

JVM

JVM stands for Java Virtual Machine.

It is a ***software-based engine*** that runs Java programs.

It does not understand Java source code — instead, it executes bytecode (.class files).

It provides a runtime environment in which Java bytecode can be executed, regardless of the underlying operating system.

頻 In simple words:

You write Java code → compile it → JVM runs it on any OS.

That's why Java is platform-independent.

□ How Java Program Runs (Java Execution Flow)

Write source code

You write a .java file (e.g., Hello.java).

Compile with javac

The Java Compiler (javac) converts .java → .class file (bytecode).

Example:

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

After javac Hello.java → Hello.class (bytecode).

JVM loads bytecode

The Class Loader loads .class files into memory.

Bytecode verification

Bytecode Verifier checks the code for security (no illegal code, stack overflow, etc.).

Execution by JVM

The Execution Engine runs the bytecode. This can happen in two ways:

Interpreter: Reads and executes bytecode line by line (slow).

JIT Compiler (Just-In-Time): Compiles frequently used bytecode into native machine code for faster execution.

Runtime support

JVM interacts with the Java Runtime Environment (JRE), which includes class libraries, garbage collector, memory manager, etc.

□ JVM Architecture (Major Components)

Class Loader Subsystem

Loads class files into JVM memory.

Three types:

Bootstrap ClassLoader → loads core Java classes (java.lang.*).

Extension ClassLoader → loads classes from jre/lib/ext.

Application ClassLoader → loads user-defined classes.

Runtime Data Areas (Memory in JVM)

Method Area: Stores class metadata (code for methods, fields, static variables).

Heap: Stores objects and instance variables (shared across threads).

Stack: Each thread has its own stack → stores method calls, local variables.

Program Counter Register (PC Register): Stores address of current executing instruction.

Native Method Stack: For native (non-Java) methods.

Execution Engine

Interpreter → executes bytecode line by line.

JIT Compiler → compiles frequently used code into native machine code (fast).

Garbage Collector → automatically manages memory by cleaning unused objects.

Native Interface

Provides a way to call code written in other languages like C/C++.

Native Method Libraries

Libraries written in native languages required by JVM.

□ Simple Diagram (Textual)

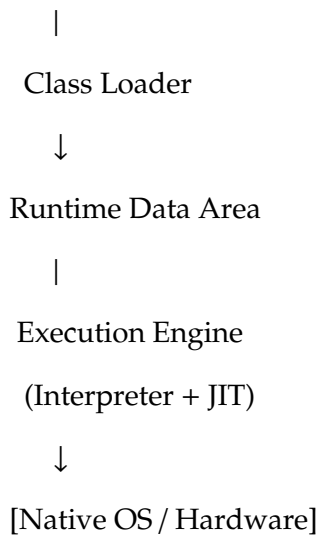
[Your Java Code] (.java)

|

javac

↓

[Bytecode] (.class)



□ Key Benefits of JVM

Platform independence → same .class file runs on Windows, Linux, Mac.

Memory management → automatic Garbage Collection.

Security → bytecode verifier + sandboxing.

Performance → JIT compiler improves execution speed.

✓ Summary:

JVM is a virtual machine that takes compiled Java bytecode and translates it into machine code so it can run on any platform. It handles class loading, memory management, security, and execution of programs.

explain the difference between JDK, JRE, and JVM?

JVM (Java Virtual Machine)

The engine that runs Java bytecode.

It doesn't understand .java files, only .class bytecode files.

Handles:

Loading classes

Memory management (Heap, Stack, Garbage Collection)

Bytecode execution (Interpreter + JIT)

顯 Think of JVM as the driver who actually runs your program.

□ JRE (Java Runtime Environment)

Provides everything needed to run Java applications.

Includes:

JVM

Core libraries (java.lang, java.util, etc.)

Supporting files

Does not include the compiler (javac).

So with JRE, you can run Java programs, but you cannot develop them.

顯 Think of JRE as the car (with engine inside) ready to drive.

□ JDK (Java Development Kit)

Full package for Java developers.

Includes:

JRE (so it has JVM + libraries)

Compiler (javac)

Development tools (javadoc, javap, debuggers, etc.)

With JDK, you can write, compile, and run Java programs.

顯 Think of JDK as the garage + car + tools needed to build, maintain, and drive.

% Relationship Diagram

JDK (Development Kit)

├─ JRE (Runtime Environment)

| └─ JVM (Virtual Machine)

| └─ Core Libraries

└─ Development Tools (javac, javadoc, etc.)

✓ Quick Table

Feature	JVM	JRE	JDK
What it is	Virtual machine that executes bytecode	Runtime environment to run Java apps	
	Development kit to develop + run Java apps		
Contains tools	Execution engine, GC, classloader	JVM + libraries	JRE + compiler + dev
Used for	Running Java bytecode	Running Java programs	Developing and running Java programs
Can you compile code?		✗ No	✗ No ✓ Yes
Can you run code?	✓ Yes	✓ Yes	✓ Yes

顯 Simple analogy:

JVM = Engine

JRE = Car (engine + fuel + parts)

JDK = Car + Mechanic tools

□ String-based Questions

Reverse a string without using built-in methods.

Check if a string is a palindrome.

Find duplicate characters in a string.

Count the occurrence of each character in a string.

Find the first non-repeating character in a string.

Remove all vowels from a string.

Reverse each word in a sentence.

Anagram check (e.g., "listen" and "silent").

□ Array-based Questions

Find the second largest number in an array.

Find missing number in an array of 1 to n.

Find duplicates in an array.

Find the sum of elements in an array.

Rotate an array by k steps.

Merge two sorted arrays into one sorted array.

Find the pair of numbers whose sum is equal to a given number.

□ Collections & Java 8 (important!)

Sort a list of integers using Java 8 Streams.

Convert a list of strings to uppercase using map().

Find the maximum and minimum from a list using Stream.

Count how many times each word appears in a list of strings.

Find all even numbers from a list.

Convert a list to a Map using Java 8.

Difference between map() and flatMap() (with coding example).

□ Number Problems

Check if a number is prime.

Find factorial of a number.

Fibonacci series up to n terms.

Check if a number is Armstrong.

Reverse a number.

Find the sum of digits in a number.

□ SQL-Related (if asked in coding round)

Write a query to get the second highest salary.

Write a query to fetch employees whose name starts with "A".

Query to count number of employees department-wise.

Query to find duplicate records in a table.

□ Scenario-based Coding

You have a list of employees → group them by department using Streams.

Sort employees by salary (ascending & descending).

Find the highest paid employee in each department.

Given a string of numbers "11223344" → compress it to "1234".

Write a program to implement your own version of HashMap.

顯 For your level (2.4 years), companies usually mix:

1–2 core Java coding questions (string/array).

1 problem with Java 8 Streams.

Some SQL queries.

1 scenario from real-time project.

Toughest Interview includes basic to advance concepts with real time examples, "Tricky" Questions: These often involve edge cases, subtle differences between similar concepts (== vs. .equals(), String vs. StringBuilder vs. StringBuffer), or questions designed to probe your analytical thinking rather than just recall. Examples include questions about immutable objects, static methods, or custom classloaders.

Java 8 features, Collection framework and spring boot, other. SQL, cloud if u mention in resume. Managerial round - day to day activities. Jira, Confluence, Agile methodologies and other tools. Hr round with salary discussion.

Technical Round: Q1. Explain the Spring Security. Q2. Explain the flow of a Rest API call from frontend to backend and response back to frontend. Q3. Explain JWT token. Q4. Define Filters and Segmentation in Spring Security. Q5. How one microservices authenticate another microservice with jwt? Q6. What is a functional interface? Q7. Give an example of a java interface which uses a functional interface. Q8. Write a code to implement Runnable using lambda. Q9. Which is springboot default server and How to use another server in srpingboot? Q10. What are profiles and how to use them? Q11. How is bean Injection working? Q12. How to inject a specific bean of the same object? Q13. How to set up a discovery server? Like Eureka Q14. What is the use of @Primary annotation? Q15. Which method can we use to check if a service is up or not? Q16. How would you externalize a microservice? Q17. What is the default port of springboot? Q18. What is an optional class and its use? Q19. Optional.Of() vs Optional.ofNullable() Q20. Write sql query to return 2nd highest salary. Q21. What is a Rest template?

Type: Technical Interview (Initial Round) Mode: Likely virtual or telephonic Duration: 20–25 minutes 六淺 1. Introduction Brief self-introduction was taken. I shared my background, experience, and core technical strengths (e.g., Java, Spring Boot, REST APIs). 泉 2. Coding Question A stream-based coding question was given 摠 3. Database Questions Basic to intermediate questions around SQL or PostgreSQL

Interview was 3 rounds, first one was Face-To-Face, Second round was technical a bit, third was managerial round .first round was regarding technical like oops concept, logging , spring boot , microservices, and asked about static and final . Managerial was regarding ur project details , what are the aspects you have accomplished from your end.

What exactly static and final keywords, oops concept and explain springboot and microservices.

What is a REST API?

REST = Representational State Transfer.

It is a set of rules/architecture style **to build web services that allow communication between systems using HTTP** (just like your browser does).

A REST API is an API that follows REST principles.

Data is represented as resources (like users, books, orders).

Each resource can be accessed using URLs (endpoints).

You use HTTP methods to interact:

GET → fetch data

POST → create new data

PUT → update data

DELETE → remove data

REST APIs usually return data in JSON format (sometimes XML).

□ Example (Practical in Spring Boot)

Suppose we have a resource called Book.

We want to build REST APIs to manage books in a library.

1) Book Model (Entity)

```
public class Book {  
    private Long id;  
    private String title;  
    private String author;  
  
    // constructor, getters, setters  
}
```

2) REST Controller

```
import org.springframework.web.bind.annotation.*;  
import java.util.*;  
  
@RestController  
@RequestMapping("/api/books")  
public class BookController {  
  
    private List<Book> books = new ArrayList<>();
```


// Constructor: add sample books

```
public BookController() {  
    books.add(new Book(1L, "Java Basics", "Author A"));  
    books.add(new Book(2L, "Spring Boot Guide", "Author B"));  
}
```

// GET: fetch all books

```
@GetMapping  
public List<Book> getAllBooks() {  
    return books;  
}
```

// GET: fetch book by ID

```
@GetMapping("/{id}")  
public Book getBookById(@PathVariable Long id) {  
    return books.stream()  
        .filter(b -> b.getId().equals(id))  
        .findFirst()  
        .orElse(null);  
}
```

// POST: add a new book

```
@PostMapping  
public Book addBook(@RequestBody Book book) {  
    books.add(book);  
    return book;  
}
```

// PUT: update a book

```
@PutMapping("/{id}")  
public Book updateBook(@PathVariable Long id, @RequestBody Book bookDetails) {
```

```

for (Book book : books) {
    if (book.getId().equals(id)) {
        book.setTitle(bookDetails.getTitle());
        book.setAuthor(bookDetails.getAuthor());
        return book;
    }
}
return null;
}

```

// DELETE: delete a book

```

@DeleteMapping("/{id}")
public String deleteBook(@PathVariable Long id) {
    books.removeIf(b -> b.getId().equals(id));
    return "Book deleted with id " + id;
}
}

```

3) How it works (API endpoints)

HTTP Method	Endpoint	What it does
GET	/api/books	Get all books
GET	/api/books/1	Get book with ID = 1
POST	/api/books	Add a new book (send JSON body)
PUT	/api/books/1	Update book with ID = 1
DELETE	/api/books/1	Delete book with ID = 1

4) Example using JSON

➔ POST Request (Add Book)

POST /api/books

Content-Type: application/json

```
{
```

```
"id": 3,  
"title": "Hibernate in Action",  
"author": "Author C"  
}
```

✓ Response

```
{  
  "id": 3,  
  "title": "Hibernate in Action",  
  "author": "Author C"  
}
```

➔ GET Request (All Books)

GET /api/books

✓ Response

```
[  
  {"id":1,"title":"Java Basics","author":"Author A"},  
  {"id":2,"title":"Spring Boot Guide","author":"Author B"},  
  {"id":3,"title":"Hibernate in Action","author":"Author C"}  
]
```

□ In simple words:

A REST API lets two systems talk over the internet using standard HTTP.

In Java + Spring Boot, you build REST APIs with annotations like:

@RestController → marks class as REST API controller.

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping → define endpoints.

LAMBDA EXPRESSION

A lambda expression is a short way to write an implementation of a functional interface (an interface with exactly one abstract method). Instead of writing a full anonymous class, you give the parameters and the body directly:

(params) -> expression or (params) -> { statements }.

Example 1 — Sorting strings (no lambda vs lambda)

Without lambda (anonymous inner class)

```
import java.util.*;

public class SortWithoutLambda {

    public static void main(String[] args) {

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

        // old style: anonymous inner class implementing Comparator
        Collections.sort(names, new Comparator<String>() {

            @Override

            public int compare(String a, String b) {

                return a.compareTo(b);

            }

        });

        System.out.println(names); // [Alice, Bob, Charlie]

    }

}
```

What's happening (step-by-step):

Collections.sort needs a Comparator<String>.

We create a one-off anonymous class new Comparator<String>() { ... }.

That class must implement the method compare(String a, String b).

Inside compare we write the comparison logic.

Java constructs the comparator object and uses it to sort.

With lambda

```
import java.util.*;
```

```

public class SortWithLambda {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>(Arrays.asList("Bob", "Alice", "Charlie"));
        // lambda: (a, b) -> a.compareTo(b)
        names.sort((a, b) -> a.compareTo(b));
        // even shorter using method reference:
        // names.sort(String::compareTo);
        System.out.println(names); // [Alice, Bob, Charlie]
    }
}

```

What's happening now (step-by-step):

`names.sort` expects a `Comparator<String>`.

`(a, b) -> a.compareTo(b)` is a lambda that implements `compare(a,b)` for us.

Java infers types (`String`) from the context — you don't write them.

The lambda is converted to an instance of `Comparator` behind the scenes and used to sort.

Why shorter? The anonymous class required a lot of boilerplate (`new Comparator... @Override`) — the lambda replaces it with a single expression.

