CORE JAVA — INTERVIEW QUESTIONS (Based on JD + Her Resume)

1. Fundamentals

1. What is Java? Why is it platform-independent?

■ Explanation (simple, conceptual)

Java is a **high-level, object-oriented programming language** used to build applications like web apps, mobile apps, enterprise systems, and microservices.

It follows the principle **"Write Once, Run Anywhere (WORA)"**, meaning the same Java program can run on any operating system.

■ Why is Java platform-independent?

Because Java does **not** run directly on the OS.

Instead, Java code is compiled into **Bytecode**, and Bytecode runs on **JVM** (Java Virtual Machine).

Every operating system (Windows, Linux, Mac) has its own version of JVM.

So the code remains the same — only JVM changes.

Flow:

Java Code → Compiler → Bytecode → JVM → Runs on any OS

■ Example

If you write a Java program on Windows and send the `.class` file to a Linux machine, it will still run — because Linux has its own JVM.

■ Short interview summary

"Java is a portable, high-level OOP language. It is platform-independent because Java programs compile into Bytecode, which runs on JVMs available for all operating systems."

2. Explain JVM, JRE, and JDK with examples.

■ Explanation

JVM (Java Virtual Machine)

- A virtual runtime environment that executes Java Bytecode.
- It performs memory management, garbage collection, and also handles security.
- JVM is **platform-dependent**.

JRE (Java Runtime Environment)

- Contains **JVM + Libraries + Runtime files**.
- Used to **run** Java programs.
- Does NOT contain compilers.

JDK (Java Development Kit)

- Contains **JRE + Development tools** like compilers (javac), debugger, jar tool, etc.
- Used to **develop + run** Java programs.

■ Example

Component	Purpose	Example Tools
JVM	Executes Bytecode	Execution engine, GC
JRE	Runs Java apps	JVM + core libraries
JDK	Writes + compiles + runs	`javac`, `java`, `jar`

■ Short interview summary

"JVM executes Bytecode, JRE provides runtime libraries plus JVM, and JDK provides everything (JRE + compiler) needed to develop and run Java apps."

---

3. What happens when you run a Java program internally?

■ Explanation (simple, step-by-step)

Step 1 – Writing Code

You write a `.java` file.

Step 2 – Compilation (`javac`)

The Java compiler converts source code into **Bytecode** (`.class` file).

Compiler also checks:

- Syntax
- Semantic errors
- Type checking

Step 3 – Class Loader Loads the Bytecode

Part of JVM. Tasks:

- Loads classes into memory
- Verifies classes
- Allocates memory for class structures

Step 4 – Bytecode Verifier

Checks for illegal code, security breaches, stack overflow issues.

Step 5 – Execution by JVM

JVM executes the Bytecode using:

- **Interpreter** → Executes line by line
- **JIT Compiler** → Converts repeated Bytecode into machine code for performance

Step 6 – Garbage Collection

JVM automatically removes unused objects from memory.

■ Example Summary

```
Program.java → javac → Program.class → JVM → Output
```

■ Short interview summary

"When we run a Java program, the compiler converts it to Bytecode, class loader loads it, verifier checks it, and JVM executes it using an interpreter + JIT compiler."

---

4. What is bytecode? Why does Java use bytecode?

■ Explanation

Bytecode is an intermediate code generated after compilation of Java source code.

It is **not** machine code. It is a universal format understood by all JVMs.

File extension: `.class`

■ Why does Java use Bytecode?

1. **Platform-independence** → Same Bytecode runs everywhere.
2. **Secure execution** → JVM validates Bytecode before execution.
3. **Performance** → JIT compiles Bytecode into optimized machine code at runtime.
4. **Portability** → Only JVM changes for each OS, not the Bytecode.

■ Example

```
javac Test.java
→ creates Test.class (Bytecode)
```

You can send `Test.class` to Windows, Linux, or Mac — it will run anywhere.

■ Short interview summary

"Bytecode is platform-neutral intermediate code that JVM executes. It makes Java portable, secure, and efficient."

---

5. Difference between JDK 8 and JDK 17 features?

■ Explanation (most commonly asked differences)

Both are LTS (Long-Term Support) versions, but **JDK 17 is faster, more secure, and modern**.

JDK 8 Major Features

1. **Lambda Expressions**
2. **Functional Interfaces**
3. **Stream API**
4. **Optional Class**
5. **Default methods in interfaces**
6. **Date & Time API (java.time)**

> JDK 8 introduced modern functional programming in Java.

JDK 17 Major Features

1. **Sealed Classes** → Control inheritance
2. **Records** → Lightweight immutable data carriers
3. **Switch Expressions**
4. **Pattern Matching for instanceof**
5. **Improved Garbage Collectors**
6. **Hidden classes**, stronger encapsulation
7. **Removed older modules (e.g., Applet, Nashorn JS engine)**
8. **Better performance (JIT + GC improvements)**

Simplified Comparison Table

Feature	JDK 8	JDK 17
Year	2014	2021
Style	Functional + OOP	Modern, performance-focused
Key Concept	Streams + Lambdas	Records + Sealed Classes
GC	Parallel, CMS (deprecated)	ZGC, G1 improvements
Security	Basic	Much stronger encapsulation
Pattern matching	❌ No	✔ Yes
Switch expressions	❌ No	✔ Yes

■ Short interview summary

"JDK 8 introduced functional programming (streams, lambdas), while JDK 17 modernized Java with records, sealed classes, switch expressions, advanced GC, and major performance/security improvements."

---

2. OOP Concepts

---

1. Explain the four pillars of OOP with real examples from your projects.

■ Explanation (simple, conceptual)

OOP (Object-Oriented Programming) is based on **four main principles**:

1. Encapsulation — “Data + methods bundled together”

Hiding data and exposing only required behavior through methods.

2. Abstraction — “Show only essential details”

Users interact with functionalities without knowing internal logic.

3. Inheritance — “Reuse code from parent to child”

A child class inherits features from a parent class.

4. Polymorphism — “Same name, different behavior”

Allows methods to behave differently based on context.

■ Real examples from your own projects

🔷 Encapsulation (Expense Tracker Project)

You used **private fields** in the Expense entity and exposed only getter/setter methods:

```
private double budget;
public void setBudget(double budget) { this.budget = budget; }
```

This prevented direct modification and ensured **secure data access**.

🔷 Abstraction (Fraud Detection Middleware)

You exposed a simple API:

```
/validateTransaction
```

Internally it performed fraud checks, ML rules, and scoring —

but clients only saw **high-level functionality**, not internal logic.

🔷 Inheritance (Spring Boot Components)

Controllers extended base classes or reused common service logic:

```
public class BaseService {
  public void logRequest(){}
}
public class FraudService extends BaseService {}
```

Code reuse + consistency.

🔷 Polymorphism (Strategy pattern for expense classification)

Different expense calculation strategies implemented the same interface:

```
interface Calculator { double calculate(); }
class FoodCalculator implements Calculator { ... }
class TravelCalculator implements Calculator { ... }
```

Same method, different behavior.

■ Short summary to speak in interview

“In my projects, I used encapsulation for securing entity fields, abstraction through REST APIs hiding business logic, inheritance to reuse service logic, and polymorphism using different strategy implementations for calculations.”

✅ 2. Difference between abstraction vs encapsulation in Java

■ Explanation

Abstraction → Hiding internal implementation and showing only features.

- Achieved by **interfaces, abstract classes**.
- Focuses on **what the object does**.

Encapsulation → Binding data and methods and protecting data.

- Achieved by **private fields + getters/setters**.
- Focuses on **how data is stored and accessed**.

■ When/Why used

Concept	When Used	Why
Abstraction	Designing API endpoints, services	To hide complexity
Encapsulation	Handling sensitive data	To protect data integrity

---

■ Example

Abstraction:

```
interface Payment {
  void pay();
}
```

Encapsulation:

```
private String cardNumber;
public void setCardNumber(String number) { this.cardNumber = number; }
```

■ Short interview summary

“Abstraction hides implementation; encapsulation hides data. Abstraction is about functionality; encapsulation is about data protection.”

---

3. Can we achieve 100% abstraction using abstract classes?

■ Explanation

No.

