `map()`, `sorted()`); transforms one stream into another stream
3. **Terminal Operation** → Produces a result (`collect()`, `forEach()`, `reduce()`, `count()`)

Example:

```
List<String> names = Arrays.asList("Irina", "Mila", "Ana", "Marko");

names.stream()
     .filter(n -> n.length() > 3)
     .map(String::toUpperCase)
     .sorted()
     .forEach(System.out::println);
```

🧩 Stream Operations Overview

Operation	Type	Description
`filter(Predicate)`	Intermediate	Filters elements

`map(Function)`	Intermediate	Transforms elements
`distinct()`	Intermediate	Removes duplicates
`sorted()`	Intermediate	Sorts elements
`limit(n)` / `skip(n)`	Intermediate	Limits or skips elements
`forEach(Consumer)`	Terminal	Performs action for each
`collect(Collector)`	Terminal	Gathers results
`reduce()`	Terminal	Combines elements into one
`count()`	Terminal	Returns number of elements in the stream
`max()`	Terminal	Returns maximum of elements in the stream

💡 Example: Reduce

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5);
int sum = nums.stream().reduce(0, (a, b) -> a + b);
System.out.println(sum); // 15
```

💡 Collectors

Use `Collectors` to transform a stream into a collection or a single value.

```
List<String> names = Arrays.asList("Ana", "Bojan", "Marko");
String joined = names.stream()
                     .collect(Collectors.joining(", "));
System.out.println(joined);
```

💡 Parallel Streams

Parallel streams use multiple threads for faster processing on large data sets.

```
list.parallelStream()
    .filter(n -> n > 10)
    .forEach(System.out::println);
```

⚠️ Parallel streams are **not always faster** — use only when data size is large and operations are independent (stateless).

🧱 Stream Examples Recap

Goal	Example
Filter	`stream.filter(x -> x > 0)`
Map	`stream.map(String::length)`
Sort	`stream.sorted()`
Limit	`stream.limit(5)`
Collect	`stream.collect(Collectors.toList())`
Count	`stream.count()`

✅ Stateless vs Stateful Stream Operations (Java Streams)

🔹 Stateless Operations

A **stateless** intermediate operation does **not need to remember anything about previously processed elements**.
Each element is processed **independently**.

Examples:

- `map()`
- `filter()`

- `flatMap()`
- `peek()`
- `mapToInt()` / `mapToLong()`

Why “stateless”?

Because when processing an element, it **doesn’t depend on**:

- what came before
- what will come after
- any shared state across elements

⭐ **Characteristics:**

- Fast
- Easy to parallelize
- No need for internal buffers

Simple example:

```
list.stream()
    .filter(x -> x > 10)
    .map(x -> x * 2)
```

Each element is processed without storing previous ones.

🔹 Stateful Operations

A **stateful** intermediate operation **must maintain state** (store elements, compare elements, look back or ahead).

Examples:

- `sorted()` → needs all elements to sort
- `distinct()` → tracks all previously seen elements
- `limit()` and `skip()` → need counters
- `takeWhile()` / `dropWhile()` → context dependent

Why "stateful"?

Because these operations **cannot be computed one element at a time** without knowledge of:

- all other elements (`sorted`, `distinct`)
- position in the stream (`limit`, `skip`)

⭐ **Characteristics:**

- Usually slower
- Can require buffering of some or all elements
- Parallelization may require combining states from substreams

Example:

```
list.stream()
    .distinct()     // must track all unique elements seen so far
    .sorted()       // must collect all items before producing output
```

`sorted()` cannot output anything until it **has seen the entire stream**.

🔹 Summary Table

Operation Type	Requires remembering past elements?	Examples	Performance

Stateless	❌ No	`map`, `filter`, `flatMap`	Fast, fully parallelizable
Stateful	✔ Yes	`sorted`, `distinct`, `limit`, `skip`	Slower, may buffer data

🔹 How to explain in 1–2 sentences (interview)

Stateless operations process each element independently and do not depend on previously processed elements (e.g., `map`, `filter`).
Stateful operations need to maintain information about other elements, which may require buffering or seeing the full stream (e.g., `sorted`, `distinct`, `limit`).

🧰 9. Miscellaneous Java Topics

⚙️ Java Development and Build Tools

🧩 Maven

Maven is a **build automation and dependency management tool** for Java projects.
It simplifies project setup, builds, testing, and dependency resolution.

🧱 Key Concepts

Concept	Description
POM (Project Object Model)	XML file (`pom.xml`) describing project structure, dependencies, plugins, etc.
Dependencies	External libraries that Maven downloads automatically.
Repository	Central or local storage for dependencies.

Phases	Build lifecycle steps like `compile`, `test`, `package`, `install`, `deploy`.

💡 Example: `pom.xml`

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>demo</artifactId>
  <version>1.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
      <version>3.0.0</version>
    </dependency>
  </dependencies>
</project>
```

🔧 Common Maven Commands

Command	Purpose
`mvn clean`	Deletes the `target` directory
`mvn compile`	Compiles project source code
`mvn test`	Runs unit tests
`mvn package`	Packages compiled code into a JAR/WAR
`mvn install`	Installs artifact into local repository
`mvn deploy`	Uploads artifact to remote repository

🧩 Gradle (Alternative to Maven)

Gradle is a modern, flexible build tool that uses **Groovy** or **Kotlin DSL** instead of XML.