SOLID PRINCIPLE IN JAVA

SOLID Principles in Java

S → Single Responsibility Principle (SRP)

O → Open/Closed Principle (OCP)

L → Liskov Substitution Principle (LSP)

I → Interface Segregation Principle (ISP)

D → Dependency Inversion Principle (DIP)

☐ Single Responsibility Principle (SRP)

Definition: A class should have only one reason to change, i.e., one responsibility.

✗ Bad Example:

```
class Report {  
    public String generateReport() { return →Report Data→; }  
    public void saveToFile(String data) { /* file I/O */ }  
}
```

Here, Report does two things: generating a report + saving it to a file.

✓ Good Example

```
class Report {  
    public String generateReport() { return →Report Data→; }  
}  
  
class FileSaver {  
    public void saveToFile(String data) { /* file I/O */ }  
}
```

Now, Report is only responsible for report generation.

FileSaver handles saving.

Open/Closed Principle (OCP)

Definition: Classes should be open for extension, but closed for modification.

✗ Bad Example:

```
class AreaCalculator {  
    public double calculate(Object shape) {  
        if (shape instanceof Circle) {  
            Circle c = (Circle) shape;  
            return Math.PI * c.radius * c.radius;  
        } else if (shape instanceof Rectangle) {  
            Rectangle r = (Rectangle) shape;  
            return r.length * r.width;  
        }  
    }  
}
```

```

        return 0;
    }
}

```

Every time a new shape is added, we must modify the class.

✓ Good Example (using polymorphism):

```

interface Shape {
    double area();
}
class Circle implements Shape {
    double radius;

    Circle(double r) { this.radius = r; }

    public double area() { return Math.PI * radius * radius; }
}

class Rectangle implements Shape {
    double length, width;

    Rectangle(double l, double w) { this.length = l; this.width = w; }

    public double area() { return length * width; }
}

class AreaCalculator {
    public double calculate(Shape shape) {
        return shape.area(); // open for extension (new shapes), closed for modification
    }
}

```

If we add Triangle, no need to modify AreaCalculator.

Liskov Substitution Principle (LSP)

Definition: Subclasses should be substitutable for their parent class without breaking functionality.

✗ Bad Example:

```

class Bird {
    void fly() { System.out.println(→Flying→); }
}

```

```
}
```

```
class Ostrich extends Bird {
```

```
    @Override
```

```
    void fly() { throw new UnsupportedOperationException("→Ostrich can't fly→"); }
```

```
}
```

Ostrich breaks substitution — it violates the expectation that all Birds can fly.

✓ Good Example:

```
interface Bird { void eat(); }
```

```
interface FlyableBird extends Bird { void fly(); }
```

```
class Sparrow implements FlyableBird {
```

```
    public void eat() { System.out.println("→Sparrow eats→"); }
```

```
    public void fly() { System.out.println("→Sparrow flies→"); }
```

```
}
```

```
class Ostrich implements Bird {
```

```
    public void eat() { System.out.println("→Ostrich eats→"); }
```

```
}
```

Now, Ostrich doesn't incorrectly override fly().

☐ Interface Segregation Principle (ISP)

Definition: No client should be forced to implement interfaces it doesn't use.

✗ Bad Example:

```
interface Worker {
```

```
    void work();
```

```
    void eat();
```

```
}
```

```
class Robot implements Worker {
```

```
    public void work() { System.out.println("→Robot works→"); }
```

```
    public void eat() { /* Not needed, but forced */ }
```



```
}
```

✓ Good Example:

```
interface Workable { void work(); }
```

```
interface Eatable { void eat(); }
```

```
class Human implements Workable, Eatable {
```

```
    public void work() { System.out.println(→Human works→); }
```

```
    public void eat() { System.out.println(→Human eats→); }
```

```
}
```

```
class Robot implements Workable {
```

```
    public void work() { System.out.println(→Robot works→); }
```

```
}
```

Each class implements only what it needs.

Dependency Inversion Principle (DIP)

Definition: Depend on abstractions, not concrete classes.

✗ Bad Example:

```
class MySQLDatabase {
```

```
    public void connect() { System.out.println(→Connected to MySQL→); }
```

```
}
```

```
class Application {
```

```
    private MySQLDatabase db = new MySQLDatabase(); // tightly coupled
```

```
    public void start() { db.connect(); }
```

```
}
```

If we switch to Oracle, we must modify Application.

✓ Good Example:

```
interface Database {
```

```
void connect();  
}
```

```
class MySQLDatabase implements Database {  
    public void connect() { System.out.println(→Connected to MySQL→); }  
}
```

```
class OracleDatabase implements Database {  
    public void connect() { System.out.println(→Connected to Oracle→); }  
}
```

```
class Application {  
    private Database db;  
    Application(Database db) { this.db = db; } // dependency injection  
    public void start() { db.connect(); }  
}
```

Application works with any Database implementation.

□ Quick Summary (for interviews)

S: One class = One responsibility.

O: Extend behavior without modifying existing code.

L: Subtypes must be replaceable without breaking code.

I: Many small interfaces > one big interface.

D: Depend on abstractions, not concrete classes.

Q: Can you override a private or static method in Java?

A: No, You cannot override a private or static method in Java because :

1. A private method cannot be overridden in a subclass because it's not accessible there. If you define one with the same name in the child class, it's just a new method.

2. Static methods can be redefined in the child class, but this is called method hiding, not overriding.

Q: Does finally always execute in Java?

A: Not in the following cases:

1. When System.exit() is called
2. When the JVM/system crashes

Q: What methods does the Object class have?

A: java.lang.Object, the parent of all classes, has the following methods:

1. protected Object clone() throws CloneNotSupportedException → Creates and returns a copy of this object.
2. public boolean equals(Object obj) → Indicates whether some other object is "equal to" this one.
3. protected void finalize() throws Throwable → Called by the garbage collector before object destruction.
4. public final Class<?> getClass() → Returns the runtime class of this object.
5. public int hashCode() → Returns a hash code value for the object.
6. public String toString() → Returns a string representation of the object.
7. public final void wait() / wait(long) / wait(long, int) → Causes the current thread to wait.
8. public final void notify() and notifyAll() → Wakes up threads waiting on this object.

Q: How can you make a class immutable? (This was actually asked to me)

A: In 6 easy steps:

1. Declare the class as final so it can't be extended
2. Make all fields private
3. Don't provide setters
4. Make all mutable fields final
5. Initialize all fields via constructor (with deep copy where needed)
6. In getters, return copies instead of actual references

Q: What is a singleton class in Java and how can we make one? (Imp most asked)

A: A Singleton class allows only one instance at a time in a JVM.

```
public class Animal {  
    private static Animal single_instance = null;
```

```

private Animal() {}

public static Animal getInstance() {
    if (single_instance == null)
        single_instance = new Animal();
    return single_instance;
}
}

```

How exception handling works in Java — explained clearly and in detail?

Java's exception mechanism is how the language reports and reacts to errors and unusual conditions at runtime. I'll cover the what, how, why, and show practical code examples and best practices.

1) The basics — Throwable hierarchy

Object

└─ Throwable

└─ Error (serious problems; usually do NOT catch)

└─ Exception

└─ RuntimeException (unchecked; programmer errors)

└─ (checked exceptions) e.g. IOException

Error: JVM-level problems (OutOfMemoryError). Don't normally catch.

Exception: things you can (or should) handle.

Checked Exceptions (e.g. IOException, SQLException): compiler forces you to handle or declare them (throws).

Unchecked Exceptions (RuntimeException and subclasses, e.g. NullPointerException, IllegalArgumentException): not required to be declared or caught.

2) Throwing and catching — syntax & flow

try / catch / finally

```
try {
```

```
    // code that might throw
```

```
} catch (SpecificException e) {
```

```
    // handle that exception
```

```

} catch (AnotherException e) {
    // handle another
} finally {
    // always executed (except JVM exit, thread kill, or crash)
}

```

Example:

```

try {
    int result = 10 / value; // might throw ArithmeticException
    Files.readAllLines(path); // might throw IOException
} catch (IOException e) {
    System.err.println("I/O problem: " + e.getMessage());
} catch (RuntimeException e) {
    System.err.println("Runtime problem: " + e); // catches NPE, AE, etc.
} finally {
    cleanup(); // always runs (unless JVM terminates)
}

```

How it flows (stack unwinding)

When throw happens, JVM looks for a catch block in the current method that matches the exception type.

If none found, the method exits and the exception propagates to caller (stack frame popped).

This continues up the call stack until a handler is found or the program terminates.

3) throws and throw

throws — declares that a method can propagate checked exceptions:

```
void readFile(String path) throws IOException { ... }
```

throw — actually creates and throws an exception:

```
if (value < 0) throw new IllegalArgumentException("value must be >= 0");
```

4) Checked vs Unchecked — when to use what

Checked: use for recoverable conditions the caller should handle (e.g., IOException).

Unchecked: for programming errors (null pointers, bad arguments). Usually not declared.

Libraries differ — think carefully about API design.

5) Multi-catch and rethrowing

Multi-catch (Java 7+)

```

try {
    // ...
} catch (IOException | SQLException e) {
    // e is effectively final; you cannot reassign it
    log(e);
}

```

Rethrowing and preserving cause

```

try {
    doSomething();
} catch (SQLException e) {
    throw new MyAppException("Database failed", e); // preserve cause
}

```

Use constructors that take the cause so stack traces show the full chain.

6) try-with-resources (automatic resource management)

Use when working with resources that implement `AutoCloseable` (files, streams, DB connections):

```

try (BufferedReader br = new BufferedReader(new FileReader(path))) {
    System.out.println(br.readLine());
} catch (IOException e) {
    e.printStackTrace();
}

```

Resources are closed automatically at the end.

If both the block and `close()` throw exceptions, the `close()` exception is suppressed and attached to the primary exception (accessible via `Throwable.getSuppressed()`).

Example of suppressed exceptions:

```

try (MyCloseable r = new MyCloseable()) {
    throw new RuntimeException("Main");
}

```

// If `r.close()` also throws, that exception will be suppressed.

7) Inspecting exceptions

Common useful methods on `Throwable`:

`getMessage()` — human message.

`printStackTrace()` — prints the stack trace.

`getCause()` — the underlying cause (exception chaining).

`getSuppressed()` — suppressed exceptions from try-with-resources.

8) Custom exceptions

Create domain-specific exceptions when needed:

```
public class PaymentException extends Exception {  
    public PaymentException(String msg) { super(msg); }  
    public PaymentException(String msg, Throwable cause) { super(msg, cause); }  
}
```

Prefer meaningful names and include cause when wrapping lower-level exceptions.

9) Examples (compact)

Simple try/catch

```
try {  
    int x = Integer.parseInt("abc"); // NumberFormatException (unchecked)  
} catch (NumberFormatException e) {  
    System.out.println("Bad number: " + e.getMessage());  
}
```

Declaring checked exception

```
void readFile(String path) throws IOException {  
    Files.readAllLines(Paths.get(path));  
}
```

Rethrow and preserve cause

```
try {  
    someDbCall();  
} catch (SQLException e) {  
    throw new DataAccessException("Failed DB query", e);  
}
```

try-with-resources and suppressed

```
try (BufferedReader br = Files.newBufferedReader(path)) {  
    // read  
} // br.close() called automatically; exceptions suppressed if needed
```

Finally and return behavior

```
int foo() {  
    try {  
        return 1;  
    } finally {  
        System.out.println("finally runs");  
        // if you 'return' here, it overrides the try's return (avoid that)  
    }  
}
```

finally executes even if return in try (unless JVM exits or thread killed). Avoid return in finally.

10) Best practices / dos & don'ts

Do

Catch the most specific exception you can handle.

Use try-with-resources for resource management.

Preserve the cause when wrapping exceptions (new Exception("msg", cause)).

Log exceptions with stack trace (logger.error("failed", e)).

Validate inputs and throw appropriate IllegalArgumentException for bad usage.

Don't

Don't catch Throwable or Error (unless you have a very good reason).

Don't swallow exceptions silently (catch (Exception e) {}) — at least log.

Don't use exceptions for normal control flow (they are expensive and make code hard to read).

Don't return from finally — it hides original exceptions/returns.

11) Performance note

Throwing exceptions is relatively expensive (object creation, filling stack trace). Use exceptions for exceptional conditions, not frequent control decisions.

12) Final summary (short)

Java throws a Throwable object and unwinds the stack searching for a matching catch.

Use try/catch/finally or try-with-resources to handle/clean up.

Prefer specific catches, preserve causes, and avoid catching Error or Throwable.

Design API exception strategy (checked vs unchecked) consciously.

Log and rethrow thoughtfully; don't swallow exceptions.

If you want, I can now:

Show a small complete example that demonstrates throwing, catching, rethrowing with cause, try-with-resources, and suppressed exceptions — all in one runnable snippet. Would you like that?

How do you define custom sorting logic?--> in java explain with example

□ 1. Using Comparable (Natural Sorting)

If a class implements Comparable<T>, it defines its natural ordering by overriding the compareTo() method.

顯 Example: Sort students by their id (ascending)

```
import java.util.*;

class Student implements Comparable<Student> {

    int id;

    String name;

    Student(int id, String name) {

        this.id = id;

        this.name = name;

    }

    // Natural sorting: sort by id

    @Override

    public int compareTo(Student other) {

        return Integer.compare(this.id, other.id);

    }

    @Override

    public String toString() {

        return id + " - " + name;

    }

}

public class Main {

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

```

List<Student> list = Arrays.asList(
    new Student(3, →Nisha→),
    new Student(1, →Amit→),
    new Student(2, →Raj→)
);

Collections.sort(list); // uses compareTo()

System.out.println(list);
}
}

```

✓ Output:

[1 - Amit, 2 - Raj, 3 - Nisha]

□ 2. Using Comparator (Custom Sorting Logic)

If you want multiple different sort orders, use `Comparator<T>`.

顯 Example: Sort students by name (alphabetically).