Abstract classes can contain:

- **abstract methods (unimplemented)**
- **concrete methods (with implementation)**

So they cannot guarantee 100% abstraction.

Only **interfaces** (before Java 8) provided 100% abstraction.

In Java 8+, default/static methods were added, but **interfaces still allow full abstraction**.

■ When/Why used

Use **abstract class** when:

- You want some shared code
- You want constructors
- You want non-static variables

Use **interface** when:

- You want strict abstraction
- You want multiple inheritance

■ Example

```
abstract class A {
   abstract void test();
   void log(){ System.out.println("logging"); }  // concrete method
}
```

■ Short interview summary

“No, abstract classes can’t provide 100% abstraction because they can contain concrete methods. Only interfaces can give full abstraction.”

---

4. Why is multiple inheritance not allowed in Java?

■ Explanation

Java avoids multiple inheritance of classes to prevent the **Diamond Problem**.

Diamond Problem Example:

```
A
↙ ↘
B   C
 ↘ ↙
  D
```

If both B and C have a method `display()`, and D extends both,

Java won’t know **which display() to use**.

■ When/Why

- Avoids ambiguity
- Keeps Java simple
- Prevents complexity in method resolution

- Reduces runtime errors

■ Example

```
class A { void show(){} }
class B { void show(){} }

// class C extends A, B  // NOT allowed
```

■ Short interview summary

“Java avoids multiple inheritance to prevent ambiguity and the Diamond Problem. Instead, Java allows multiple inheritance through interfaces.”

5. How does method overriding work internally?

■ Explanation

Method overriding means a child class provides **its own implementation** of a parent method.

Internally, overriding is implemented using **dynamic method dispatch (runtime polymorphism)**.

JVM decides which method to call at runtime based on object type, not reference type.

■ How JVM Works Internally

Step 1 — Two classes contain same method signature

```
class Parent { void show(){} }
class Child extends Parent { void show(){} }
```

Step 2 — Reference vs Object

```
Parent p = new Child();
p.show(); // calls Child.show()
```

Step 3 — JVM Method Table (v-table)

Each class has a method table with function pointers.

At runtime, JVM:

1. Looks at the **actual object type** (Child)
2. Searches the child's method table
3. Calls the child's overridden method

■ Example

```
Parent obj = new Child();
obj.show(); // runtime binding to Child.show()
```

■ Short interview summary

"Overriding is implemented using dynamic method dispatch. JVM picks the method at runtime by checking the actual object type, using a virtual method table."

---

3. Classes, Objects & Keywords

✅ 1. Difference between `this` and `super`

■ Explanation (simple, conceptual)

Both are **keywords** but used in different contexts:

`this` keyword → Refers to CURRENT class object

Used when:

- Accessing current class variables
- Calling current class methods
- Calling another constructor in the same class

`super` keyword → Refers to PARENT class object

Used when:

- Accessing parent class variables
- Calling parent class methods

- Calling parent class constructor

■ When/Why it is used

Use Case	this	super
Access current class variable	✓	✗
Access parent class variable	✗	✓
Resolve variable shadowing	✓	✓
Call parent constructor	✗	✓
Call another constructor (same class)	✓	✗

■ Example

```
class Parent {
    int x = 10;
}

class Child extends Parent {
    int x = 20;

    void show() {
        System.out.println(this.x);  // 20
        System.out.println(super.x); // 10
    }
}
```

■ Short summary

"`this` refers to current object, `super` refers to parent object. `this` is for current class access, `super` is for parent class access."

✅ 2. What is the purpose of the `final` keyword?

■ Explanation

`final` is used to restrict modification.

Java provides **3 uses**:

1. final variable → value cannot change

```
final int MAX = 100;
```

2. final method → cannot be overridden

Used to prevent changes in behavior.

3. final class → cannot be inherited

Used for security and design stability.

■ When/Why used

- To create constants
- To prevent unwanted inheritance
- To prevent overriding in sensitive logic
- To improve performance (JVM optimization possible)

■ Example

```
final class A {}          // cannot extend
final void test() {}      // cannot override
final int age = 25;        // cannot reassign
```

■ Short summary

“final restricts modification: final variable → no reassignment, final method → no overriding, final class → no inheritance.”

✅ 3. What is a static method? Can static methods be overridden?

■ Explanation

A **static method belongs to the class**, not the object.

You can call it using the class name.

Static methods:

- Cannot access non-static variables
- Are loaded when class loads
- Never use `this` or `super`

■ Can static methods be overridden?

❌ **No. Static methods cannot be overridden.**

But they can be **hidden**, not overridden.

Why?

- Overriding works on objects.
- Static methods belong to **class**, not object.

So JVM binds static methods at **compile time (static binding)**.

■ Example

```
class A {
    static void test() { System.out.println("A"); }
}

class B extends A {
    static void test() { System.out.println("B"); }
}

A obj = new B();
obj.test(); // Calls A.test (method hiding)
```

■ Short summary

"Static methods belong to class, not object. They cannot be overridden, only hidden, because they use compile-time binding."

✅ 4. What is object cloning? Types of cloning?

■ Explanation

Object cloning means creating a **copy** of an existing object.

Java provides cloning using:

```
clone() method from Object class
```

■ Types of Cloning

1. Shallow Cloning

- Only **primitive data + object references** are copied
- Referenced objects **are NOT duplicated**
- Faster, default behavior of `clone()`

2. Deep Cloning

- Everything is copied
- Referenced objects **ARE duplicated**
- More expensive but safer

■ Example

Shallow Clone

```
class Person implements Cloneable {
    int age;
    Address address;

    public Person clone() throws CloneNotSupportedException {
        return (Person) super.clone();
    }
}
```

Deep Clone

```
Person clone = new Person();
clone.address = new Address(original.address);
```

Or using serialization.

■ Short summary

“Cloning creates a copy of an object. Shallow clone copies references; deep clone copies everything including referenced objects.”

✅ 5. What is a constructor? Types of constructors?

■ Explanation

A constructor is a **special method used to create and initialize objects**.

Characteristics:

- Same name as class
- No return type
- Called automatically during object creation

■ Types of Constructors

1. Default Constructor

Provided automatically by Java if no constructor is written.

2. No-Argument Constructor

Explicitly created without parameters.

3. Parameterized Constructor

Used when you want to initialize object with dynamic values.

■ Example

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

    User() {                    // No-arg constructor
        this.name = "Unknown";
    }

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

■ Short summary

“A constructor initializes objects. Types are: default, no-arg, and parameterized constructors.”

---

4. Strings

✅ 1. Why is String immutable in Java?

■ Explanation (simple, conceptual)

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

If you modify it, Java creates a **new String object**.

This immutability is **intentional and crucial**.

■ Why Java made String immutable (5 important reasons)

1. Security (MOST IMPORTANT)

Strings are used in:

- Database URLs
- Usernames
- Passwords
- File paths
- Class names
- Network connections

If Strings were mutable, someone could change them after creation:

```
String password = "admin";
```

Malicious code could modify it → security risk.

2. String Pool optimization

String Pool stores **only immutable objects**.

If Strings were mutable, the same pooled string could change for every reference → breaking the JVM memory model.

3. Thread safety

Immutable objects are **automatically thread-safe**.

So multiple threads can use the same String without synchronization.

4. Caching hashCode()

Since Strings are immutable, their hashCode never changes.

This makes Strings perfect for **HashMap keys, HashSet**, etc.

---

5. Performance optimization

JVM can safely:

- Reuse string objects
- Cache them
- Share across threads
- Avoid defensive copies

■ Short summary

"Strings are immutable for security, efficient memory (String Pool), thread-safety, and performance reasons."

✅ 2. Difference between String, StringBuilder, and StringBuffer

■ Explanation

Feature	String	StringBuilder	StringBuffer
Mutability	❌ Immutable	✓ Mutable	✓ Mutable
Thread-safe	✓ Yes (implicit)	❌ No	✓ Yes (synchronized)
Performance	Slow (new object each time)	Fastest	Slower than Builder
Use Case	Fixed data	Fast operations	Multi-threading

■ When/Why used

String

- When data never changes
- Password, username, API keys
- HashMap keys

StringBuilder

- When string changes frequently
- Best choice for performance

- Used in loops

StringBuffer

- When working in multi-threaded environments
- Rarely used today (replaced by modern concurrency tools)

■ Example

```
String s = "Hello";
s = s + " World";  // creates new object

StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");  // modifies same object

StringBuffer sbf = new StringBuffer("Hello");
sbf.append(" World"); // thread-safe
```

■ Short summary

"String is immutable, StringBuilder is mutable and fastest, StringBuffer is mutable and thread-safe."