Advantages:

- Faster incremental builds.
- More readable configuration.
- Supports custom build scripts.

Example (`build.gradle`):

```
plugins {
    id 'java'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation
'org.springframework.boot:spring-boot-starter-web:3.0.0'
}
```

🧠 JDBC — Java Database Connectivity

JDBC provides APIs for connecting and executing queries in a relational database.

⚙️ Steps to Use JDBC

1. **Import packages**

```
import java.sql.*;
```

2. **Register driver**

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

3. **Establish connection**

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

4. **Create statement and execute**

```
Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("SELECT * FROM users");
while (rs.next()) {
    System.out.println(rs.getString("username"));
}
```

5. **Close connection**

```
con.close();
```

🧩 `Statement` vs `PreparedStatement`

Feature	`Statement`	`PreparedStatement`
SQL Injection	Vulnerable	Safe
Performance	Slower	Faster (precompiled)
Parameters	Static	Uses `?` placeholders

Example:

```
PreparedStatement ps = con.prepareStatement(
    "SELECT * FROM users WHERE id = ?");
ps.setInt(1, 5);
ResultSet rs = ps.executeQuery();
```

🧠 Version Control with Git

Git is a **distributed version control system** used to track changes in source code.

🔧 Common Git Commands

Command	Description
`git init`	Initialize a new repository
`git clone <url>`	Clone existing repo
`git status`	Show current changes
`git add <file>`	Stage file for commit
`git commit -m "msg"`	Commit staged changes
`git push`	Upload to remote repository
`git pull`	Download latest changes
`git branch`	List or create branches
`git checkout <branch>`	Switch branches
`git merge <branch>`	Merge branch changes

🐳 Docker and Java

Docker allows you to package Java applications with all dependencies into containers for consistent deployment.

🧩 Example Dockerfile

```
FROM openjdk:17
WORKDIR /app
```

```
COPY target/myapp.jar /app/myapp.jar
CMD ["java", "-jar", "myapp.jar"]
```

Build and run:

```
docker build -t myapp .
docker run -p 8080:8080 myapp
```

✅ This ensures your Java app runs identically on any system.

🧱 JAR vs WAR Files

Type	Full Form	Used For	Deployed On
JAR	Java ARchive	Standalone apps	Java runtime
WAR	Web ARchive	Web applications	Application servers (Tomcat, Jetty)

🧠 Logging in Java

Logging helps track runtime information, warnings, and errors.

Common Frameworks:

- `java.util.logging`
- **Log4j**
- **SLF4J** (Lombok: @Slf4j)
- **Logback**

Example with SLF4J:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
public class Example {
    private static final Logger logger =
LoggerFactory.getLogger(Example.class);

    public static void main(String[] args) {
        logger.info("App started");
        logger.error("Something went wrong");
    }
}
```

Or, with Lombok:

```
import lombok.extern.slf4j.Slf4j;

@Slf4j
public class Example {

    public static void main(String[] args) {
        logger.info("App started");
        logger.error("Something went wrong");
    }
}
```

🧩 Unit Testing (JUnit)

JUnit is a testing framework for writing repeatable unit tests.

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class MathUtilsTest {
    @Test
    void testAdd() {
        assertEquals(4, Math.add(2, 2));
    }
}
```

Run tests via IDE, Maven, or Gradle.

🧩 Java 8+ Key Features Recap

Version	Major Features
Java 8	Lambdas, Streams, Optional, Date/Time API
Java 9	Modules, private methods in interfaces
Java 10	`var` keyword for local inference
Java 11	HTTP Client API, String enhancements
Java 14+	Switch expressions, records
Java 17 (LTS)	Pattern matching, sealed classes

🔍 10. Java Modern Features

✅ 1. Records (Java 16/17 LTS)

What is a record?

A **record** is a special kind of class in Java designed to hold data.
Think of it as a **data container** with:

- fields
- a constructor (canonical - with all fields)
- getters

- `equals()`
- `hashCode()`
- `toString()`

All generated automatically.

Where can a record be applied?

- It is a **class** (not a method, not an annotation).
- It can be declared:
    - as a **top-level type**
    - or as an **inner/nested type**

Example:

```
public record Point(int x, int y) { }
```

Important properties of records:

1. Fields are **implicitly** `private final`
   → once created, the values **cannot change** (or for reference types - the reference cannot be reassigned).
2. Records are **implicitly** `final` - we can not extend record
3. Records **can't extend other classes**
   (because they already implicitly extend `java.lang.Record`).
4. They **can implement interfaces**.
5. They can contain methods and static fields, but **can't define additional instance fields**.
6. Records are **final by design**, but fields can reference mutable objects (e.g. `public record User(List<String> roles) {}`)

Example with custom logic:

```
public record User(String name, int age, List<String> roles) {
      // compact canonical constructor
      public User {
              if (age < 0) throw new IllegalArgumentException("Age must
      be positive");

             // making field immutable
             roles = List.copyOf(roles);

           // cannot assign the fields in compact constructor
          }

      // custom constructor must call canonical constructor
      public User(String name) {
              this(name, 0, List.of(“USER”));
          }

      public String greeting() {
              return "Hello " + name;
          }
}
```

When should you use records?