```

import java.util.*;

class Student {
    int id;

    String name;

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
    public String toString() {
        return id + → - → + name;
    }
}

public class Main {
    public static void main(String[] args) {

```

```

List<Student> list = Arrays.asList(
    new Student(3, →Nisha→),
    new Student(1, →Amit→),
    new Student(2, →Raj→)
);

// Custom sorting by name
Comparator<Student> byName = (s1, s2) -> s1.name.compareTo(s2.name);
Collections.sort(list, byName);
System.out.println(list);
}
}

```

✓ Output:

[1 - Amit, 3 - Nisha, 2 - Raj]

□ 3. Using Streams + Comparator (Modern Java ←+)

Instead of Collections.sort(), you can use Streams.

顯 Example: Sort students by name length

```

import java.util.*;
import java.util.stream.Collectors;

class Student {
    int id;
    String name;
    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
    @Override
    public String toString() {
        return id + → - → + name;
    }
}

```

```

    }
}

public class Main {

    public static void main(String[] args) {

        List<Student> list = Arrays.asList(

            new Student(3, →Nisha→),

            new Student(1, →Amit→),

            new Student(2, →Rajesh→)

        );

        List<Student> sorted = list.stream()

            .sorted(Comparator.comparingInt(s -> s.name.length()))

            .collect(Collectors.toList());

        System.out.println(sorted);

    }

}

```

✓ Output:[1 - Amit, 3 - Nisha, 2 - Rajesh]

□ 4. Summary

Comparable → Use when the class has a default (natural) sorting logic.

Comparator → Use when you want different custom sorting rules (flexible).

Java 8+ → Use Comparator.comparing(), reversed(), thenComparing() for chaining.

顯 Example with chained sorting: sort by name, then by id

```

list.sort(

    Comparator.comparing((Student s) -> s.name)

        .thenComparing(s -> s.id)

);

```

Q.Difference between string,stringbuilder,stringbuffer in java?

1. String

Immutable: Once a String object is created, its value cannot be changed. Any modification (concatenation, replace, etc.) creates a new object in memory.

Stored in the String Constant Pool (SCP) for reusability.

Thread-safety: inherently safe since it's immutable.

Performance: Slower when doing many modifications (because it creates new objects).

📄 Example:

```
public class StringExample {  
    public static void main(String[] args) {  
        String s1 = →Hello→;  
        String s2 = s1.concat(→ World→);  
  
        System.out.println(s1); // Hello (unchanged)  
        System.out.println(s2); // Hello World (new object)  
    }  
}
```

2. **StringBuilder**

Mutable: Can be modified without creating new objects.

Not thread-safe: No synchronization → better performance in single-threaded applications.

Performance: Faster than String and StringBuffer when frequent modifications are needed.

📄 Example:

```
public class StringBuilderExample {  
    public static void main(String[] args) {  
        StringBuilder sb = new StringBuilder(→Hello→);  
        sb.append(→ World→);  
        System.out.println(sb); // Hello World (same object modified)  
    }  
}
```

3. **StringBuffer**

Mutable: Just like StringBuilder.

Thread-safe: Methods are synchronized → safe in multi-threaded environments.

Performance: Slower than StringBuilder because of synchronization overhead.

顯 Example:

```
public class StringBufferExample {  
    public static void main(String[] args) {  
        StringBuffer sb = new StringBuffer(→Hello→);  
        sb.append(→ World→);  
        System.out.println(sb); // Hello World (same object modified)  
    }  
}
```

% Comparison Table

Feature	String (Immutable)	StringBuilder (Mutable)	StringBuffer (Mutable)
Mutability	✗ Immutable	✓ Mutable	✓ Mutable
Thread-Safe	✓ (immutable)	✗ Not thread-safe	✓ Thread-safe (sync)
Performance (single-thread)	Slowest (new object creation)	Fastest (no sync)	Slower than StringBuilder
Use Case	Constant values, few modifications	Many modifications, single-thread apps Many modifications, multi-thread apps	

Use String → When text won't change often.

Use StringBuilder → When text changes frequently and thread safety isn't a concern.

Use StringBuffer → When text changes frequently and thread safety is required.

JDBC

Steps to Establish JDBC Connection and Fetch Results

Load JDBC Driver (not mandatory for modern JDBC 4.0+ but good to show).

Establish a connection to the database using `DriverManager.getConnection(url, username, password)`.

Create a Statement or PreparedStatement.

Execute a SQL query using `executeQuery()`.

Process ResultSet row by row.

Close resources (ResultSet, Statement, Connection) in finally or use try-with-resources.

```

import java.sql.*;

public class JdbcExample {
    public static void main(String[] args) {
        // DB connection details

        String url = "jdbc:mysql://localhost:3306/testdb"; // database URL
        String user = "root"; // username
        String password = "password"; // password

        // SQL query
        String query = "SELECT id, name, email FROM users";

        // Try-with-resources automatically closes Connection, Statement, ResultSet
        try (Connection conn = DriverManager.getConnection(url, user, password);
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(query)) {

            System.out.println("Connection established successfully!\n");
            System.out.println("User Records:");
            System.out.println("-----");

            // Process result set
            while (rs.next()) {
                int id = rs.getInt("id");    // fetch column by name
                String name = rs.getString("name");
                String email = rs.getString("email");

                System.out.println(id + " | " + name + " | " + email);
            }
        } catch (SQLException e) {

```

```
e.printStackTrace();  
}  
  
}  
  
}
```

JDBC URL format for MySQL:

`jdbc:mysql://<host>:<port>/<databaseName>`

e.g., `jdbc:mysql://localhost:3306/testdb`

Driver dependency:

You need the MySQL Connector/J (`mysql-connector-java-x.x.x.jar`) in your classpath (or Maven/Gradle dependency).

Exception handling: Always handle `SQLException`.

Best practice: Use `PreparedStatement` instead of `Statement` to prevent SQL injection.

For other DBs (Oracle, PostgreSQL, etc.), just change the URL and driver JAR.

Java's access modifiers — explained (with examples)

Short version: Java has four accessibility levels for classes, fields, methods and constructors:

`public` — visible everywhere.

`private` — visible only inside the same class.

`protected` — **visible inside the same package, and also visible to subclasses (including subclasses in other packages)** with an important nuance (explained below).

default (no modifier) a.k.a. package-private — visible only to code in the same package.

`private` < default < `protected` < `public`

Below I'll explain each in detail and then give a concrete multi-package example that demonstrates what compiles and what doesn't.

What each modifier means

`public`

Can be accessed from any other class, anywhere (any package).

Use for public APIs.

`private`

Only accessible inside the same class where the member is declared.

Not visible to subclasses, same-package classes, or any other class.

Use to enforce encapsulation (hide implementation details).

protected

Accessible to any class in the same package (like package-private).

Also accessible to subclasses even if the subclass is in a different package.

Important nuance: a subclass in a different package can access the protected member as inherited (i.e., via this or directly) but cannot access a protected member of another instance of the superclass (or of a plain superclass reference) if that instance is not of the subclass type. See example below.

default / package-private (no keyword)

If you omit any modifier, the member is package-private.

Accessible to any class in the same package, but not to subclasses in another package and not to classes in other packages.

Top-level classes: Only public or package-private (no modifier) are allowed. You cannot declare a top-level class as private or protected. Nested (inner) classes can be private/protected.

Quick accessibility table

Modifier	Same class		Same package (other class)		Subclass (same package)		Subclass (different package)	
	Yes	No	Yes	No	Yes	No	Yes	No
public	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	Yes	Yes (only in subclass code / via inheritance)	No	No	No
default (package-private)	Yes	No	Yes	Yes	Yes	No	No	No
private	Yes	No	No	No	No	No	No	No

Explain the internal working of ConcurrentHashMap in Java 8.

- ☐ Difference between synchronized, ReentrantLock and ReadWriteLock.
- ☐ How does Java handle memory leaks and how would you detect them?
- ☐ How would you design a thread-safe singleton in Java?
- ☐ Explain Java memory model (heap, stack, metaspace) and garbage collection tuning.

◆ Spring & Spring Boot

- ☐ Explain how dependency injection works internally in Spring.
- ☐ Difference between Spring Boot starters vs manual dependency management.
- ☐ How do you implement idempotency in REST APIs?

☐ How do you handle huge traffic in Spring Boot APIs? (connection pool, caching, async calls)

🔊 How would you secure AWS-hosted Spring Boot APIs with IAM roles / Cognito / JWT?

◆ JPA / Hibernate

☐ How would you optimize large batch inserts/updates in Hibernate?

☐ Explain difference between entity graphs and fetch joins for optimizing queries.

☐ How does 2nd level caching work in Hibernate and which providers have you used?

☐ How to handle optimistic locking failures gracefully?

☐ How do you tune SQL queries generated by JPA for performance?

◆ Microservices & Architecture

☐ Explain the difference between orchestration and choreography in microservices.

☐ How would you handle distributed transactions at scale? (Saga / Outbox pattern)

☐ Explain API throttling, rate limiting, and request bursting.

☐ How do you design a high-availability, fault-tolerant microservice?

☐ Describe an event-driven architecture you have implemented (Kafka, SQS, SNS, etc.).

◆ Cloud & DevOps Basics (AWS Focus)

☐ Explain difference between AWS EC2, ECS, EKS, and Lambda.

☐ How would you secure secrets and credentials in AWS? (Parameter Store, Secrets Manager)

☐ How do you achieve blue/green or canary deployments on AWS?

- ☐ How do you implement logging, tracing and monitoring (CloudWatch, X-Ray, ELK)?
- ☐ Difference between scaling using Auto Scaling Groups vs Kubernetes HPA.

◆ DSA & System Design (Amazon Special Focus)

- ☐ Design a rate limiter (like Amazon API Gateway throttling).
- ☐ Design a cache system with expiration and eviction policy.
- ☐ Given a massive log file stored in S3, find top 10 frequently accessed URLs efficiently.
- ☐ Given two sorted streams of data, merge them in real-time with minimal memory.
- ☐ Implement a thread-safe LRU cache in Java.

◆ Behavioral / Leadership Principles (Always Asked at Amazon)

- ☐ Tell me about a time you disagreed with your manager.
- ☐ Describe a situation where you delivered results under tight deadlines.
- ☐ Tell me about a time when you improved a process or system.
- ☐ Describe how you handle failure in a project.
- ☐ Tell me about a time when you influenced a team without authority

Collection Framework in Java — explained like I'm teaching a kid 🧒

Think of the Collection Framework as the toy-storage system in your room. Java gives you different kinds of toy boxes and shelves so you can store, find, and play with toys (data) easily. Each box has rules: some keep order, some don't allow duplicates, some let you grab from both ends, etc. I'll explain each box (interface/implementation) one by one and show a simple code example.

Big picture (short)

Collection is the parent interface for many things that hold groups of items (like lists, sets, queues).

List — an ordered box where position/index matters (can have duplicates).

Set — a box that never keeps two identical toys (no duplicates).

Queue / Deque — boxes where you add and remove in special ways (like line or both ends).

Map — a map is not a Collection, but it's part of the framework: it stores key → value pairs (like name-tag → toy).

1) Collection (basic rules)

This is the root interface. Common methods:

add(item), remove(item), size(), contains(item), iterator().

Think: Collection = any toy container that follows basic rules (add, remove, count).

2) List — ordered, indexable (ArrayList, LinkedList)

What it is: A row of toys where each toy has a position (index). You can have same toy many times.

Analogy: A bookshelf — books in order. You can ask “what book is at position 2?”

Common implementations:

ArrayList — like an array-backed shelf. Fast to get by index, slower to insert in middle.

LinkedList — like a chain of boxes. Fast to add/remove at ends or when you have position node, slower to get by index.

Example (ArrayList):

```
import java.util.*;

public class ListExample {

    public static void main(String[] args) {

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

        fruits.add("Apple");

        fruits.add("Banana");

        fruits.add("Apple"); // duplicate allowed

        System.out.println(fruits); // [Apple, Banana, Apple]

        // access by index

        System.out.println(fruits.get(1)); // Banana

        // remove by index

        fruits.remove(0); // removes first Apple

        System.out.println(fruits); // [Banana, Apple]

    }

}
```

3) Set — no duplicates (HashSet, LinkedHashSet, TreeSet)

What it is: A toy box that won't accept two exactly same toys.

Analogy: Names on name-tags — you can't have two identical tags.

Common implementations:

HashSet — fast, no guaranteed order.

LinkedHashSet — keeps insertion order.

TreeSet — keeps items sorted (like alphabetical order).

Example (HashSet):

```
import java.util.*;

public class SetExample {

    public static void main(String[] args) {

        Set<String> names = new HashSet<>();

        names.add("Nisha");

        names.add("Ava");

        names.add("Nisha"); // duplicate ignored

        System.out.println(names); // [Ava, Nisha] (order may vary)

    }

}
```

4) Queue and Deque — lines and double-ended lines

What it is: Places where we add items to one side and remove from another — like queues at a slide.

Analogy:

Queue = kids in a line (first in, first out = FIFO).

Deque = line where you can add/remove from both front and back.

Implementations: LinkedList implements Queue; ArrayDeque is a fast double-ended queue; PriorityQueue removes smallest/priority item first.

Example (Queue with LinkedList):

```
import java.util.*;

public class QueueExample {

    public static void main(String[] args) {

        Queue<String> line = new LinkedList<>();

        line.offer("Anna"); // join line

        line.offer("Ben");
```

```

        System.out.println(line.poll()); // Anna (first)

        System.out.println(line); // [Ben]
    }
}

```

Example (PriorityQueue - small becomes first):

```

PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.add(30);
pq.add(10);
pq.add(20);

System.out.println(pq.poll()); // 10 (smallest number)

```

5) Map — key → value pairs (HashMap, LinkedHashMap, TreeMap)

What it is: A dictionary where each toy has a label (key) and a value. Keys must be unique.

Analogy: Label your boxes with names: ("red box" → "cars").

Common implementations:

HashMap — fast, no order.

LinkedHashMap — remembers insertion order.

TreeMap — keys sorted.

Example (HashMap):

```

import java.util.*;

public class MapExample {

    public static void main(String[] args) {

        Map<String, Integer> ages = new HashMap<>();
        ages.put("Nisha", 26);
        ages.put("Ava", 10);

        System.out.println(ages.get("Nisha")); // 26

        // iterate entries
        for (Map.Entry<String,Integer> e : ages.entrySet()) {

            System.out.println(e.getKey() + " -> " + e.getValue());

        }

    }

}

```

6) Iterating safely: Iterator and for-each

Use for (Type x : collection) when you only read. Use Iterator if you want to remove while looping.

Example (removing with iterator):