✅ 3. How does String's hashCode() work?

■ Explanation

String's `hashCode()` is based on **characters inside the string**.

Java uses this formula:

```
hash = 31 * hash + charAt(i)
```

31 is chosen because:

- It's a prime number
- Reduces collisions
- Faster multiplication ($31 = 2^5 - 1$)

■ When/Why important

- When Strings are used as **HashMap keys**

- Ensures efficient lookup
- Immutable strings guarantee same hash always

■ Example

For `"ABC"`:

```
hash = 0
hash = 31*0 + 'A' = 65
hash = 31*65 + 'B' = 2111 + 66 = 2177
hash = 31*2177 + 'C' = 67587 + 67 = 67654
```

■ Short summary

"String's hashCode is computed using a 31-multiplier formula based on characters, making it fast and collision-resistant."

✅ 4. What is String Pool? Example?

■ Explanation

String Pool is a **special memory area inside the Heap** where Java stores **unique String objects**.

If two strings have the same value, Java stores only **one copy** in the pool.

This saves memory and improves performance.

■ When/Why String Pool is used

- To avoid creating duplicate string objects
- To improve performance
- To reduce memory usage in large applications

■ Example

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

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

Here, `"Java"` is created only **once** in the String Pool.

But:

```
String s3 = new String("Java");
System.out.println(s1 == s3); // false
```

`new String()` always creates a **new object in heap**, not in pool.

To force it into pool:

```
s3.intern();
```

■ Short summary

"String Pool stores unique strings to save memory. Literal strings share the same object, while `new String()` creates a new one."

---

5. Collections Framework

✅ 1. Explain the hierarchy of the Collections framework.

■ Explanation (simple, conceptual)

The **Collections Framework** is a unified architecture to store and manipulate groups of objects.

Its hierarchy starts from the **Collection interface** and branches into **List, Set, and Queue**.

The root interfaces:

- **Iterable**
  - **Collection**
    - **List** → Ordered, allows duplicates
      - ArrayList
      - LinkedList
      - Vector
      - Stack
    - **Set** → Unordered, no duplicates
      - HashSet

        - LinkedHashSet
        - TreeSet
    - **Queue** → FIFO structure
        - PriorityQueue
        - ArrayDeque

Outside the Collection:

- **Map** (NOT part of Collection interface)
    - HashMap
    - LinkedHashMap
    - TreeMap
    - ConcurrentHashMap
    - WeakHashMap
    - Hashtable

■ When/Why this hierarchy is used

- To choose the correct data structure for performance requirements
- To ensure consistent APIs (add, remove, size, iterator)
- To allow switching implementation (e.g., List → ArrayList or LinkedList)

■ Example

```
List<Integer> list = new ArrayList<>();
Set<String> set = new HashSet<>();
Map<Integer, String> map = new HashMap<>();
```

■ Short summary

"The Collections hierarchy divides structures into List, Set, Queue, and Map—each designed for specific use-cases like order, uniqueness, or key-value storage."

✅ 2. How does HashMap work internally? (VERY IMPORTANT)

■ Explanation (simplified but deep)

A HashMap stores key-value pairs using **buckets**.

Each bucket is a **linked list or balanced tree (red-black tree)**.

Steps when you put a key in HashMap:

1. Compute hash

```
hash = key.hashCode()
```

HashMap applies another hash function to reduce collision chance.

---

2. Determine bucket index

```
index = hash % arrayLength
```

---

3. Insert Entry

Three cases:

Case 1 → Bucket empty

Insert directly.

Case 2 → Bucket contains entries (collision)

HashMap compares **equals()**:

- If same key → update value
- Else → add to linked list or convert to tree if size ≥ 8 (JDK 8+)

Case 3 → Tree node present

Binary search tree logic based on hash differences.

Retrieving value:

1. Compute hash
2. Find bucket
3. Compare keys using equals()
4. Return value

■ Key improvements in JDK 8

- Linked list → Tree (Red-Black Tree) when bucket grows large

- Reduces worst-case lookup from **O(n)** to **O(log n)**

■ Short summary

"HashMap stores entries in buckets using hashing. Collisions are handled using linked lists or trees, and key retrieval uses both hashCode() and equals()."

✅ 3. What happens when two keys have same hashcode?

■ Explanation

If two keys have the same hashCode, they will go to the **same bucket**.

This is called a **collision**.

HashMap handles collisions using:

1. equals() check

If equals() returns:

- **true** → values are updated (same key)
- **false** → treat as separate keys

---

2. LinkedList or Tree Structure

- ≤ 8 entries → LinkedList
- 8 entries → Converted to Red-Black Tree (JDK 8+)

This improves performance.

■ Example

```
"FB".hashCode() == "Ea".hashCode()  // true
```

But since `equals()` is different, both get inserted.

■ Short summary

"When two keys share a hashcode, HashMap places them in the same bucket and differentiates them using equals()."

✅ 4. Difference between HashMap and ConcurrentHashMap

■ Explanation

Both store key-value pairs, but differ in thread-safety and performance.

Feature	HashMap	ConcurrentHashMap
Thread-safe	❌ No	✓ Yes
Locks	None	Uses **segment-level / bucket-level locking**
Null keys allowed	✓ Yes	❌ No
Fail-safe	❌ Fail-fast	✓ Fail-safe
Performance	Faster in single-threaded	Faster in multi-threaded
Iterators	Fail-fast	Weakly consistent

■ When/Why used

- Use **HashMap** → when working in single-thread environment.
- Use **ConcurrentHashMap** → when multiple threads read/update frequently.

■ Short summary

"HashMap is not thread-safe, while ConcurrentHashMap provides high-performance thread safety using fine-grained locking."

✅ 5. Why is ArrayList not thread-safe?

■ Explanation

ArrayList is **not synchronized**, meaning:

- Multiple threads can add/remove simultaneously
- Internal array may get corrupted
- IndexOutOfBounds can occur

ArrayList prioritizes **speed over safety**.

■ When/Why this is important

If multiple threads modify ArrayList at same time:

- Data inconsistency occurs

- JVM exceptions may arise

Use:

- `Collections.synchronizedList()`
- `CopyOnWriteArrayList` (for frequent reads)

■ Example

```
List<Integer> list = new ArrayList<>();
```

No locking → unsafe for multi-threading.

■ Short summary

"ArrayList is not thread-safe because it is unsynchronized and allows concurrent modifications without locks."

✅ 6. When to use LinkedList over ArrayList?

■ Explanation

ArrayList uses **dynamic array**, LinkedList uses **doubly linked nodes**.

So performance differs.

■ Use LinkedList when:

✔ 1. Frequent insertions/deletions in the middle

LinkedList handles this in **O(1)** (just change pointers).

ArrayList needs **shifting elements → O(n)**.

✔ 2. Implementing queues or stacks

`addFirst()`, `removeFirst()` are fast.

✔ 3. You need consistent performance for sequential access

LinkedList is good for iteration.

■ When NOT to use LinkedList

- When random access is needed

  `list.get(index)` is **O(n)** in LinkedList

(ArrayList is **O(1)**)

■ Short summary

"Use LinkedList for frequent insert/delete operations, especially in the middle of the list. Use ArrayList when fast random access is needed."

6. Exception Handling

✅ 1. Difference between checked vs unchecked exceptions

■ Explanation (simple, conceptual)

- **Checked exceptions** are checked at **compile-time**. The compiler forces you to either handle them with `try-catch` or declare them with `throws`. They represent **recoverable** conditions (e.g., I/O failure).
- **Unchecked exceptions** (runtime exceptions & errors) are checked at **runtime**. They are subclasses of `RuntimeException` (or `Error`) and usually indicate **programming bugs** or unrecoverable conditions (e.g., `NullPointerException`, `ArrayIndexOutOfBoundsException`).