Whenever your class only holds data and doesn't need complex behavior:

- DTOs
- API responses
- Config objects
- Value objects (like Point, Money, Range…)

✅ 2. Pattern Matching

Pattern matching helps Java check types AND extract values in one step.

2a. Pattern Matching for `instanceof`

Before Java 16:

```
if (obj instanceof String) {
    String s = (String) obj;  // manual cast
    System.out.println(s.length());
}
```

After Java 16:

```
if (obj instanceof String s) {
    System.out.println(s.length());
}
```

What changed?

- `instanceof` no longer only checks the type.
- It can also automatically **declare a variable of that type** (`s` in this case).
- Cast is done **implicitly** and safely.

Why is this helpful?

Because previously you often repeated yourself:

1. Check type
2. Cast
3. Use

Now steps 1 and 2 are combined.

2b. Pattern Matching for `switch` (Java 17+)

What is it?

Java now lets you match different types or patterns inside a `switch`.

Example:

```
switch (obj) {
    case String s -> System.out.println("String length: " +
s.length());
    case Integer i -> System.out.println("Number doubled: " + i * 2);
    case null -> System.out.println("It's null!");
    default -> System.out.println("Unknown");
}
```

How does it work?

Each case **checks the type** AND **creates a variable** if the type matches.

✅ 3. Sealed Classes (Java 17)

What are sealed classes?

A **sealed class** (or interface) lets you explicitly control who is allowed to extend (or implement) it.

Normal class: ANY subclass can be created somewhere in the project.

Sealed class: ONLY listed subclasses are allowed.

Example:

```
public sealed class Shape permits Circle, Triangle, Rectangle {}

public final class Circle extends Shape {}
public non-sealed class Triangle extends Shape {}
public sealed class Rectangle extends Shape permits Square {}
```

📌 What subclasses MUST declare

Every permitted subclass must be one of:

✅ `final`

(no one can extend it further)

```
public final class Circle extends Shape {}
```

✅ `sealed`

(can continue restricting hierarchy)

```
public sealed class Rectangle extends Shape
        permits Square {}
```

✅ `non-sealed`

(opens hierarchy again)

```
public non-sealed class Triangle extends Shape {}
```

❗ This is mandatory

You MUST explicitly choose one.

This is illegal:

```
public class Circle extends Shape {} // ❌ compile error
```

Compiler forces you to declare inheritance intent.

Why is this useful?

For safe and controlled hierarchies — when you want:

- only specific allowed shapes

- closed domain models
- switching over all possible types

It works **perfectly together with pattern matching** because Java knows the whole hierarchy.

✅ 4. Switch Expressions (Java 14+)

Traditional switch (old):

- Only a *statement* (cannot return a value).
- Must use `break` to avoid fall-through.
- Verbose.

New switch expression:

Works like an expression: it **produces a value**.

```
int result = switch (day) {
    case MONDAY, FRIDAY -> 6;
    case TUESDAY -> 7;
    default -> 0;
};
```

What is `yield`?

`yield` is used inside a multi-line switch block to **return a value**.

Example:

```
String result = switch (obj) {
    case Integer i -> {
        int doubled = i * 2;
        yield "Double: " + doubled;  // return value from block
    }
    case String s -> "String: " + s;
```

```
    default -> "Other";
};
```

Why not use `return`?

Because `return` would return from the **method**, not the switch.

`yield` returns from the **switch expression**.

✅ 5. `var` & Local Type Inference (Java 10+)

What does `var` do?

Allows Java to **infer the type** of a local variable.

```
var text = "hello";          // inferred String
var number = 10;             // inferred int
var list = new ArrayList<String>();  // inferred ArrayList<String>
```

Important rules:

- Only for **local variables** inside methods, not fields.
- The type is still **static and known at compile time**.
- The variable still has a concrete type — just hidden for readability.

Why is this useful?

- Shortens long generic types
- Makes code cleaner
- Encourages good variable names (since type is not visible)

Bad example:

```
var x = doSomething(); // what is x???
```

Good example:

```
var productMap = new HashMap<String, Product>();
```

⭐ Summary: Why these features exist

Feature	Why it exists	What problem it solves
Records	Simpler data classes - immutable	Removes boilerplate getters/equals/hashCode
Pattern matching	Safer type checks and casts	Avoids repeating `obj instanceof` + cast
Sealed classes	Safer inheritance control	Defines exactly which classes can extend
Switch expressions	More powerful switch logic	Switch that returns values, no fall-through
`yield`	Return from switch expression	Needed for multi-line switch blocks
`var`	Cleaner, shorter locally scoped code	Removes noisy type declarations

TODO: Reactive Java, Reactive Streams