```
List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C"));
Iterator<String> it = list.iterator();
while (it.hasNext()) {
    String s = it.next();
    if (s.equals("B")) it.remove(); // safe removal
}
System.out.println(list); // [A, C]
```

7) Sorting: Comparable vs Comparator

Comparable — the class itself says how to compare (natural order).

Comparator — external object that defines a custom order.

Example (Comparator):

```
List<String> fruits = Arrays.asList("Apple", "Banana", "Kiwi");
fruits.sort(Comparator.reverseOrder()); // sort descending
System.out.println(fruits); // [Kiwi, Banana, Apple]
```

8) Performance (very simple)

ArrayList — fast .get(index), slower insert/remove in middle.

LinkedList — slower .get(index), faster insert/remove at ends.

HashSet / HashMap — usually very fast for add/get (average O(1)).

TreeSet / TreeMap — keeps things sorted, slower (O(log n)).

Think: choose the right box depending on what you do most (read a lot? write a lot? need sorted? no duplicates?).

9) Extras (short)

Collections class has helpers like Collections.sort(list), Collections.shuffle(list).

Thread-safe versions: Collections.synchronizedList(...) or ConcurrentHashMap (for many kids sharing the box at once).

Quick compact example that shows many things

```
import java.util.*;

public class QuickDemo {
    public static void main(String[] args) {
```

```

// List
List<String> list = new ArrayList<>(Arrays.asList("A","B","C","B"));
System.out.println("List: " + list);

// Set
Set<String> set = new HashSet<>(list);
System.out.println("Set: " + set);

// Queue
Queue<String> q = new LinkedList<>();
q.offer("First"); q.offer("Second");
System.out.println("Queue poll: " + q.poll());

// Map
Map<String,Integer> map = new HashMap<>();
map.put("A", 1); map.put("B", 2);
map.forEach((k,v) -> System.out.println(k + " -> " + v));
}
}

```

Summary (kid-friendly)

List = order + duplicates (books on shelf).

Set = no duplicates (name-tags, unique).

Queue/Deque = line or double-ended line (playground slide).

Map = keys and values (label → toy).

Time Complexity

Time Complexity tells us how the runtime of an algorithm grows with respect to the size of the input (n).

It does not measure actual seconds, but the rate of growth (approximate).

顯 Represented using Big-O Notation:

$O(1)$ → Constant time

$O(\log n)$ → Logarithmic time

$O(n)$ → Linear time

$O(n \log n)$ → Linearithmic time

$O(n^2)$ → Quadratic time

$O(2^n)$, $O(n!)$ → Exponential/Fast-growing

✓ Example 1: $O(1)$ → Constant Time

```
class Example {  
    public static void main(String[] args) {  
        int[] arr = {10, 20, 30, 40};  
        System.out.println(arr[0]); // Accessing first element  
    }  
}
```

頻 Always takes constant time, regardless of array size.

Time Complexity: $O(1)$

✓ Example 2: $O(n)$ → Linear Time

```
class Example {  
    public static void main(String[] args) {  
        int[] arr = {10, 20, 30, 40};  
        for (int i = 0; i < arr.length; i++) {  
            System.out.println(arr[i]);  
        }  
    }  
}
```

頻 Loop runs once for each element → if array has n elements, operations = n .

Time Complexity: $O(n)$

✓ Example 3: $O(n^2)$ → Quadratic Time

```
class Example {  
    public static void main(String[] args) {  
        int[] arr = {1, 2, 3, 4};  
        for (int i = 0; i < arr.length; i++) {  
            for (int j = 0; j < arr.length; j++) {  
                System.out.println(arr[i] + " " + arr[j]);  
            }  
        }  
    }  
}
```

```
}  
}
```

顯 Nested loop $\rightarrow n * n$ operations.

Time Complexity: $O(n^2)$

✓ Example 4: $O(\log n) \rightarrow$ Logarithmic Time

```
class Example {  
    public static void main(String[] args) {  
        int n = 16;  
        while (n > 1) {  
            n = n / 2; // keep dividing  
            System.out.println(n);  
        }  
    }  
}
```

顯 Each step reduces input size by half $\rightarrow \log(n)$ steps.

Time Complexity: $O(\log n)$

Space Complexity

Space Complexity measures the amount of memory an algorithm uses relative to input size (n).

It includes:

Memory for input variables

Memory for auxiliary data structures (arrays, hash maps, stacks, etc.)

Memory used by recursion (function call stack)

✓ Example 1: $O(1) \rightarrow$ Constant Space

```
class Example {  
    public static void main(String[] args) {  
        int a = 10, b = 20; // Only 2 variables  
        int sum = a + b;  
        System.out.println(sum);  
    }  
}
```

```
}  
}
```

顯 No matter how big input is, memory usage is constant.

Space Complexity: $O(1)$

✓ Example 2: $O(n) \rightarrow$ Linear Space

```
class Example {  
    public static void main(String[] args) {  
        int n = 5;  
        int[] arr = new int[n]; // Array size grows with n  
        for (int i = 0; i < n; i++) {  
            arr[i] = i;  
        }  
    }  
}
```

顯 Array of size n takes linear space.

Space Complexity: $O(n)$

✓ Example 3: Recursion \rightarrow Stack Space

```
class Factorial {  
    static int fact(int n) {  
        if (n == 0) return 1;  
        return n * fact(n - 1);  
    }  
    public static void main(String[] args) {  
        System.out.println(fact(5));  
    }  
}
```

顯 Each recursive call stores data on the function call stack.

If $n = 5$, stack holds 5 calls.

Space Complexity: $O(n)$ (because recursion depth = n).

□ 3. Key Points for Interviews

Time Complexity → "How fast is your algorithm?"

Space Complexity → "How much memory does it need?"

Always analyze both when giving answers.

Use Big-O for worst-case analysis (most common in interviews).

✓ Example Interview Q:

顯 Q: What is the time and space complexity of Binary Search?

Time Complexity: $O(\log n)$ (since we divide array in half each step)

Space Complexity:

Iterative → $O(1)$ (constant space)

Recursive → $O(\log n)$ (because of recursion stack)

How do you calculate the difference between two dates in Java ↔?

37. What is the purpose of the java.time package in Java ↔?
38. What are the advantages of the new Date/Time API over the old java.util.Date class?
39. How does Duration work in Java ↔for date-time calculations?
40. What is the purpose of CompletableFuture in Java ↔?
41. What is the difference between Future and CompletableFuture in Java ↔?
42. How do you handle asynchronous programming in Java ↔?
43. What are the new java.util.function interfaces introduced in Java ↔?
44. What is the purpose of BiFunction, UnaryOperator, BinaryOperator in Java ↔?
45. Explain how Stream can be used for filtering and mapping objects.
46. What is the difference between intermediate and terminal operations in Java ↔Streams?
47. Can you explain how the map() operation works in Java ↔Streams?
48. How do you sort elements in a stream in Java ↔?
49. What is the purpose of Collectors.toList(), Collectors.toMap(), and Collectors.joining()?
50. What is the Optional.orElse() method, and how does it work?
51. How do you handle exceptions in streams in Java ↔?
52. What does the distinct() method do in Java ↔Streams?
53. How can you find the maximum or minimum value in a stream in Java ↔?
54. How do you create a stream from a file or a collection in Java ↔?

55. What does the peek() method do in Java ←Streams?
56. What is the MethodHandles class in Java ←, and how is it used?
57. What is the Stream's skip() and limit() methods used for in Java ←?
58. What are the new default methods introduced in Java ←Interfaces?
59. How do you combine streams using flatMap() in Java ←?
60. How does Java ←handle parallel processing with parallelStream()?
61. Explain the difference between findFirst() and findAny() in Java ←Streams.
62. What is the purpose of Collectors.groupingBy() in Java ←?
63. How does the Collectors.partitioningBy() work in Java ←Streams?
64. What is the use of Stream.concat() in Java ←?
65. Can you explain the purpose of Spliterator in Java ←?

1. Key Difference

Feature == Operator .equals() Method

Type Operator Method (defined in Object class)

Compares References / Memory addresses Content / Values (unless not overridden)

Default Behavior Checks if two references point to the same object in memory In Object class, it behaves like ==. But many classes (like String, Integer) override it to compare contents.

For Primitives Compares values Cannot be used directly (not a method on primitives)

For Objects Compares memory addresses Compares content / state of objects if overridden

✓ 2. Examples

□ (A) With Primitive Types

```
int a = 10;
```

```
int b = 10;
```

```
System.out.println(a == b);    // true (compares value)
```

✓ For primitives, == compares actual values (no .equals() method exists for primitives).

□ (B) With String Literals

```
String s1 = →Java→;
```

```
String s2 = →Java→;
```

```
System.out.println(s1 == s2);    // true (both refer to same string in String pool)
```

```
System.out.println(s1.equals(s2)); // true (contents are same)
```

✓ Both return true because string literals with the same value are stored in the string pool, so they share the same reference.

□ (C) With New String Objects

```
String s1 = new String(→Java→);
```

```
String s2 = new String(→Java→);
```

```
System.out.println(s1 == s2);    // false (different objects in memory)
```

```
System.out.println(s1.equals(s2)); // true (contents are same)
```

✓ Here:

`==` → false because they are two different objects.

`.equals()` → true because String class overrides `.equals()` to compare contents.

□ (D) With Custom Objects

```
class Person {
```

```
    String name;
```

```
Person(String name) {  
    this.name = name;  
}  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Person p1 = new Person("Alice");  
        Person p2 = new Person("Alice");  
  
        System.out.println(p1 == p2);    // false (different objects in memory)  
        System.out.println(p1.equals(p2)); // false (default equals() compares references)  
    }  
}
```

✓ Here both == and .equals() return false because:

They refer to two different objects in memory.

We did not override .equals(), so it behaves like ==.

□ (E) Overriding .equals()

```
class Person {  
    String name;  
  
    Person(String name) {
```

```
    this.name = name;
}
```

@Override

```
public boolean equals(Object obj) {
    if (this == obj) return true;        // same reference
    if (obj == null || getClass() != obj.getClass()) return false;
    Person p = (Person) obj;
    return this.name.equals(p.name);     // compare content
}
}
```

```
public class Main {
    public static void main(String[] args) {
        Person p1 = new Person(→Alice→);
        Person p2 = new Person(→Alice→);

        System.out.println(p1 == p2);    // false (different objects)
        System.out.println(p1.equals(p2)); // true (contents match)
    }
}
```

✓ After overriding equals(), it now checks the content of the Person object.

✂ 3. Summary Table

Case == .equals()

Primitive types	Compares values	Not applicable
-----------------	-----------------	----------------

String literals Compares reference (often true because of String pool) Compares content (true)

New String objects Compares reference (false) Compares content (true)

Custom objects (no override) Compares reference (false) Compares reference (false)

Custom objects (override) Compares reference Compares content (depends on override)

□ Interview Tip

Always use `.equals()` for comparing contents of objects (like String, Integer, custom classes).

Use `==` for primitive types or to check if two references point to the exact same object.

If you write custom classes and want meaningful equality, override `equals()` (and `hashCode()`).

✓ Quick Example Recap

```
String a = →Hello→;
```

```
String b = new String(→Hello→);
```

```
System.out.println(a == b);   // false (different memory)
```

```
System.out.println(a.equals(b)); // true (same content)
```

里 1. Definition

Abstraction

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

It focuses on what an object does, not how it does it.

顯 Implemented using abstract classes or interfaces in Java.

Example idea:

When you drive a car, you only know how to operate the steering, brakes, and accelerator — you don't know the internal working of the engine.

That's abstraction — showing only necessary details.

Encapsulation

➡ Encapsulation is the process of wrapping data (variables) and methods (functions) together into a single unit (class).

It also helps in data hiding — preventing direct access to the fields from outside.

顯 Implemented using private fields with public getter/setter methods.

Example idea:

A bank account class hides its balance variable and allows you to access it only through deposit() or withdraw() methods.

That's encapsulation — protecting and controlling access to data.

責 2. Real-life Analogy

Concept	Analogy	Explanation
Abstraction	A TV remote	You press buttons to change channels — you don't know how signals are sent internally.
Encapsulation	A medicine capsule	The medicine (data) is enclosed inside a capsule (class), preventing direct access.

穽 Polymorphism in Java

簽 Definition:

Polymorphism means “many forms”.

In Java, it allows a single action (method or behavior) to behave differently based on the object that is calling it.

易 Simple Example:

```
class Animal {
    void sound() {
        System.out.println("Animal makes a 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 Main {
    public static void main(String[] args) {
```

```

        Animal a1 = new Dog();    // Animal reference, Dog object
        Animal a2 = new Cat();    // Animal reference, Cat object

        a1.sound();    // Output → Dog barks
        a2.sound();    // Output → Cat meows
    }
}

```

🗨️ Explanation:

- The method `sound()` behaves differently based on the object (Dog or Cat) — even though both are called from the same reference type (Animal).
- This is called Polymorphism.

Types of Polymorphism in Java

Type	Also Called	When It Occurs
1 Compile-time Polymorphism	Static Binding / Early Binding	During compilation
2 Runtime Polymorphism	Dynamic Binding / Late Binding	During program execution

1 Compile-Time Polymorphism (Method Overloading)

📌 Definition:

When multiple methods in the same class have the same name but different parameters (type, number, or order), it is called method overloading.

It is resolved at compile time by the Java compiler.

✓ Example: Method Overloading

```

class Calculator {
    // Method 1
    int add(int a, int b) {
        return a + b;
    }

    // Method 2 - different number of parameters
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method 3 - different data types
    double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator c = new Calculator();
        System.out.println(c.add(5, 10));    // Calls method 1
        System.out.println(c.add(5, 10, 15)); // Calls method 2
    }
}

```

```

        System.out.println(c.add(2.5, 3.5));    // Calls method 3
    }
}

```

里 Output:

```

15
30
6.0

```

簽 Key Points:

- Same method name, but different parameter list.
- **Return type alone cannot distinguish overloaded methods.**
- **Resolved at compile-time (static binding).**

2 Runtime Polymorphism (Method Overriding)

責 Definition:

When a **subclass provides a specific implementation** of a method that is already defined in its parent class, it's called **method overriding**.

It is resolved **at runtime**, not compile time.

✓ Example: Method Overriding