■ When / Why is it used

- Use **checked exceptions** for conditions you expect callers can reasonably handle (file-not-found, network timeouts). They force handling and make error paths explicit.
- Use **unchecked exceptions** for programming mistakes or unexpected states that cannot be sensibly handled at every call site (invalid argument, null pointer). Overusing checked exceptions leads to bulky APIs.

■ Example / Code snippet

```
// Checked exception
public void readFile(String path) throws IOException {
    FileReader fr = new FileReader(path); // FileNotFoundException is checked
}

// Unchecked exception
public int divide(int a, int b) {
```

```
    return a / b; // may throw ArithmeticException at runtime if b==0
}
```

■ Short summary line to speak in interview

"Checked exceptions are compile-time and force handling (for recoverable conditions); unchecked exceptions (RuntimeException/Error) are runtime and usually signal programming errors."

✅ 2. What is the purpose of finally block?

■ Explanation (simple, conceptual)

A `finally` block contains code that **always** runs after a `try` / `catch` block—no matter whether an exception occurred or was caught—so it's ideal for cleanup: closing files, releasing DB connections, releasing locks, etc.

■ When / Why is it used

Use `finally` for resource cleanup that must occur irrespective of success/failure (e.g., closing streams, returning pooled connections). Since Java 7, prefer **try-with-resources** for AutoCloseable resources—but `finally` is still used when explicit cleanup or additional post-processing is needed.

■ Example / Code snippet

```
InputStream in = null;
try {
    in = new FileInputStream("data.csv");
    // process stream
} catch (IOException e) {
    // handle I/O error
} finally {
    if (in != null) {
        try { in.close(); } catch (IOException ignore) {}
    }
}
```

Or with try-with-resources (preferred):

```
try (InputStream in = new FileInputStream("data.csv")) {
    // process stream
} // auto-closed
```

■ Short summary line to speak in interview

"`finally` is used for cleanup that must run regardless of exceptions; for AutoCloseable resources prefer try-with-resources but use `finally` for other forced cleanup."

✅ 3. Can finally block be skipped?

■ Explanation (simple, conceptual)

Under normal program execution, `finally` executes **always**, even if the `try` or `catch` has a `return` or throws another exception. However, there are **rare scenarios** where `finally` may not execute.

■ When/Why it can be skipped (exceptions to the rule)

`finally` may not run if:

- The JVM **crashes** (native crash, `SIGKILL`, OutOfMemory in a way that halts JVM).
- `System.exit(int)` is invoked (JVM termination) — note: shutdown hooks may still run, but `finally` in the currently running thread is not guaranteed.
- The thread is forcibly stopped or killed (deprecated `Thread.stop()` or external OS kill).
- The process is killed by the OS (power loss).

■ Example / Demonstration

```
try {
    System.exit(0);
} finally {
    System.out.println("This may not be printed"); // likely won't run
}
```

But:

```
try {
    return 1;
} finally {
    System.out.println("Finally executes even after return"); // this WILL run
}
```

■ Short summary line to speak in interview

"`finally` runs in normal flows (even after return/exception), but it can be skipped if the JVM is terminated (System.exit, native crash, or OS kill)."

✅ 4. How do you create custom exceptions?

■ Explanation (simple, conceptual)

Create a custom exception by **extending** `Exception` **(checked)** or `RuntimeException` **(unchecked)**. Provide constructors and, if needed, extra fields (error codes, metadata). Use custom exceptions to express domain-specific errors clearly.

■ When/Why is it used

- To represent **business or domain errors** (e.g., `InsufficientFundsException`, `InvalidOtpException`).
- To provide **clearer semantics** and error handling policies across layers.
- To carry structured data (error codes, remediation steps) to the API layer.

■ Example / Code snippet

```
// Checked custom exception
public class InsufficientBalanceException extends Exception {
    private final String accountId;
    public InsufficientBalanceException(String accountId, String msg) {
        super(msg);
        this.accountId = accountId;
    }
    public String getAccountId() { return accountId; }
}

// Unchecked custom exception
public class InvalidRequestException extends RuntimeException {
    public InvalidRequestException(String msg) { super(msg); }
}
```

Usage:

```
if (balance < amount) {
    throw new InsufficientBalanceException(accountId, "Balance too low");
}
```

■ Short summary line to speak in interview

“Create domain-specific exceptions by extending `Exception` (checked) or `RuntimeException` (unchecked); use them to make error handling expressive and consistent.”

✅ 5. In your project, where did you handle exceptions? (practical / project-based answer)

■ Explanation (simple, conceptual)

In production REST microservices (as in your Expense Tracker and Fraud Services), exception handling is layered:

- **At API layer** — translate exceptions to HTTP responses (status + body).
- **At service layer** — throw domain exceptions for business failures.
- **At persistence / integration layer** — catch and wrap lower-level exceptions to hide implementation details.
- **Centralized logging & monitoring** — log full stack traces, metrics, and send alerts.

■ When/Why (how it applied in your projects)

- **Expense Tracker** — used `@ControllerAdvice` + `@ExceptionHandler` to convert `InsufficientBalanceException` → `HTTP 400` with structured JSON `{code, message, details}`; always returned user-friendly messages while logging the original stacktrace. Also used try-with-resources for DB cursors and closed streams in finally for file exports.
- **Fraud Services (middleware)** — domain exceptions like `TransactionValidationException` were thrown for invalid payloads (checked or unchecked as appropriate). For external system timeouts we implemented retry + fallback and caught `IOException` or client library exceptions in integration adapters and rethrew a `DownstreamServiceException`. Multithreaded workers caught exceptions per-task to avoid killing the worker thread and recorded failure metrics (Prometheus) + moved problematic events to a dead-letter queue (Kafka) for later replay.

■ Example snippets (Spring Boot style)

Global handler

```
@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(InsufficientBalanceException.class)
    public ResponseEntity<ErrorDto> handleInsufficient(InsufficientBalanceException
ex) {
        ErrorDto body = new ErrorDto("ERR_1001", ex.getMessage());
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(body);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorDto> handleGeneric(Exception ex) {
```

```
        // log ex
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                        .body(new ErrorDto("ERR_500","Internal error"));
    }
}
```

Service layer

```
public void processTransaction(Transaction tx) {
    try {
      // validation and DB operations
    } catch (SQLTransientException e) {
      // retry logic
      throw new DownstreamServiceException("DB temporary failure", e);
    }
}
```

Multithreaded worker

```
executor.submit(() -> {
    try {
        process(tx);
    } catch (Exception e) {
        log.error("Task failed", e);
        // push to dead-letter queue
    }
});
```

■ Short summary line to speak in interview

“In my projects I used layered exception handling: domain exceptions at service layer, `@ControllerAdvice` to map exceptions to HTTP responses, retries/fallbacks for downstream calls, and per-task try/catch in multithreaded workers with logging and dead-lettering for failed messages.”

---

7. Multithreading

✅ 1. Difference between Process vs Thread

■ Explanation (simple, conceptual)

A **process** is an independent program with its own memory, resources, and execution environment.

A **thread** is a lightweight sub-unit of a process that shares the same memory and resources with other threads.

■ When/Why this matters

- Multiple processes → full isolation, safer but heavier.
- Multiple threads → faster communication, lower overhead, ideal for parallel tasks inside the same application.

■ Key differences

Feature	Process	Thread
Memory	Independent memory	Shared memory inside process
Context switching	Slow, heavy	Fast, lightweight
Communication	Inter-process communication (IPC)	Direct memory access
Crash impact	Only that process	One thread crash may affect entire process
Resource	Heavy	Light

■ Example

Chrome browser → multiple processes

Your Java program → multiple worker threads inside same JVM

■ Short summary

"A process is a heavyweight independent program; a thread is a lightweight unit inside a process sharing the same memory."

✅ 2. What is the difference between Runnable and Thread class?

■ Explanation

Both are ways to create threads, but they serve different purposes.

Runnable interface

- Contains only one method: `run()`
- You pass it to a Thread object
- Allows **multiple threads to share the same task**
- Preferred for real applications

Thread class

- You extend `Thread` and override `run()`
- You cannot extend any other class (because Java doesn't support multiple inheritance)

■ When/Why used

Use **Runnable** when:

- You want to separate task from the thread
- You want better design (implements interface)
- Your class must extend another class

Use **Thread** when:

- You want to override built-in Thread methods (rare)

■ Example

```
// Runnable
class Task implements Runnable {
    public void run() { System.out.println("Running task"); }
}
new Thread(new Task()).start();

// Thread class
class TaskThread extends Thread {
    public void run() { System.out.println("Running task"); }
}
new TaskThread().start();
```

■ Short summary

"Runnable is preferred because it separates task from thread and allows multiple inheritance; extending Thread is rarely needed."

✅ 3. What is synchronization?

■ Explanation

Synchronization ensures that only **one thread at a time** can access a shared resource.

It prevents **race conditions**, **data inconsistency**, and **corrupted state**.

Java provides:

- `synchronized` keyword
- Locks (`ReentrantLock`)
- Synchronization blocks
- Monitor-based locking

■ When/Why used

Use synchronization when:

- Multiple threads update shared variables
- Critical sections must be protected
- Thread safety is required

Avoid unnecessary synchronization because it:

- Reduces performance
- Causes blocking

■ Example

```
public synchronized void deposit(int amount) {
    balance += amount;
}
```

Or block-level:

```
synchronized(this) {
    count++;
}
```

■ Short summary

"Synchronization ensures one thread uses a shared resource at a time to prevent race conditions."

✅ 4. What is deadlock? How to prevent it?

■ Explanation

Deadlock occurs when **two or more threads** are waiting on each other forever, each holding a lock that the other needs.

Example situation

Thread A holds Lock 1, waiting for Lock 2

Thread B holds Lock 2, waiting for Lock 1 → both stuck forever

---

■ When/Why it occurs

- Nested synchronized blocks with different lock orders
- Multiple resources locked inconsistently
- Long-running locks
- Poor design of concurrency

■ How to prevent deadlock

1. **Use consistent lock ordering**
   - Always acquire locks in the same sequence
2. **Use tryLock() with timeout** (ReentrantLock)
3. **Avoid holding multiple locks if possible**
4. **Prefer immutable objects** → no need for locks
5. **Use concurrency packages** instead of low-level locking
6. **Reduce synchronized scope**

■ Example of deadlock

```
synchronized(lock1) {
   synchronized(lock2) { }
}

synchronized(lock2) {
   synchronized(lock1) { }
}
```

■ Short summary

“Deadlock happens when threads wait forever on each other’s locks. Prevent it via lock ordering, tryLock(), or reducing nested synchronized blocks.”

✅ 5. What is volatile keyword?

■ Explanation

`volatile` ensures:

1. **Visibility** → updates to a variable by one thread become immediately visible to others.
2. **Ordering** → prevents instruction reordering issues.

Without volatile:

- A thread may read **cached/stale value** instead of latest value.

Volatile does **NOT** provide atomicity (i.e., ++ is not safe even if volatile).

■ When/Why used

Use volatile when:

- One thread writes, another thread reads
- You have a simple shared flag (e.g., stop thread)
- No compound operations (like ++, x = x + 1)

■ Example

```
volatile boolean running = true;

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

Another thread:

```
running = false; // immediately visible
```

■ Short summary

“volatile guarantees visibility and ordering of reads/writes across threads but not atomicity.”

✅ 6. What is ThreadPool? Why needed?

■ Explanation

A ThreadPool is a pool of reusable worker threads managed by the JVM.

Instead of creating a new thread for every task, ThreadPool **reuses existing threads**.

This avoids:

- High cost of thread creation/destruction
- Unbounded thread creation (which can crash JVM)
- Poor resource management

Java provides via `Executors`:

- FixedThreadPool
- CachedThreadPool
- ScheduledThreadPool
- WorkStealingPool

■ When/Why used

Use a ThreadPool when:

- Many short tasks need fast execution
- Application requires scalability
- You want controlled concurrency
- Avoid memory overhead & CPU overload
- You want clean management of asynchronous tasks

In your **Fraud Detection Service**, ThreadPool would be ideal for:

- Parallel fraud rule checks
- Async operations
- Processing high-volume transactions

■ Example

```
ExecutorService pool = Executors.newFixedThreadPool(10);

pool.submit(() -> {
    System.out.println("Task executed by: " + Thread.currentThread().getName());
});
```

■ Short summary

“A ThreadPool reuses worker threads to execute tasks efficiently. It's essential for scalable, high-performance, multi-threaded applications.”

8. Java 8 Features

✅ 1. What are functional interfaces?

■ Explanation (simple, conceptual)

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

It can have:

- any number of default methods
- any number of static methods

A functional interface is used as the **target type for lambda expressions**.

Common built-in functional interfaces:

- **Runnable** → `run()`
- **Callable** → `call()`
- **Comparator** → `compare()`
- **Supplier, Consumer, Predicate, Function**

■ When/Why used

- For writing cleaner, more concise code
- For passing behavior as a parameter
- For functional programming (map, filter, reduce)
- For asynchronous tasks (ExecutorService)

Functional interfaces enabled Java to support **lambda expressions** and Stream API.

■ Example

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

Usage:

```
Calculator c = (a, b) -> a + b;
```

■ Short summary

"A functional interface contains exactly one abstract method and is the foundation for lambda expressions and Stream operations."

✅ 2. What is a lambda expression?

■ Explanation

A lambda expression is a **short, readable way to represent a method as an expression**.

It removes the boilerplate code of anonymous classes.

Syntax:

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

■ When/Why used

- To simplify functional interface implementation
- To make code concise
- For Streams (`map`, `filter`, `sorted`)
- For async tasks using ExecutorService
- To pass behavior as a parameter (strategy patterns)

■ Example

Without lambda:

```
Runnable r = new Runnable() {
    public void run() {
```

```
        System.out.println("Hello");
    }
};
```

With lambda:

```
Runnable r = () → System.out.println("Hello");
```

■ Short summary

“A lambda expression is a compact way to pass behavior as a value, making Java functional and concise.”

✅ 3. Difference between map() and flatMap()

■ Explanation

map()

- Transforms **each element** of a stream
- Output is **one-to-one mapping**
- Result is another Stream

Example: convert Strings to uppercase.

---

flatMap()

- Transforms and **flattens nested structures**
- Output is **one-to-many mapping**
- Used when each input produces multiple outputs (e.g., list of lists)

■ When/Why used

Use Case	map()	flatMap()
Transform values	✓	✓
Convert list → list	✓	✓
Handle nested lists/streams	✖	✓ flattening
Output becomes Stream<Stream>	✓ map produces	✖ flatMap flattens

---

■ Example

```
List<String> words = List.of("Java", "Spring");

// map example
words.stream()
    .map(String::toUpperCase)
    .forEach(System.out::println);

// flatMap example
List<List<Integer>> nums = List.of(List.of(1,2), List.of(3,4));
nums.stream()
   .flatMap(list → list.stream())
   .forEach(System.out::println);
```

■ Short summary

“map transforms each element; flatMap transforms **and flattens** nested structures.”

✅ 4. What is Optional?

■ Explanation

`Optional` is a container object that may or may not contain a value.

It helps prevent **NullPointerException** and makes null-handling explicit.

■ When/Why used

- When a method may return null
- To avoid null checks everywhere
- To explicitly represent “value may be absent”
- To apply functional operations like `map`, `filter`

It improves code safety and readability.

■ Example

Returning Optional:

```
Optional<User> findById(int id) {
   return Optional.ofNullable(userMap.get(id));
}
```

Usage:

```
userService.findById(10)
        .ifPresent(user → System.out.println(user.getName()));
```

Providing default:

```
String name = optionalUser.map(User::getName)
                    .orElse("Unknown");
```

■ Short summary

“Optional is a wrapper that prevents NullPointerExceptions by representing optional values explicitly.”

✅ 5. What is Stream API? Give a real example from your project.

■ Explanation

Stream API is a **functional programming tool** in Java that processes collections in a **declarative**, **efficient**, and **pipeline-based** manner.

Streams allow operations like:

- map
- filter
- sort
- reduce
- collect
- flatMap

It supports lazy evaluation, parallel execution, and cleaner code.

■ When/Why used

Use Stream API when:

- You want clean, functional-style operations
- You need to filter, map, reduce data
- You want pipelines instead of loops

- You want parallel processing (`parallelStream()`)

■ Real example from your Expense Tracker project

Scenario:

Calculate total monthly expenses for a user and filter only expenses above a threshold, then convert them to DTOs.