```

class Bank {
    float getRateOfInterest() {
        return 0;
    }
}

class SBI extends Bank {
    float getRateOfInterest() {
        return 7.5f;
    }
}

class HDFC extends Bank {
    float getRateOfInterest() {
        return 8.0f;
    }
}

public class Main {
    public static void main(String[] args) {
        Bank b;

        b = new SBI();
        System.out.println("SBI ROI: " + b.getRateOfInterest());

        b = new HDFC();
        System.out.println("HDFC ROI: " + b.getRateOfInterest());
    }
}

```

```
}  
}
```

里 Output:

SBI ROI: 7.5
HDFC ROI: 8.0

簽 Key Points:

- The method in the child class overrides the method in the parent class.
- Method call depends on the object type, not the reference type.
- Achieved using inheritance and upcasting (Bank b = new HDFC());
- Resolved at runtime (dynamic binding).

易 Difference Between Overloading and Overriding

Feature	Method Overloading	Method Overriding
Binding	Compile-time	Runtime
Method Name	Same	Same
Parameters	Must be different	Must be same
Return Type	Can be same or different	Must be same (or covariant)
Access Modifier	Can be different	Cannot reduce visibility
Inheritance	Not required	Required (between parent & child)
Static Methods	Can be overloaded	Cannot be overridden (they are hidden)

⚙ Example Showing Both Together

```
class Shape {  
    void draw() {  
        System.out.println("Drawing a shape");  
    }  
}  
  
class Circle extends Shape {  
    void draw() {  
        System.out.println("Drawing a circle");  
    }  
  
    // Overloaded version  
    void draw(String color) {  
        System.out.println("Drawing a " + color + " circle");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Shape s = new Circle(); // Runtime polymorphism  
        s.draw();                // Output: Drawing a circle  
  
        Circle c = new Circle();
```

```
        c.draw("red");           // Compile-time polymorphism
    }
}
```

拓 Summary

Type	Description	Example	Resolved
Compile-time Polymorphism	Method Overloading	Same method name, different parameters	At Compile time
Runtime Polymorphism	Method Overriding	Parent and Child have same method	At Runtime

李 Real-World Example of Polymorphism

Suppose we have a Payment system:

```
class Payment {
    void pay() {
        System.out.println("Making a generic payment");
    }
}

class CreditCardPayment extends Payment {
    void pay() {
        System.out.println("Paying using Credit Card");
    }
}

class UpiPayment extends Payment {
    void pay() {
        System.out.println("Paying using UPI");
    }
}

public class Main {
    public static void main(String[] args) {
        Payment p1 = new CreditCardPayment();
        Payment p2 = new UpiPayment();

        p1.pay(); // Output: Paying using Credit Card
        p2.pay(); // Output: Paying using UPI
    }
}
```

顯 This is runtime polymorphism — the same method call (pay()) behaves differently depending on the actual object type.

Would you like me to include a diagram showing how compile-time and runtime polymorphism differ internally (memory + binding flow)?

It's very helpful for interview visualization.

You said:

what is dynamic polymorphism?

ChatGPT said:

Excellent question 趂 — this is one of the most important concepts in Object-Oriented Programming (OOP) and Java interviews, often asked like:

“What is dynamic polymorphism in Java?”

or

“How does runtime method dispatch work?”

Let’s understand it clearly and deeply, with examples

穉 Dynamic Polymorphism in Java

簽 Definition:

Dynamic Polymorphism (also called Runtime Polymorphism) in Java occurs when a method call is resolved at runtime (not at compile time). It happens when a parent class reference variable refers to a child class object, and the overridden method of the child class is executed.

責 Simple Meaning:

Same method → behaves differently depending on which object it’s called on (at runtime).

✓ Example:

```
class Animal {
    void sound() {
        System.out.println("Animal makes a 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 Main {
    public static void main(String[] args) {
        Animal a;           // reference variable of parent class

        a = new Dog();       // Dog object
        a.sound();           // Output: Dog barks
    }
}
```

```
        a = new Cat(); // Cat object
        a.sound();     // Output: Cat meows
    }
}
```

% Explanation:

- The parent class Animal has a method sound().
- The child classes Dog and Cat override the method.
- The reference variable a is of type Animal, but the object changes (Dog, then Cat).
- At runtime, Java decides which version of sound() to call based on the actual object type, not the reference type.

This behavior is Dynamic (Runtime) Polymorphism. ✓

⚙ How It Works Internally (Runtime Method Dispatch):

1. At compile-time:
The compiler only checks that sound() exists in the Animal class.
 2. At runtime:
The JVM checks the actual object the reference points to (like Dog or Cat)
→ Then executes that class's overridden version of the method.
-

⚡ Key Rules for Dynamic Polymorphism

Rule	Description
1	Requires Inheritance (a subclass extending a superclass).
2	Requires Method Overriding (same method name, same parameters, same return type).
3	Requires Upcasting (parent reference → child object).
4	Decided at runtime, not at compile-time.

🏠 Real-World Example

```
class Payment {
    void pay() {
        System.out.println("Processing generic payment");
    }
}

class CreditCardPayment extends Payment {
    void pay() {
        System.out.println("Processing credit card payment");
    }
}

class UpiPayment extends Payment {
    void pay() {
```



```

        System.out.println("Processing UPI payment");
    }
}

public class Main {
    public static void main(String[] args) {
        Payment p;

        p = new CreditCardPayment();
        p.pay(); // Output: Processing credit card payment

        p = new UpiPayment();
        p.pay(); // Output: Processing UPI payment
    }
}

```

➔ The same method `pay()` behaves differently depending on the actual object type. This is Dynamic Polymorphism in action.

Dynamic Polymorphism vs Static Polymorphism

Feature	Static Polymorphism	Dynamic Polymorphism
Also Called	Compile-time Polymorphism	Runtime Polymorphism
Achieved By	Method Overloading	Method Overriding
Binding Time	Compile-time	Runtime
Decided By	Compiler	JVM
Example	<code>add(int, int)</code> and <code>add(double, double)</code>	<code>Animal a = new Dog(); a.sound();</code>

臭 Note:

- **Static methods, constructors, and private methods cannot be overridden, so they do not participate in dynamic polymorphism.**
 - **Only instance methods (non-static) can be dynamically bound at runtime.**
-

✔ Summary

Point	Description
Concept	A method call is resolved at runtime, based on the actual object type
Achieved By	Method overriding + inheritance + upcasting
Decided By	JVM (dynamic binding)
Benefit	Enables flexibility and code reusability (you can add new subclasses without changing parent class code)

⚙️ 1. “During Compilation” — Compile Time

責 Meaning:

When we say “during compilation”, we’re talking about the phase when your Java source code (.java file) is converted into bytecode (.class file) by the Java compiler (javac).

This happens before the program is run.

It’s when the compiler checks your code for syntax and type errors — anything that violates Java’s rules.

✓ Example:

```
class Example {
    public static void main(String[] args) {
        int a = 10;
        String b = "Hello";

        // Compile-time error: type mismatch
        a = b; // ✗ Error: cannot assign a String to an int
    }
}
```

里 What happens here:

- The compiler (javac) reads your code line by line.
- It finds that you’re trying to assign a String to an int.
- It stops and throws a compile-time error (so the program never runs).

易 In short:

"During compilation" means when your code is being checked and converted into bytecode, before running it.

2. “During Program Execution” — Runtime

責 Meaning:

When we say “during program execution”, we mean the phase when your compiled .class file is actually running inside the Java Virtual Machine (JVM) — this is runtime.

The JVM loads the bytecode and executes it line by line.

At this stage, runtime errors (or exceptions) can occur — things that the compiler cannot catch.

✓ Example:

```
class Example {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3};

        // Runtime error: ArrayIndexOutOfBoundsException
        System.out.println(arr[5]); // ✗ Error at runtime
    }
}
```

```
}
```

里 What happens here:

- The compiler sees nothing wrong — syntactically correct.
- The code compiles successfully into bytecode.
- When the program runs, the JVM tries to access `arr[5]`, which doesn't exist.
- JVM throws a runtime exception (`ArrayIndexOutOfBoundsException`).

易 In short:

"During execution" means while the program is running on the JVM — errors here are called runtime errors or exceptions.

⚖ Summary — Compile Time vs Runtime in Java

Aspect	Compile Time	Runtime
Who checks it	Compiler (javac)	JVM
When it happens	Before execution	While the program runs
Type of errors	Syntax errors, type errors	Exceptions, logical errors
Examples	Missing semicolon, type mismatch, undeclared variable	<code>NullPointerException</code> , <code>ArrayIndexOutOfBoundsException</code> , <code>ArithmeticException</code>
Fix	Must fix before program runs	Program may crash at runtime if not handled with try-catch

拓 Real-world analogy:

Think of writing and running a program like building and flying an airplane ✈

- **Compilation time = inspecting and assembling the airplane before takeoff (checking structure, missing parts — can't fly if broken).**
- **Runtime = flying the airplane — now unexpected issues (like engine failure) can happen even if it passed inspection.**

What is static in Java?

In Java, the **static** keyword is used for **memory management** and to define **class-level members**.

It means that the member (variable, method, or block) **belongs to the class** rather than to any specific instance (object) of that class.

So, you can access static members **without creating an object** of the class.

⚙ Simple Definition

The **static** keyword in Java is used to share the same variable or method of a class among all objects of that class.

里 Where can we use **static** in Java?

static can be used with:

1. Static Variables (Class Variables)
2. Static Methods
3. Static Blocks
4. Static Nested Classes
5. Static Import

Let's understand them one by one

燐 1. Static Variable (Class Variable)

A **static variable** is shared by **all instances** of the class.
It is created only **once** when the class is loaded into memory.

簽 **Example:**

```
class Employee {
    int empId;
    String empName;
    static String company = "LTI Mindtree"; // static variable

    Employee(int id, String name) {
        empId = id;
        empName = name;
    }

    void display() {
        System.out.println(empId + " " + empName + " " + company);
    }

    public static void main(String[] args) {
        Employee e1 = new Employee(101, "Nisha");
        Employee e2 = new Employee(102, "Rahul");
        e1.display();
        e2.display();
    }
}
```

貼 Output:

```
101 Nisha LTI Mindtree
102 Rahul LTI Mindtree
```

✓ Here, company is shared by both objects.

If one changes it, all see the change.

✂ 2. Static Method

A static method belongs to the class, not objects.

- It can be called without creating an object.
- It can access only static data (not instance variables or instance methods).

簽 Example:

```
class MathUtil {
    static int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        int result = MathUtil.add(10, 20); // no object needed
        System.out.println("Sum: " + result);
    }
}
```

貼 Output:

Sum: 30

⚠ Static methods cannot use this or super keywords, because they are not tied to any specific object.

隣 3. Static Block

A static block is used to initialize static variables or to execute some code only once when the class is loaded.

簽 Example:

```
class Demo {
    static int value;

    static {
        System.out.println("Static block executed");
        value = 100;
    }

    public static void main(String[] args) {
        System.out.println("Main method executed");
        System.out.println("Value: " + value);
    }
}
```

貼 Output:

Static block executed

Main method executed
Value: 100

✓ Static blocks run before the main() method and only once.

里 4. Static Nested Class

You can define a class inside another class using the static keyword.

簽 Example:

```
class Outer {
    static class Inner {
        void show() {
            System.out.println("Static Nested Class Method");
        }
    }

    public static void main(String[] args) {
        Outer.Inner obj = new Outer.Inner();
        obj.show();
    }
}
```

貼 Output:

Static Nested Class Method

✓ A static nested class does not need an instance of the outer class to be instantiated.

輒 5. Static Import

You can import static members (methods/fields) of another class directly.

簽 Example:

```
import static java.lang.Math.*; // static import

class Test {
    public static void main(String[] args) {
        System.out.println(sqrt(25)); // no need for Math.sqrt()
        System.out.println(pow(2, 3)); // no need for Math.pow()
    }
}
```

貼 Output:

5.0
8.0

易 Why is static used in Java?

✓ Reasons:

1. **Memory efficiency:** Shared memory for static members — no need for duplicate copies per object.
2. **Utility methods:** Used for helper methods like `Math.sqrt()`, `Math.max()`, etc.
3. **Constants:** Common for all objects (e.g., `static final` constants).
4. **Initialization:** Static blocks initialize configuration or static data at class load time.

責 Advantages of static:

Advantage	Description
Memory Efficiency	Saves memory because only one copy of a static member exists.
No Object Needed	You can access methods/variables directly using the class name.
Useful for Utility/Helper Classes	Like <code>Math</code> , <code>Collections</code> , or <code>Arrays</code> in Java.
Shared Data	Easy to maintain values common for all objects.

⚠ Disadvantages of static:

Disadvantage	Description
Tight Coupling	Static members make code less flexible and harder to modify.
No Polymorphism	Static methods cannot be overridden (only hidden).
Thread Safety	Shared static variables can cause issues in multi-threaded programs.
Less Object-Oriented	Overuse of static breaks OOP principles like encapsulation.

竄 Summary Table

Usage	Belongs To	Accessible Using	Access Instance Members?
Static Variable	Class	Class name or object	✓ (Yes, but not recommended)
Static Method	Class	Class name	✗ (Only static data)
Static Block	Class	Runs Automatically	✗
Static Nested Class	Class	Class name	✗ (Needs outer class name)

詹 Example Combining All Static Uses

```
class Utility {
    static int count;
    static final String APP_NAME;

    static {
        APP_NAME = "MyStaticApp";
        System.out.println("Static Block Initialized");
    }
}
```

```

    }

    static void increment() {
        count++;
    }

    void show() {
        System.out.println("Count: " + count);
    }

    static class Inner {
        void message() {
            System.out.println("Inside static nested class of " + APP_NAME);
        }
    }

    public static void main(String[] args) {
        Utility.increment();
        Utility.increment();

        Utility u = new Utility();
        u.show();

        Utility.Inner inner = new Utility.Inner();
        inner.message();
    }
}

```

貼 Output:

```

Static Block Initialized
Count: 2
Inside static nested class of MyStaticApp

```

✓ In Short

Term	Meaning
static variable	Shared across all objects
static method	Belongs to class, not object
static block	Executes once when class loads
static nested class	Independent inner class
Used for	Utility methods, constants, shared data
Avoid when	You need polymorphism, encapsulation, or thread safety

1. Comparable Interface

► Definition

Comparable is an interface present in java.lang package.

It is used to define the natural ordering of objects (i.e., how objects should be compared with each other by default).

► Syntax

```
public interface Comparable<T> {  
    int compareTo(T obj);  
}
```

► How it Works

- A class implements Comparable when it wants to define its own sorting logic.
 - The compareTo() method returns:
 - 0 → if both objects are equal
 - a **positive number** → if the current object is greater
 - a **negative number** → if the current object is smaller
-

► Example

```
class Employee implements Comparable<Employee> {  
    int id;  
    String name;  
    int salary;  
  
    Employee(int id, String name, int salary) {  
        this.id = id;  
        this.name = name;  
        this.salary = salary;  
    }  
  
    // Natural ordering: by salary  
    public int compareTo(Employee e) {  
        return this.salary - e.salary; // ascending order  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        List<Employee> list = new ArrayList<>();  
        list.add(new Employee(1, "Ravi", 40000));  
        list.add(new Employee(2, "Amit", 60000));  
        list.add(new Employee(3, "Riya", 50000));  
  
        Collections.sort(list); // Uses compareTo()  
  
        for (Employee e : list)  
            System.out.println(e.name + " " + e.salary);  
    }  
}
```

Output:

```
Ravi 40000  
Riya 50000  
Amit 60000
```

里 2. Comparator Interface

► Definition

Comparator is an interface present in java.util package.

It is used when you want to define multiple ways to sort objects — without changing the original class.

► Syntax

```
public interface Comparator<T> {  
    int compare(T obj1, T obj2);  
}
```

► How it Works

- Used to define custom sorting logic outside the class.
 - You can create multiple Comparator implementations for different sorting criteria.
-

► Example

```
import java.util.*;  
  
class Employee {  
    int id;  
    String name;  
    int salary;  
  
    Employee(int id, String name, int salary) {  
        this.id = id;  
        this.name = name;  
        this.salary = salary;  
    }  
}  
  
// Comparator to sort by Name  
class NameComparator implements Comparator<Employee> {  
    public int compare(Employee e1, Employee e2) {  
        return e1.name.compareTo(e2.name);  
    }  
}  
  
// Comparator to sort by Salary (Descending)  
class SalaryComparator implements Comparator<Employee> {  
    public int compare(Employee e1, Employee e2) {  
        return e2.salary - e1.salary;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        List<Employee> list = new ArrayList<>();  
        list.add(new Employee(1, "Ravi", 40000));  
        list.add(new Employee(2, "Amit", 60000));  
    }  
}
```

```

        list.add(new Employee(3, "Riya", 50000));

        Collections.sort(list, new NameComparator());
        System.out.println("Sorted by Name:");
        for (Employee e : list)
            System.out.println(e.name + " " + e.salary);

        Collections.sort(list, new SalaryComparator());
        System.out.println("\nSorted by Salary (Desc):");
        for (Employee e : list)
            System.out.println(e.name + " " + e.salary);
    }
}

```

Output:

Sorted by Name:
Amit 60000
Ravi 40000
Riya 50000

Sorted by Salary (Desc):
Amit 60000
Riya 50000
Ravi 40000

⚖️ Comparable vs Comparator — Key Differences

Feature	Comparable	Comparator
Package	java.lang	java.util
Method	compareTo(Object o)	compare(Object o1, Object o2)
Sorting logic location	Inside the class	Outside the class
Number of sorting sequences	Only one (natural ordering)	Multiple (custom orders possible)
Used with	Collections.sort(list)	Collections.sort(list, comparator)
Example	Sorting employees by salary (default)	Sorting employees by name, id, or salary dynamically

⚙️ Java 8+ Simplification

You can use lambda expressions for Comparator:

```
Collections.sort(list, (e1, e2) -> e1.name.compareTo(e2.name));
```

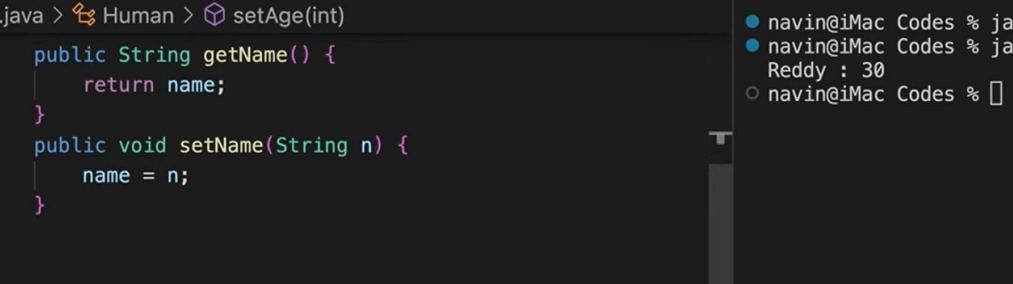
Or even simpler:

```
list.sort(Comparator.comparing(e -> e.salary));
```

```
J Demo.java ×  
J Demo.java > Human > setAge(int)  
1  
2 class Human  
3 {  
4     private int age;  
5     private String name;  
6  
7     public int getAge() {  
8         return age;  
9     }  
10    public void setAge(int a) {  
11        age = a;  
12    }  
13    public String getName() {  
14        return name;  
15    }  
16    public void setName(String n) {  
17        name = n;  
18    }  
19 }
```

TERMINAL ... zsh + ×

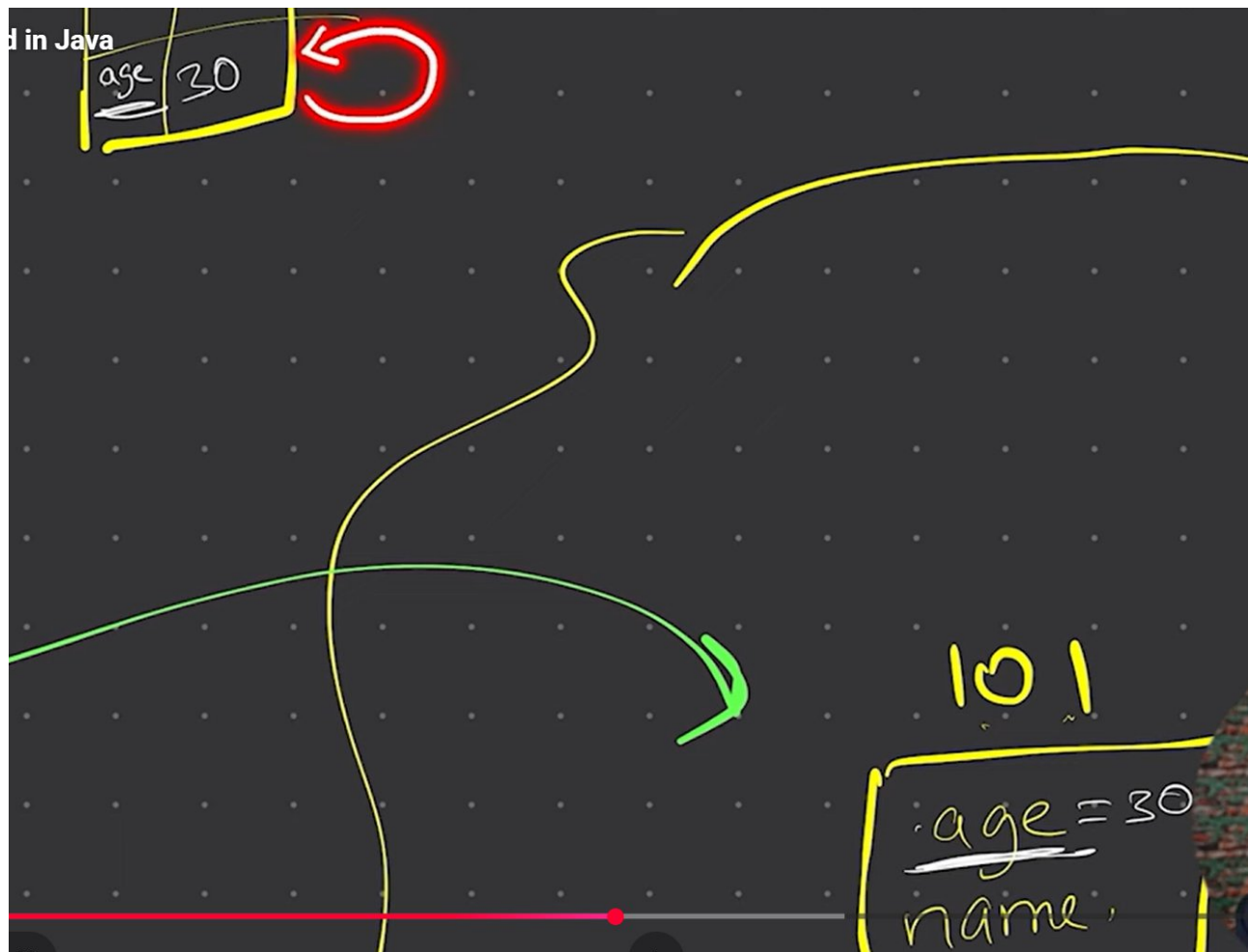
```
navin@iMac Codes % javac Demo.java  
navin@iMac Codes % java Demo  
Reddy : 30  
navin@iMac Codes %
```



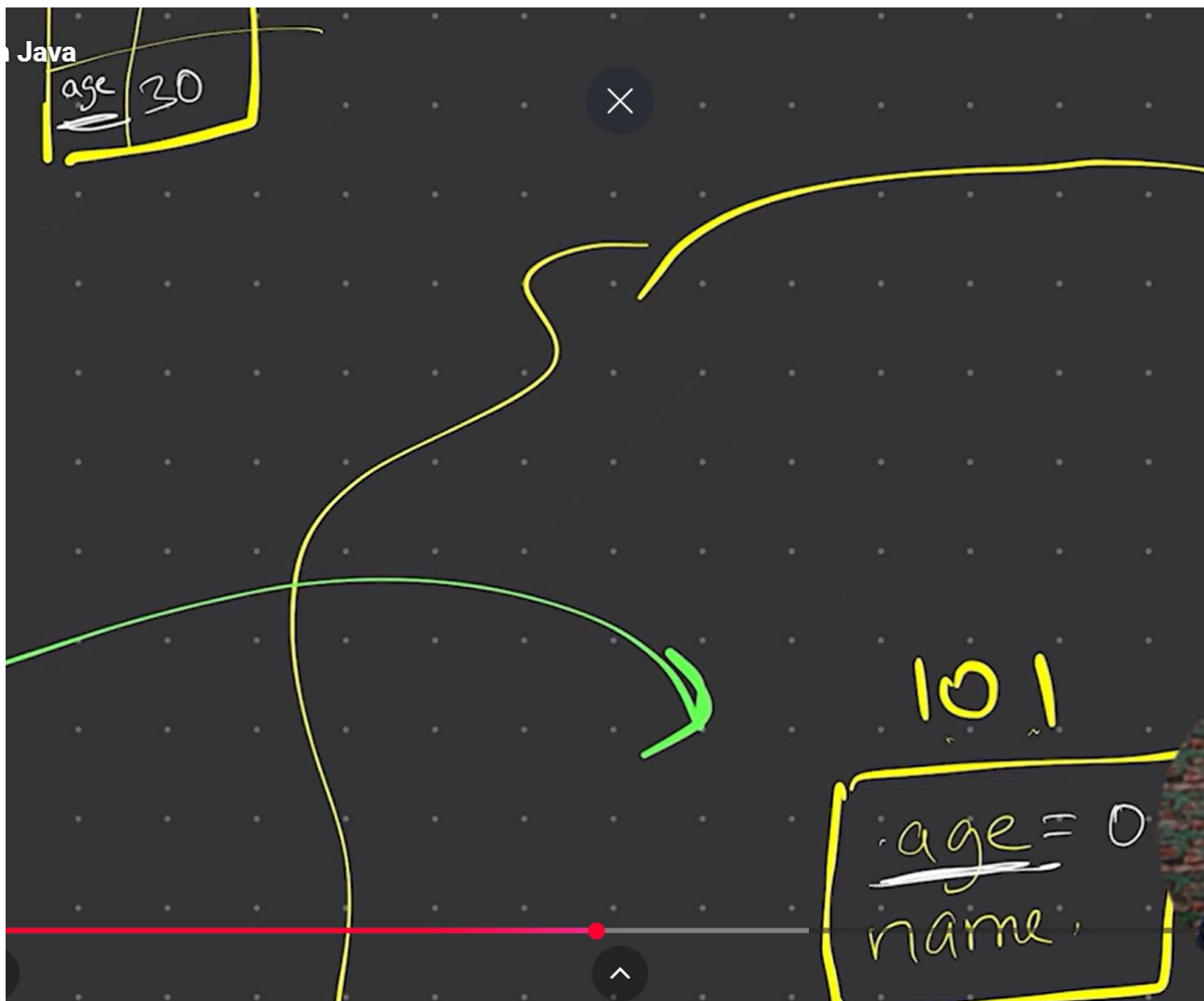
```
Demo.java X
Demo.java > Human > setAge(int)
13 public String getName() {
14     return name;
15 }
16 public void setName(String n) {
17     name = n;
18 }
19
20
21
22 }
23
24 public class Demo
25 {
26     public static void main(String a[])
27     {
28         Human obj = new Human();
29         obj.setAge(a: 30);
30         obj.setName(n: "Reddy");
31
32         System.out.println(obj.getName() + " : " + obj.getAge());
33     }
34 }
```

TERMINAL ...