```
double total = expenses.stream()
    .filter(e → e.getAmount() > 100)        // filter
    .map(Expense::getAmount)                 // convert to just amounts
    .reduce(0.0, Double::sum);              // sum
```

Creating DTO list:

```
List<ExpenseDTO> dtos =
    expenses.stream()
          .map(exp → new ExpenseDTO(exp.getId(), exp.getAmount()))
          .collect(Collectors.toList());
```

■ Real example from your Fraud Detection service

E.g., **parallel fraud scoring**:

```
List<Integer> scores = transactions.parallelStream()
    .map(tx → fraudEngine.calculateScore(tx))
    .collect(Collectors.toList());
```

This improved performance for high transaction volumes.

■ Short summary

“Stream API provides functional-style operations for filtering, mapping, and reducing data. I used it in my Expense Tracker to calculate totals, filter expenses, and convert entities to DTOs, and in Fraud Service for parallel scoring.”

---

9. Memory Management

✅ 1. Explain heap vs stack memory.

■ Explanation (simple, conceptual)

Stack Memory

- Stores **method frames**, **local variables**, **references**, and **function call data**.
- Follows **LIFO** (Last-In-First-Out).
- Memory is **automatically** created and removed when method enters/exits.
- **Thread-specific** → each thread has its own stack.

Heap Memory

- Stores **objects**, **arrays**, **class instances**, **String Pool (partially)**.
- Shared by **all threads**.
- Managed by **Garbage Collector (GC)**.
- Larger and slower than stack.

■ When/Why used

Stack	Heap
For temporary variables inside methods	For dynamically created objects
Fast access, limited size	Larger memory, slower access
Thread-safe automatically	Needs synchronization if shared

■ Example

```
public void test() {
    int x = 10;         // stored on Stack
    User u = new User();// 'u' reference on stack; User object in Heap
}
```

■ Short summary

"Stack stores method-local data and references; Heap stores objects shared by the entire application and is managed by Garbage Collector."

✅ 2. What is garbage collection?

■ Explanation

Garbage Collection (GC) is the JVM's automatic memory management process that **removes unused objects from the heap**.

An object becomes eligible for GC when **no active reference** points to it.

GC relieves developers from manually freeing memory, preventing many memory-related bugs.

■ When/Why used

- Prevents memory overflow
- Optimizes heap usage
- Removes unreachable objects
- Improves application stability

■ Example

```
User u = new User();
u = null;   // eligible for GC
```

OR:

Local objects become unreachable after method exits.

■ Short summary

"Garbage Collection automatically frees heap memory by removing unreachable objects."

✅ 3. Explain minor GC vs major GC.

■ Explanation

Java divides heap into generations:

Young Generation

- Eden + Survivor spaces
- Where new objects are created
- When it fills → **Minor GC**

Old (Tenured) Generation

- Older, long-lived objects
- When full → **Major GC**

Minor GC

- Cleans young generation
- Very fast
- Happens frequently
- Moves surviving objects → Survivor spaces → Old gen

Major GC (Full GC)

- Cleans old generation
- Slow, expensive
- May stop-the-world (application pause)

⬛ Why important

Minor GC affects performance slightly.

Major GC can cause **lags, pauses**, and must be avoided in high-performance apps.

⬛ Short summary

"Minor GC cleans new objects; Major GC cleans old-generation objects and is slower."

✅ 4. Which GC algorithms do you know (G1, Serial, Parallel)?

⬛ Explanation

Java provides multiple Garbage Collectors:

---

1. Serial GC

- Single-threaded
- Best for small applications
- Pauses entire application
- Uses **Mark-Sweep-Compact** algorithm

Use when: Low memory, single-threaded system.

---

2. Parallel GC

- Multi-threaded GC

- Focused on high throughput
- Still uses stop-the-world events
- Default in older Java versions

Use when: CPU-rich applications where short pauses are acceptable.

---

3. G1 (Garbage First) GC → Modern default

- Splits heap into **regions**
- Performs **parallel + incremental** GC
- Predictable pause times
- Performs **concurrent marking**

Use when: Large heap, low-latency applications (e.g., enterprise servers).

---

4. ZGC (if asked)

- Ultra-low pause time (< 1ms)
- Performs all GC work concurrently
- Ideal for extremely large heaps (multi-GB)

■ Short summary

"Serial → single thread; Parallel → multi-threaded throughput GC; G1 → modern low-pause regional GC; ZGC → ultra-low pause concurrent GC."

✅ 5. What is memory leak in Java?

■ Explanation

A memory leak occurs when objects are **not in use anymore** but are **still referenced**, so GC cannot remove them.

This leads to:

- Increasing heap usage
- OutOfMemoryError
- Slow application over time

■ Common causes

1. **Static references** holding objects forever

2. **Unclosed resources** (ResultSet, Streams, Sockets)
3. **Improper caching** (e.g., HashMap growing forever)
4. **Listeners, observers not removed**
5. **ThreadLocal misuse**
6. Long-lived threads holding references

■ Project-level example (Fraud Detection Service)

If fraud rule results were cached incorrectly in a HashMap without removal:

```
cache.put(txId, result);  // No eviction policy → memory leak
```

Or ExecutorService thread pool not shutting down:

```
ExecutorService pool = Executors.newFixedThreadPool(10);
// shutting down missing → threads hold memory forever
```

■ Short summary

"A memory leak happens when unused objects remain referenced, preventing GC and gradually consuming heap."

---

10. File Handling & I/O

✅ 1. Difference between FileReader and BufferedReader

■ Explanation (simple, conceptual)

FileReader

- A **character stream** that reads characters directly from a file.
- Reads **one character at a time**, causing frequent disk access.

BufferedReader

- Wraps another Reader (like FileReader).
- Reads a **large chunk of characters into an internal buffer**.
- Provides convenient methods like **readLine()**.

■ When/Why used

Feature	FileReader	BufferedReader
Efficiency	Slower	Faster due to buffering
Reading	Character-by-character	Line-by-line & chunk reading
Best use case	Small files	Large files, logs, text processing

BufferedReader is almost always preferred for reading text files.

■ Example / Code snippet

```
FileReader fr = new FileReader("data.txt");
BufferedReader br = new BufferedReader(fr);

String line;
while((line = br.readLine()) != null) {
    System.out.println(line);
}
```

■ Short summary

“FileReader reads characters directly, while BufferedReader adds a buffer for faster, line-based reading.”

✅ 2. Why is BufferedReader faster?

■ Explanation

BufferedReader reads data from the file in **large chunks** (buffer size ~8KB or more), and stores it internally.

Then when your program requests data:

- It serves it from memory buffer
- Rather than accessing disk repeatedly

Disk access is **very slow**, memory access is **very fast** → huge performance improvement.

■ When/Why used

Use BufferedReader when:

- Reading large text files

- Reading line-by-line
- Performance matters
- Minimizing I/O operations
- Parsing logs, CSVs, JSON, config files

■ Example

```
Without Buffer:
Read → Disk → App → Disk → App  (slow)

With Buffer:
Disk → Buffer → App → App → App (fast)
```

BufferedReader reduces disk interaction from **thousands** of calls to **just a few**.

■ Short summary

"BufferedReader is faster because it minimizes disk I/O by reading big chunks at once and serving data from memory."

✅ 3. How does InputStream differ from Reader?

■ Explanation

Java provides two major types of streams:

InputStream (byte-based stream)

- Reads **raw bytes (8-bit)**
- Used for **binary data**: images, videos, PDF, audio

Reader (character-based stream)

- Reads **Unicode characters (16-bit)**
- Used for **text files**

■ When/Why used

Feature	InputStream	Reader
Data type	Bytes	Characters

Feature	InputStream	Reader
Use case	Binary data	Text data
Encoding	Needs manual handling	Handles encoding automatically
Examples	FileInputStream, BufferedInputStream	FileReader, BufferedReader

■ Example

InputStream (binary data example)

```
InputStream in = new FileInputStream("image.png");
int data = in.read(); // reads bytes
```

Reader (text example)