```
> zsh +
● navin@iMac Codes % javac Demo.java
● navin@iMac Codes % java Demo
Reddy : 30
○ navin@iMac Codes %
```



jh



kj

```
Demo.java x
J Demo.java > Human > setAge(int)
1
2 class Human
3 {
4     private int age;
5     private String name;
6
7     public int getAge() {
8         return age;
9     }
10    public void setAge(int age) {
11        Human obj1 = new Human();
12        obj1.age = age;
13    }
14    public String getName() {
15        return name;
16    }
17    public void setName(String n) {
18        name = n;
19    }
20 }
```

TERMINAL ... zsh + v


```
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
Reddy : 0
navin@iMac Codes %
```

kj

```
J Demo.java x
J Demo.java > Human > setAge(int)
3 {
4     private int age;
5     private String name;
6
7     public int getAge() {
8         return age;
9     }
10    public void setAge(int age) {
11        Human obj1 = new Human();
12        obj1.age = age;
13    }
14    public String getName() {
15        return name;
16    }
17    public void setName(String n) {
18        name = n;
19    }
20
21
22
```

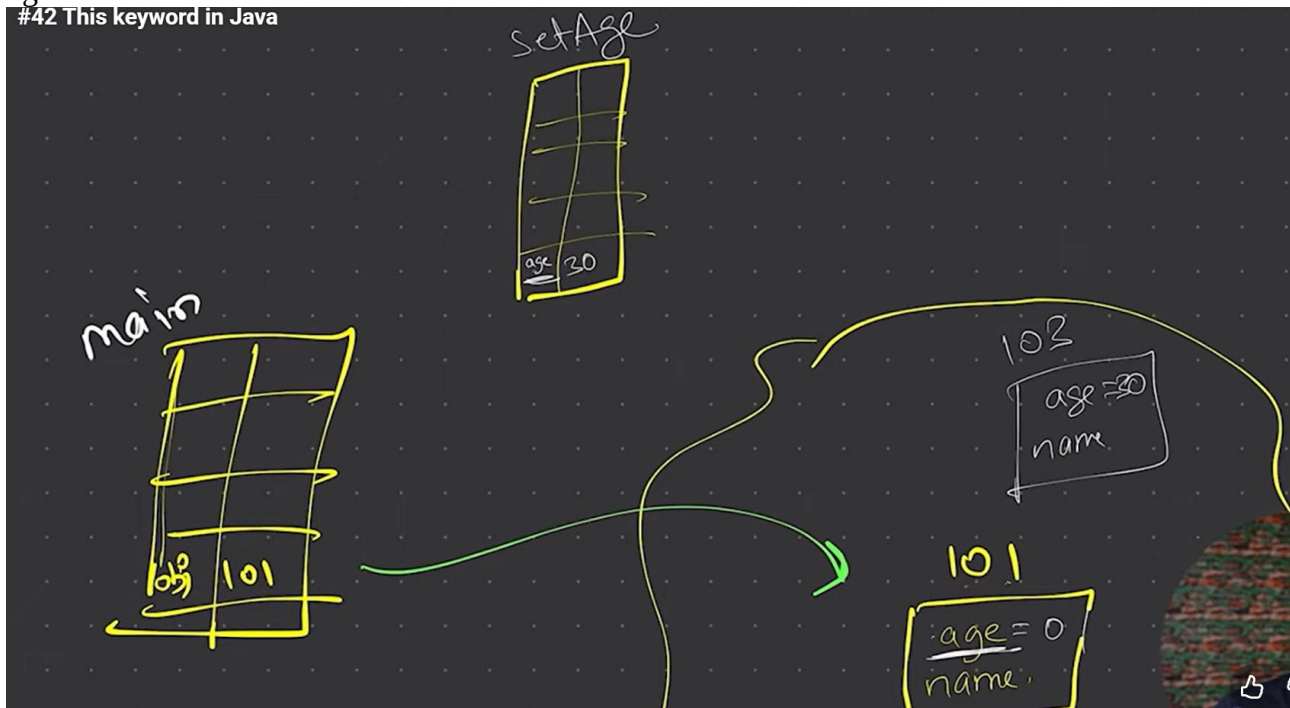
TERMINAL ... zsh + v [] [] < x

- navin@iMac Codes % javac Demo.java
- navin@iMac Codes % java Demo
- Reddy : 0
- navin@iMac Codes %



kgh

#42 This keyword in Java



sfd

```
Demo.java > Human > setAge(int, Human)
8      return age;
9      }
10     public void setAge(int age, Human obj) {
11         Human obj1 = obj;
12         obj1.age = age;
13     }
14     public String getName() {
15         return name;
16     }
17     public void setName(String n) {
18         name = n;
19     }
20
21
22
```

TERMINAL

```
navin@iMac Codes % java Demo
Reddy : 0
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
Reddy : 30
navin@iMac Codes %
```



```

33
34
35 }
36
37 public class Demo
38 {
39     public static void main(String a[])
40     {
41         Human obj = new Human();
42         Human obj1 = new Human(a: 18, n: "Navin");
43         System.out.println(obj.getName() + " : " + obj.getAge(
44
45         // obj.setAge(30);
46         // obj.setName("Reddy");
47
48         // System.out.println(obj1.getName() + " : " + obj1.getA

```

1 charAt() → When you need a specific character from a string

✓ Use when:

You just want **one character** at a specific position (index) from a String.

易 Syntax:

```
char ch = str.charAt(index);
```

- Index starts from 0 (like arrays).
- It returns one character.

燐 Example:

```

public class ExampleCharAt {
    public static void main(String[] args) {
        String name = "Nisha";

        // Get the 1st character (index 0)
        char first = name.charAt(0);

        // Get the 3rd character (index 2)
        char third = name.charAt(2);

        System.out.println("First character: " + first); // Output: N
        System.out.println("Third character: " + third); // Output: s
    }
}

```

✓ When to use:

- When you only need to check or use one or a few characters.
- For example, checking if the first letter is uppercase:

```
if (Character.isUpperCase(str.charAt(0))) {
```

```
        System.out.println("Starts with uppercase!");  
    }
```

`toCharArray()` → **When you need all characters in a String**

✓ **Use when:**

You want to process or modify **all characters** — for example, looping, reversing, counting, etc.

Syntax:

```
char[] chars = str.toCharArray();
```

It converts the whole string into a character array.

Example:

```
public class ExampleToCharArray {  
    public static void main(String[] args) {  
        String name = "Nisha";  
  
        // Convert string to character array  
        char[] letters = name.toCharArray();  
  
        // Loop through each character  
        for (char ch : letters) {  
            System.out.println(ch);  
        }  
    }  
}
```

✓ **When to use:**

- When you want to:
 - Loop through all characters
 - Modify or reverse a string
 - Count vowels, consonants, digits, etc.

Example — count vowels in a string:

```
public class CountVowels {  
    public static void main(String[] args) {  
        String str = "Nisha Prasad";  
        int count = 0;  
  
        for (char c : str.toCharArray()) {  
            if ("aeiouAEIOU".indexOf(c) != -1) {  
                count++;  
            }  
        }  
  
        System.out.println("Number of vowels: " + count);  
    }  
}
```

}

茶 Summary Table

Use Case	Method	Returns	Best For
Get a single character	<code>charAt(index)</code>	One character	Access by position
Work with all characters	<code>toCharArray()</code>	Array of characters	Loops, modify, count, reverse

里 What is an API?

API stands for **Application Programming Interface**.

It's a **bridge** that allows **two software applications** to **communicate** with each other.

➔ It defines **how** different software components should **interact** — what functions are available, what inputs they accept, and what outputs they return.

易 Simple Example:

When you use a **mobile banking app**, it calls APIs to:

- Fetch your account balance
- Send money
- Get transaction history

The app's frontend (UI) calls backend APIs built in Java / Spring Boot, which then talk to the database or core systems.

Example of an API in Java

A simple example in Core Java (without Spring Boot):

```
public class CalculatorAPI {  
    public int add(int a, int b) {  
        return a + b;  
    }  
}
```

Here, the `add()` method is acting like a local API — other parts of your code can call it to perform addition.

泉 In Web Development (Real-world Meaning of API)

When we say API in backend/web development, we usually mean a Web API — it's an interface that allows communication over the internet (typically using HTTP).

Example:

GET `https://api.weather.com/v1/city/Mumbai`

It returns data in JSON or XML format.

Types of APIs (Based on Communication)

Type	Description	Example
Open/Public API	Available to everyone	Google Maps API, Weather API
Internal/Private API	Used within a company	Bank internal APIs
Partner API	Shared between business partners	SBI ↔ TCS systems
Composite API	Combines multiple APIs in one call	Fetch user + account + transaction in one request

Types of Web APIs (Based on Architecture)

API Type	Description	Example
REST API (Representational State Transfer)	Uses HTTP methods (GET, POST, PUT, DELETE). Most popular in Spring Boot	@RestController
SOAP API (Simple Object Access Protocol)	XML-based, older, used in banking & legacy systems	JAX-WS in Java
GraphQL API	Client defines data it needs; single endpoint	Used in modern web apps
gRPC API	Uses Protocol Buffers (binary data) – very fast	Used for microservices (Google-style)

APIs in Core Java

In Core Java, you mostly work with:

- **Java Standard APIs** — classes and packages provided by Java (like `java.util`, `java.io`, `java.sql`)
- **Custom APIs** — methods or modules you write for your own application

Example:

```
import java.util.*;

public class Example {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Nisha");
        System.out.println(names);
    }
}
```

➔ Here, `ArrayList` is part of the Java Collections API.

APIs in Spring Boot

Spring Boot is mainly used to build RESTful APIs.

Example:

```
import org.springframework.web.bind.annotation.*;

@RestController
public class HelloController {

    @GetMapping("/hello")
    public String hello() {
        return "Hello, Nisha!";
    }
}
```

🔗 API Endpoint → GET <http://localhost:8080/hello>

✓ Explanation:

- `@RestController` → defines this class as an API controller
- `@GetMapping("/hello")` → defines the API endpoint
- The function returns a response (in JSON or text)

Other Types of APIs in Spring Ecosystem

API Type	Description	Example
REST API	Communicates using HTTP and JSON	<code>@RestController</code>
SOAP API	XML-based	<code>@WebService</code>
Reactive API	For asynchronous/non-blocking APIs	WebFlux
GraphQL API	Flexible query API	spring-boot-starter-graphql

🍵 Interview Tip (Short Answer)

“API stands for Application Programming Interface. It allows different systems to communicate.

In Core Java, APIs are mainly the built-in libraries and interfaces like Collection API or JDBC API.

In Spring Boot, we typically build RESTful APIs using annotations like `@RestController`, `@GetMapping`, and return JSON responses to clients.”

Statement

► What it is:

- Used to execute simple static SQL queries — no parameters.

- Query is sent directly to the database every time (no pre-compilation).

✓ Example:

```
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM employees");

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

⚠ Disadvantages:

- **Vulnerable to SQL Injection**
 - **Slower for repeated queries (each query is compiled again)**
 - **Not good for dynamic values**
-

2. PreparedStatement

► What it is:

- Used for precompiled SQL queries with placeholders (?).
- The query is compiled once and can be executed multiple times with different values.
- Helps prevent SQL Injection and improves performance.

✓ Example:

```
String sql = "SELECT * FROM employees WHERE department = ?";
PreparedStatement pstmt = connection.prepareStatement(sql);

// Set the value of parameter
pstmt.setString(1, "HR");

// Execute
ResultSet rs = pstmt.executeQuery();

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

✓ Advantages:

- Safer (prevents SQL injection)
 - Faster for repeated execution
 - Easier to read and maintain
-

3. CallableStatement

► What it is:

- Used to call stored procedures from the database.

- Stored procedures are pre-written SQL logic saved in the DB.

✓ Example (Stored Procedure):

Let's say you have a stored procedure in SQL:

```
CREATE PROCEDURE getEmployeeByDept(IN deptName VARCHAR(50))
BEGIN
    SELECT * FROM employees WHERE department = deptName;
END;
```

Now call it from Java:

```
CallableStatement cstmt = connection.prepareCall("{call getEmployeeByDept(?)}");
cstmt.setString(1, "HR");

ResultSet rs = cstmt.executeQuery();

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

✓ Advantages:

- Encapsulates business logic in DB (more secure)
- Reduces Java-side SQL complexity
- Faster execution (DB optimizes stored procedures)

易 Summary Table

Feature	Statement	PreparedStatement	CallableStatement
Purpose	Simple SQL (static)	Parameterized/precompiled SQL	Execute stored procedures
SQL Parameters	Not supported	Supported using ?	Supported using ?
Performance	Slow (compiled each time)	Faster (precompiled)	Faster (DB optimized)
SQL Injection	Vulnerable	Prevented	Prevented
Example Query	SELECT * FROM emp	SELECT * FROM emp WHERE id = ?	{call getEmp(?)}

責 Interview Tip:

A common question is:

“What is the difference between Statement and PreparedStatement?”

✓ Quick Answer:

Statement is used for simple SQL queries without parameters, while PreparedStatement is used for parameterized queries.

PreparedStatement is precompiled, faster, and prevents SQL injection.

What is transient in Java?

transient is a keyword that tells the Java Virtual Machine (JVM) not to serialize a particular variable when an object is being serialized.

In simple words:

If you mark a variable as transient, its value will not be saved when you write the object to a file or send it over the network.

Why is it used?

Sometimes, you don't want all fields of an object to be saved — for example:

- **Sensitive information (like passwords, credit card numbers)**
- **Temporary data (like cached values)**
- **Derived/computed values (that can be recalculated)**

That's where **transient** helps.

✓ Example of transient

Step 1 — Class without transient

```
import java.io.*;

class User implements Serializable {
    String name;
    String password; // we don't want to serialize this ideally

    User(String name, String password) {
        this.name = name;
        this.password = password;
    }
}
```

Now, let's serialize it:

```
public class Test {
    public static void main(String[] args) throws Exception {
        User user = new User("Nisha", "secret123");

        // Serialize the object
        ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("user.txt"));
        oos.writeObject(user);
        oos.close();

        // Deserialize
```



```

        ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("user.txt"));
        User u = (User) ois.readObject();
        ois.close();

        System.out.println("Name: " + u.name);
        System.out.println("Password: " + u.password);
    }
}

```

茶 Output:

Name: Nisha
Password: secret123

Step 2 — Using transient

Now, mark the password as transient:

```

class User implements Serializable {
    String name;
    transient String password; // transient keyword

    User(String name, String password) {
        this.name = name;
        this.password = password;
    }
}

```

茶 Output after running the same code:

Name: Nisha
Password: null

✓ Because the password field was not serialized, its value is lost when the object is deserialized.