```
Reader r = new FileReader("notes.txt");
int ch = r.read(); // reads characters
```

■ Short summary

“InputStream reads bytes (binary data), while Reader reads characters (text). Choose InputStream for images/files and Reader for text processing.”

⭐ SCENARIO-BASED QUESTIONS (Based on Projects)

✅ 1. You created REST APIs — how did you handle exceptions at API level?

■ Explanation (simple, conceptual)

I handled exceptions using a **layered approach**: throw meaningful domain exceptions from service/adapters, wrap/translate low-level errors, and use a centralized controller-level handler to convert exceptions into consistent HTTP responses (status code + JSON body). This separates error handling concerns and ensures clients get stable, user-friendly messages while logs contain full diagnostics.

■ When / Why is it used

- To maintain **consistent API contracts** (error codes, messages).

- To **avoid leaking implementation details** (stack traces) to clients.
- To enable **monitoring / alerting** on specific error types.
- To support retries, circuit-breakers or translating transient errors to 5xx vs business errors to 4xx.

■ Example / Code snippet (Spring Boot style from Expense Tracker)

Custom exceptions

```
public class ExpenseNotFoundException extends RuntimeException {
   public ExpenseNotFoundException(String id) { super("Expense not found: " + id);
}
}
```

Service throws domain error

```
public ExpenseDto getExpense(String id) {
   return repo.findById(id).orElseThrow(() → new ExpenseNotFoundException(id));
}
```

Global handler

```
@RestControllerAdvice
public class GlobalExceptionHandler {
   @ExceptionHandler(ExpenseNotFoundException.class)
   public ResponseEntity<ErrorDto> handleNotFound(ExpenseNotFoundException e
x) {
      ErrorDto body = new ErrorDto("EXP_404", ex.getMessage());
      return ResponseEntity.status(HttpStatus.NOT_FOUND).body(body);
   }

   @ExceptionHandler(Exception.class)
   public ResponseEntity<ErrorDto> handleGeneric(Exception ex) {
      log.error("Unexpected error", ex);          // full stacktrace in logs
      return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                     .body(new ErrorDto("ERR_500","Internal server error"));
   }
}
```

Notes: used `@ControllerAdvice` for mapping, retained original exception as `cause` for logs, and returned sanitized messages to clients. Transient DB errors were wrapped in `DownstreamServiceException` to trigger retry in caller.

■ Short summary line to speak in interview

"I used a layered approach: throw domain exceptions in services, and map them to consistent HTTP responses using `@ControllerAdvice`, logging full details internally while returning safe messages to the client."

✅ 2. How did you use Collections in your Expense Tracker project?

■ Explanation (simple, conceptual)

Collections were used for storing, aggregating, and transforming expense data in-memory: `List` for ordered expense entries, `Map` for grouping by user/category/date, `Set` for unique lookups (tag sets), and Streams for aggregation (sum, grouping, filtering). I also used concurrent collections for any cached/shared data that could be accessed across threads.

■ When / Why is it used

- `List` for feeds and pagination.
- `Map<userId, List<Expense>>` for quick user-wise grouping.
- `Set` to deduplicate tags or categories.
- `ConcurrentHashMap` for in-memory cache of recent calculations accessed by multiple threads.
- Streams for concise aggregations and DTO mapping.

■ Example / Code snippet

Grouping and aggregation with Streams

```
Map<String, Double> monthlySpending = expenses.stream()
   .filter(e → e.getUserId().equals(userId))
   .collect(Collectors.groupingBy(
      e → YearMonth.from(e.getDate()).toString(),
      Collectors.summingDouble(Expense::getAmount)
   ));
```

Using Map for fast lookup

```
Map<String, Expense> expenseById = expenses.stream()
   .collect(Collectors.toMap(Expense::getId, Function.identity()));
```

Concurrent cache

```
private final ConcurrentMap<String, BudgetCache> budgetCache = new Concurrent
HashMap<>();
budgetCache.computeIfAbsent(userId, id -> computeBudget(id));
```

■ Short summary line to speak in interview

“I used Lists for ordered data, Maps for fast grouping/lookup, Sets to dedupe, Streams for aggregation, and ConcurrentHashMap for thread-safe in-memory caching.”

✅ 3. In your Fraud Detection service, where did you use multithreading?

■ Explanation (simple, conceptual)

Multithreading was used to **parallelize CPU-bound and I/O-bound tasks**: parallel rule evaluation, async enrichment calls to external services (KYC, transaction history), and parallel scoring of transaction batches to meet throughput SLAs.

■ When / Why is it used

- To process high transaction volumes with low latency.
- To avoid blocking the main request thread while calling external services.
- To scale CPU-bound scoring across cores.

■ Example / Code snippet

ExecutorService for parallel scoring

```
ExecutorService pool = Executors.newFixedThreadPool(Runtime.getRuntime().availa
bleProcessors());
List<Future<Score>> futures = transactions.stream()
    .map(tx -> pool.submit(() -> fraudEngine.calculateScore(tx)))
    .collect(Collectors.toList());

for (Future<Score> f : futures) {
    try { scores.add(f.get()); }
    catch (Exception e) { log.error("per-tx failure", e); moveToDLQ(tx); }
}
```

CompletableFuture for async enrichment

```
CompletableFuture<Profile> profileFut = CompletableFuture.supplyAsync(() → profil
eClient.getProfile(tx.getUserId()), pool);
CompletableFuture<Double> externalScoreFut = CompletableFuture.supplyAsync(()
→ externalService.score(tx), pool);

CompletableFuture.allOf(profileFut, externalScoreFut).thenApply(v → {
   Profile p = profileFut.join();
   double ext = externalScoreFut.join();
   return combineScores(tx, p, ext);
});
```

Notes: each task used try/catch inside the runnable to avoid killing pool threads; failures were logged and offending messages sent to a dead-letter topic (Kafka) for replay.

■ Short summary line to speak in interview

"I used thread pools and `CompletableFuture` to parallelize scoring and async enrichment, handling per-task exceptions and moving failures to a DLQ so the worker threads remain healthy."

✅ 4. How did you optimize SQL queries using Java code?

■ Explanation (simple, conceptual)

Optimization combined **query-level improvements** (indexing, projections, joins) and **Java-level best practices**: using prepared statements / parameterized queries, batching inserts/updates, paging results, avoiding N+1 by using joins/fetch-joins, caching hot data, and using connection pooling.

■ When / Why is it used

- Reduce DB latency and CPU usage.
- Avoid transferring large result sets.
- Prevent N+1 queries that blow up under load.
- Improve throughput for bulk operations.

■ Example / Code snippet & tactics

1. Use projection (only required columns)

```
SELECT id, amount, date FROM expenses WHERE user_id = ? LIMIT ? OFFSET ?
```

2. PreparedStatement & batching (bulk insert)

```
try (PreparedStatement ps = conn.prepareStatement(sql)) {
   for (Expense e : batch) {
      ps.setString(1, e.getId());
      ps.setDouble(2, e.getAmount());
      ps.addBatch();
   }
   ps.executeBatch();
}
```

3. Avoid N+1 with JOIN / fetch-join (JPA)

```
@Query("SELECT e FROM Expense e JOIN FETCH e.category WHERE e.user.id = :ui
d")
List<Expense> findWithCategory(@Param("uid") String userId);
```

4. Use EXPLAIN & add indexes

- Run `EXPLAIN` to find full table scans and add indexes on `user_id`, `date`, or `transaction_id` as needed.

5. Caching frequently-read small tables

- Use in-memory cache (ConcurrentHashMap or Redis) for static reference data (category labels), avoiding repeated DB hits.

6. Connection pool

- Use HikariCP (configured via Spring Boot) to maintain optimal DB connections and reduce connection creation overhead.

■ Short summary line to speak in interview

"I optimized SQL by using projections, prepared-statement batching, fetch-joins to avoid N+1, adding proper indexes after EXPLAIN, using connection pooling and caching hot data to reduce DB load."

✅ 5. How did you ensure your microservices were thread-safe?

■ Explanation (simple, conceptual)

Thread-safety was mainly achieved by **designing services to be stateless**, using **thread-safe data structures** for shared in-memory state, and minimizing synchronized

blocks. In Spring, beans are singletons by default, so I made them stateless or used proper synchronization/Concurrent collections for necessary state.

■ When / Why is it used

- Stateless services avoid shared mutable state, simplifying concurrency.
- Where shared state was required (e.g., local caches, counters), use `ConcurrentHashMap`, `AtomicLong`, or `ReadWriteLock`.
- Use immutable DTOs to avoid accidental mutation across threads.

■ Example / Code snippet

Stateless service (preferred)