易 Key Points

Feature	Description
Keyword	transient
Used In	Serialization
Purpose	Skip certain fields during serialization
Default Value on Deserialize	null for objects, 0 for numbers, false for booleans
Can Be Used With	Instance variables only (not static or final)

⚡ Quick Example in One Line

```
transient int tempData = 100; // won't be saved during serialization
```

1. ABSTRACTION (Concept / OOP Principle)

Abstraction is an **Object-Oriented Programming concept**, not a Java keyword.

✓ Definition:

Abstraction means **hiding internal implementation** and **showing only essential details** to the user.

✓ Real-Life Example:

When you press “**Start**” on a washing machine:

- You don’t see motors, sensors, water pumps.
 - You only see the interface (buttons).
- This is abstraction.

✓ How Abstraction is achieved in Java?

Abstraction can be achieved using:

1. **Abstract classes**
 2. **Interfaces**
-

★ 2. ABSTRACT (Java Keyword)

abstract is a **Java keyword** used to create abstract classes and abstract methods.

✓ abstract class:

A class declared with the **abstract** keyword.

It **may or may not contain abstract methods**.

Example:

```
abstract class Vehicle {  
    abstract void start(); // abstract method (no body)  
}
```

✓ abstract method:

A method declared without implementation (no body).

Example:

```
abstract void start();
```

It must be overridden by subclasses.

★ 🎵 KEY DIFFERENCE (Easy Table for Interviews)

Feature	Abstraction	abstract (Keyword)
Type	OOP concept	Java keyword
Purpose	Hide internal details	Create abstract classes/methods
How	Using abstract classes & interfaces	Written using abstract keyword
Level	Design-level concept	Language-level implementation
Contains code?	No—it's a theory	Yes—its classes/methods contain code signatures
Example	ATM hides implementation	<code>abstract class Bank { abstract void withdraw(); }</code>

★ One-Line Interview Answer

Abstraction is an OOP principle (hiding implementation). `abstract` is a Java keyword used to achieve abstraction through abstract classes and methods.

📁 1 “Tell me about yourself” – Full Answer for THIS JD

You can say this almost word-for-word:

Answer:

“Hi, I’m Nisha. I have a little over 2 years of experience as a Java Backend Developer, mainly working with Core Java, Spring Boot, REST APIs and relational databases like Oracle and MySQL.

Currently I was working at Xybion, where I contributed to an enterprise preclinical trial application called Pristima Core. I worked on backend features and bug fixes in modules like Study Management, Pharmacy and Animal Room. My day-to-day work involved developing and maintaining REST APIs, optimizing SQL and PL/SQL queries, and resolving production issues. For example, I worked on improving a few slow-running queries and we were able to improve performance by around 30% in some screens. I’ve also fixed more than 30 production defects, which helped to reduce recurring issues and support load.

On the side, I’ve built a small Spring Boot CRUD project – an Employee Management System – using Spring Data JPA and MySQL, following a layered architecture with controller, service and repository.

I’m comfortable working in Agile teams, using Git, Jira and Postman on a daily basis. Right now, I’m looking for a role where I can work more deeply on backend design, scalable APIs and microservice-style architectures, and I find your role a good match for my skills in Java, Spring Boot and SQL.”

🗣️ Practice this 3–4 times out loud so it sounds natural.

2 “Explain your project / current work” – Pristima Core

Question: “Can you explain the project you worked on at Xybion?”

Answer:

“Sure. The main project I worked on is called Pristima Core. It’s a preclinical trial management application used for managing animal-based studies before clinical trials.

From a functional side, it helps users manage studies, schedule activities, track animals, manage dosing through the pharmacy module, and maintain related records and reports.

From a technical side, it’s a Java-based backend using Spring and relational databases like Oracle. I mainly worked on the backend part – I handled feature enhancements, bug fixing and performance improvements in modules like Study Management, Pharmacy and the Animal Room.

My responsibilities included:

- Implementing or updating REST APIs and backend logic in Java
- Writing and optimizing SQL / PL/SQL queries
- Debugging production issues based on logs and user issues
- Collaborating with QA to test and verify fixes during sprints.

Overall, the project gave me good exposure to working on a large codebase, understanding domain logic, and improving performance and stability in a live enterprise system.”

3 Example of Production Issue You Fixed

They *love* this type of question.

Question: “Tell me about a challenging bug or production issue you solved.”

Answer:

“One issue I remember clearly was related to a very slow screen in the Study Management module. Users were complaining that a particular report was taking a long time to load.

I started by checking the logs and reproducing the scenario in a lower environment. Then I enabled more detailed logging around the DAO layer and measured the time taken by each step. I found that one SQL query that joined multiple tables was taking most of the time.

I took the query, ran it directly on the database, checked the execution plan and saw that it was not using indexes properly. Together with the team, we simplified the query, added an index on an important column that was used in the WHERE clause, and reduced some unnecessary columns and joins.

After deploying the optimized query, the response time dropped significantly – from around 12–15 seconds to about 3–4 seconds.

This experience improved my understanding of how backend performance is often related to database design and query optimization, not just Java code.

(There was a bug related to the time bleed collection where the processing time was very high due to a large volume of data being fetched from the SQL query. I resolved this issue by applying multiple SQL query filters based on several parameters. Also, I improved the application logic by using the Parallel Stream API, which significantly enhanced the overall performance)

1. `volatile` in Java

✓ What is `volatile`?

`volatile` is a keyword used with **variables** to ensure **visibility of changes across multiple threads**.

顯 It tells the JVM:

“Always read this variable from **main memory**, not from a thread’s local cache.”

✓ Why do we need `volatile`?

In **multithreading**, each thread may cache variables locally.

Without **`volatile`**, one thread’s update **may not be visible** to other threads.

✓ What problems does `volatile` solve?

- ✓ Visibility problem
 - ✓ Prevents instruction reordering (partially)
 - ✗ Does NOT provide atomicity
-

✗ Problem without `volatile`

```
class FlagExample {
    boolean running = true;

    void stop() {
        running = false;
    }

    void run() {
        while (running) {
            // do work
        }
    }
}
```

✗ Issue:

- Thread A updates `running = false`
 - Thread B may never see the update
 - Infinite loop possible
-

✓ Correct usage with volatile

```
class FlagExample {
    volatile boolean running = true;

    void stop() {
        running = false;
    }

    void run() {
        while (running) {
            // do work
        }
    }
}
```

✓ Now:

- Update is immediately visible to all threads
-

⚠ Important Interview Point

`volatile` ensures visibility, NOT thread safety.

✗ This is NOT safe even with `volatile`:

```
volatile int count = 0;

count++; // NOT atomic
```

For atomicity → use `synchronized` or `AtomicInteger`.

Practical Use Case of volatile

Shutdown flag

```
volatile boolean shutdown = false;
```

Used in:

- Thread stop signals
- Status flags
- Configuration flags
- Double-checked locking (with care)

2. transient in Java

✓ What is transient?

transient is a keyword used with variables to indicate that they should NOT be serialized.

顔 It tells the JVM:

“Do NOT save this variable when the object is converted to bytes.”

✓ Why do we need transient?

Serialization converts an object into a byte stream (for:

- File storage
- Network transfer
- Caching)

Some fields:

- Are sensitive
- Are temporary
- Can be recreated

So we exclude them using transient.

✗ Without transient

```
class User implements Serializable {  
    String username;  
    String password;  
}
```

✗ Problem:

- Password gets written to file/network
 - Security risk
-

✓ Correct usage with transient

```
class User implements Serializable {  
    String username;  
    transient String password;  
}
```

✓ Result:

- password is NOT serialized
- On deserialization → password = null

Example: Serialization behavior

```
User u = new User();
u.username = "nisha";
u.password = "secret";
```

After deserialization:

```
username = "nisha"
password = null
```

Practical Use Cases of transient

- ✓ Passwords
- ✓ OTPs
- ✓ Session objects
- ✓ Cache data
- ✓ Logger objects

3. KEY DIFFERENCES: volatile vs transient

Feature	volatile	transient
Purpose	Thread visibility	Serialization control
Used in	Multithreading	Serialization
Affects	JVM memory model	Object persistence
Ensures	Latest value visible	Field not saved
Applies to	Variables	Variables
Default value impact	No	Yes (null / 0)
Thread safety	✗ No	✗ Not related

4. Real-World Examples (Interview Gold ★)

✓ volatile – Real-World Scenario

Server shutdown flag

```
volatile boolean isServerRunning = true;
```


- Admin thread sets it to false
 - Worker threads stop gracefully
 - Ensures visibility across threads
-

✓ **transient – Real-World Scenario**

Banking application

```
class Account implements Serializable {  
    String accountNumber;  
    transient String atmPin;  
}
```

- Account number saved
 - ATM PIN NOT saved
 - Security compliance maintained
-

5. Can a variable be both volatile and transient?

✓ YES

```
transient volatile boolean isActive;
```

Meaning:

- Not serialized
- Visible across threads

Used rarely, but valid.

6. One-Line Interview Answers (Very Important)

✓ **volatile**

volatile ensures that changes made by one thread are immediately visible to other threads.

✓ **transient**

transient prevents a variable from being serialized and saved.

7. Common Interview Follow-ups (Be Ready)

Q: Does volatile make code thread-safe?

✗ No, only visibility.

Q: Does transient affect static variables?

Static variables are not serialized anyway.

Q: What value does transient field get after deserialization?

Default value (null, 0, false).

1. What are `wait()`, `notify()`, and `notifyAll()`?

They are **methods of `java.lang.Object`**, not of `Thread`.

類 Used for **inter-thread communication** — allowing threads to **pause and resume execution** based on conditions.

2. Why are they in `Object` class and not in `Thread`?

Because **locks are on objects, not threads**.

- Every object in Java has a monitor (lock)
 - Threads acquire locks on objects
 - So waiting and notification are object-level operations
-

3. `wait()` — Detailed Explanation

✓ What does `wait()` do?

- Causes the current thread to release the lock
 - Moves the thread to waiting state
 - Thread stays waiting until:
 - `notify()` OR
 - `notifyAll()` is called on the **same object**
-

! Important Rules for `wait()`

1. Must be called inside synchronized block/method

2. Releases the object lock
 3. Thread enters WAITING or TIMED_WAITING state
-

✗ Illegal usage

```
obj.wait(); // IllegalMonitorStateException
```

✓ Correct usage

```
synchronized(obj) {  
    obj.wait();  
}
```

4. notify() — Detailed Explanation

✓ What does notify() do?

- Wakes up ONE random thread waiting on that object
 - Does NOT release the lock immediately
 - The awakened thread competes for the lock
-

Important points

- ✓ Must be inside synchronized block
 - ✓ Not deterministic — any waiting thread may be chosen
-

5. notifyAll() — Detailed Explanation

✓ What does notifyAll() do?

- Wakes up ALL waiting threads on that object
 - All threads compete for the lock
 - Prevents thread starvation
-

When to use?

- Multiple threads waiting
- When condition change affects all

- Safer than notify()

6. Simple Producer–Consumer Example (VERY IMPORTANT)

Shared Resource

```
class Shared {  
    int data;  
    boolean available = false;  
}
```

Producer

```
class Producer extends Thread {  
    Shared s;  
  
    Producer(Shared s) {  
        this.s = s;  
    }  
  
    public void run() {  
        synchronized(s) {  
            s.data = 10;  
            s.available = true;  
            s.notify();    // notify consumer  
        }  
    }  
}
```

Consumer

```
class Consumer extends Thread {  
    Shared s;  
  
    Consumer(Shared s) {  
        this.s = s;  
    }  
  
    public void run() {  
        synchronized(s) {  
            while (!s.available) {  
                try {  
                    s.wait();    // wait until producer produces  
                } catch (InterruptedException e) {}  
            }  
            System.out.println("Consumed: " + s.data);  
        }  
    }  
}
```

7. Why use while instead of if with wait()? (INTERVIEW FAVORITE)

```
while (!condition) {  
    wait();  
}
```

✓ Reason:

- Handles spurious wakeups
 - Ensures condition is rechecked
 - Prevents incorrect execution
-

8. Difference Between wait(), notify(), and notifyAll()

Feature	wait()	notify()	notifyAll()
Releases lock	✓ Yes	✗ No	✗ No
Wakes threads	✗ No	✓ One	✓ All
Lock needed	✓ Yes	✓ Yes	✓ Yes
Thread state	WAITING	BLOCKED	BLOCKED
Use case	Pause thread	Wake one	Wake all

9. Real-World Practical Examples

✓ Example 1: Printer Queue

- One printer
 - Multiple jobs waiting
 - wait() → when printer busy
 - notify() → wake next job
-

✓ Example 2: Banking System

- Balance = 0
- Withdraw waits
- Deposit notifies

```
synchronized(account) {  
    while (balance == 0) {  
        account.wait();  
    }
```

```
}  
    balance -= amount;  
}
```

10. Common Interview Questions & Answers

? Can we call wait() outside synchronized block?

✗ No → IllegalMonitorStateException

? Does notify() release lock?

✗ No → lock released after synchronized block ends

? Why notifyAll() is safer?

✓ Prevents deadlock and starvation

? Difference between sleep() and wait()?

- sleep() → does NOT release lock
 - wait() → releases lock
-

11. One-Line Interview Answers (MEMORIZE)

✓ **wait()** → Releases lock and waits

✓ **notify()** → Wakes one waiting thread

✓ **notifyAll()** → Wakes all waiting threads

12. When NOT to use them (Senior-level Tip)

In modern Java, prefer:

- BlockingQueue
- Semaphore
- ReentrantLock

- Condition

But interviews still expect **wait/notify basics**.

Question 4: Difference between Interface & Abstract Class

Feature	Interface	Abstract Class
Methods	Abstract + default	Abstract + concrete
Variables	public static final	Any
Multiple inheritance	✓ Yes	✗ No
Constructor	✗ No	✓ Yes
State	✗ No	✓ Yes

Use Interface when:

- Multiple inheritance needed
- Contract-based design
- Example: Runnable, Comparable

Use Abstract Class when:

- Common base logic
- Partial implementation
- Example: HttpServlet

撤 Interview line:

Interface defines what to do, abstract class defines how to do.

Core Idea (Very Simple)

Fail-Fast Iterator

- Throws **ConcurrentModificationException**
- If collection is modified while iterating
- Works on **original** collection

Fail-Safe Iterator

- Does **NOT** throw exception
- Iterates over a **copy** of collection
- Safe in multi-threaded environments