```
@Service
public class ExpenseService {
   // No mutable fields → thread-safe
   public ExpenseDto calculateBudget(String userId) { ... }
}
```

Thread-safe cache

```
private final ConcurrentMap<String, Budget> cache = new ConcurrentHashMap<>
();

public Budget getBudget(String userId) {
   return cache.computeIfAbsent(userId, id → loadBudget(id));
}
```

Atomic counters

```
private final AtomicLong requestCounter = new AtomicLong();

public void record() { requestCounter.incrementAndGet(); }
```

Avoiding mutable shared fields — never keep request-scoped state in singleton fields. Use request context or method-local variables.

■ Short summary line to speak in interview

“I ensured thread-safety by keeping services stateless, using concurrent collections/atomics where shared state was needed, and using immutable DTOs and small synchronized scopes when necessary.”

✅ 6. What Java feature helped you design secure authentication (JWT)?

■ Explanation (simple, conceptual)

JWT (JSON Web Tokens) provides stateless, signed tokens containing claims. Java features/libraries that helped: JWT libraries (jjwt, java-jwt) for signing/verification, `javax.crypto` / `java.security` for key handling, and Spring Security filters (OncePerRequestFilter) to validate tokens per request before controller logic.

■ When / Why is it used

- JWT is used for **stateless auth** across microservices (no server-side session).
- Use signed tokens (HMAC or RSA) to ensure integrity and authenticity.
- Inspect claims (roles, userId, ttl) to authorize requests.
- Use refresh tokens and short access token TTL for security.

■ Example / Code snippet

Token creation (HMAC example)

```
String token = Jwts.builder()
    .setSubject(userId)
    .claim("roles", roles)
    .setIssuedAt(new Date())
    .setExpiration(Date.from(Instant.now().plus(15, ChronoUnit.MINUTES)))
    .signWith(Keys.hmacShaKeyFor(secretBytes), SignatureAlgorithm.HS256)
    .compact();
```

Token validation filter (simplified)

```
public class JwtFilter extends OncePerRequestFilter {
    protected void doFilterInternal(HttpServletRequest req, HttpServletResponse res,
FilterChain chain) {
        String auth = req.getHeader("Authorization");
        if (auth != null && auth.startsWith("Bearer ")) {
            try {
                Claims claims = Jwts.parserBuilder().setSigningKey(secret).build().parseCl
aimsJws(token).getBody();
                // set Authentication in SecurityContext based on claims
            } catch (JwtException e) {
                res.setStatus(HttpStatus.UNAUTHORIZED.value());
                return;
```

```
            }
        }
        chain.doFilter(req, res);
    }
}
```

Security best practices used

- Store secrets in vault (AWS Secrets Manager / Azure Key Vault), not in code.
- Use RS256 (asymmetric) when multiple services need to verify tokens but only auth server signs them.
- Short access token TTL + refresh tokens and revoke lists for suspicious tokens.
- Validate `exp`, `nbf`, `iat`, audience and issuer claims.

■ Short summary line to speak in interview

“I used JWTs with secure signing/verification (library + Java crypto), validated tokens in a Spring Security filter, stored secrets in a vault, used short TTLs and refresh tokens to build a secure stateless auth flow.”

⭐ LITTLE TRICKY (Asked in product-based companies)

✅ 1. Can HashMap contain null key?

■ Explanation

Yes, **HashMap allows one null key** and multiple null values.

Internally, HashMap treats the null key as a **special case** and always stores it in bucket **index 0** (because it cannot call `hashCode()` on null).

■ When/Why is it used

- Rarely used in production; allowed mainly for compatibility with older Java versions.
- Good for representing "no key" or default cases.

■ Example

```
Map<String, Integer> map = new HashMap<>();
map.put(null, 10);
```

```
System.out.println(map.get(null)); // 10
```

■ Short interview summary

“Yes, HashMap allows one null key. It's stored in bucket 0 because null has no hashCode().”

✅ 2. How HashMap works when capacity exceeds threshold?

■ Explanation

HashMap has two parameters:

- **capacity** → size of internal array
- **load factor** → default is **0.75**

Threshold = capacity × load factor

Example: capacity 16 → threshold = 12

When size > threshold → HashMap resizes

Resizing involves:

1. Doubling the capacity (e.g., 16 → 32)
2. Rehashing existing entries
3. Redistributing entries into new buckets
4. LinkedLists or Trees are also remapped

■ When/Why is it used

- To maintain performance (O(1) average time)
- Prevent too many collisions in the same bucket
- JVM ensures even distribution across new buckets

■ Example

```
capacity = 16
load factor = 0.75
threshold = 12
When 13th entry is inserted → resize to 32
```

■ Short interview summary

“When size exceeds threshold, HashMap doubles capacity and rehashes entries to maintain performance.”

✅ 3. Why should equals() and hashCode() be overridden together?

■ Explanation

Objects stored as keys in HashMap, HashSet rely on **both**:

- `hashCode()` to find bucket
- `equals()` to check key equality in that bucket

The contract:

1. If two objects are **equal** (equals() returns true), they **must** have the **same hashCode**.
2. If they have different hashCodes, they **can’t** be equal.

If you break the contract:

- Two equal objects may end up in **different buckets**
- Collections like HashMap won’t find them → inconsistent behavior

---

■ When/Why is it used

- To ensure correct behavior in HashMap, HashSet, ConcurrentHashMap
- Avoid duplicate keys
- Support lookup/removal operations correctly

■ Example

```
@Override
public int hashCode() {
    return id;  // stable hash
}

@Override
public boolean equals(Object obj) {
    return this.id == ((User)obj).id;
}
```

■ Short interview summary

“Because HashMap uses hashCode() to locate bucket and equals() to compare keys. If they are inconsistent, lookups break.”

✅ 4. What happens internally during autoboxing?

■ Explanation

Autoboxing = converting primitive → wrapper object automatically.

Example:

```
Integer x = 10; // int → Integer
```

Internally:

1. Java calls static factory methods like `Integer.valueOf(10)`
2. ValueOf uses **wrapper object cache** (range: -128 to 127 for Integer)
3. If value is in cache → reused object
4. Else → new wrapper object is created

Unboxing works similarly:

```
int y = x; // x.intValue()
```

■ When/Why used

- When storing primitives in collections (`List<Integer>`, not `List<int>`)
- When passing primitives to methods expecting objects
- For clean, readable code

■ Example

```
Integer a = 100;  // cached
Integer b = 100;  // same object

Integer c = 200;  // not cached
Integer d = 200;  // different object
```

■ Short interview summary

“Autoboxing calls valueOf(), using cached wrapper objects when possible. Unboxing calls methods like intValue().”

✅ 5. What is the difference between fail-fast and fail-safe iterators?

■ Explanation

Fail-Fast Iterator

- Throws **ConcurrentModificationException**
- Happens when the underlying collection is **modified structurally** while iterating
- Most Java collections are fail-fast:
  - ArrayList
  - HashMap
  - HashSet

Works by maintaining a **modCount** and comparing with expectedModCount.

Fail-Safe Iterator

- Does **not** throw ConcurrentModificationException
- Works on a **copy of the collection**
- Modifications do not affect the iterator
- Used in:
  - ConcurrentHashMap
  - CopyOnWriteArrayList

■ When/Why used

Type	When to use	Why
Fail-Fast	Single-threaded or controlled modification	Detects concurrent changes early
Fail-Safe	Multi-threaded environments	Safe iteration without exceptions

---

■ Example

Fail-Fast (ArrayList)

```
for(Integer i : list) {
    list.add(20); // throws ConcurrentModificationException
}
```

Fail-Safe (ConcurrentHashMap)

```
for(Integer i : map.keySet()) {
    map.put(5, "new"); // NO exception
}
```

■ Short interview summary

“Fail-fast iterators throw ConcurrentModificationException on structural changes; fail-safe iterators work on a clone and allow safe iteration in concurrent environments.”
