**OOPS**

**What is OOPS?**

OOPS (Object-Oriented Programming System) is a methodology that simplifies software development by providing some rules.

Java uses OOPS to achieve **modularity, reuse, and maintainability**.

**OOPS Concepts**

Object-Oriented Programming in Java is based on **4 main principles**:

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

There are also supporting concepts:

* Class & Object
* Constructor
* this & super
* Association, Aggregation, Composition

**1. Class and Object**

**Class**

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

Example:

class Car {

String color;

void drive() {

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

}

}

**Object**

An object is an instance of a class.

Example:

Car car1 = new Car();

car1.color = "Red";

car1.drive();

**2. Encapsulation**

Encapsulation = **binding data + methods together**

* hiding internal details using **private** access modifier.

✔ Helps in data protection   
✔ Achieves data hiding

**Example:**

class BankAccount {

private double balance;

public void deposit(double amount) {

balance += amount;

}

public double getBalance() {

return balance;

}

}

Here:

* balance is hidden from direct access
* Interaction only through public methods  
  → This is **Encapsulation**

**3. Inheritance**

Inheritance = one class acquiring properties of another.

* Parent class → **Super class**
* Child class → **Sub class**

✔ Code reuse   
✔ Supports method overriding

* Single
* Multilevel
* Hierarchical
* Hybrid

**Example:**

class Animal {

void eat() {

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

}

}

class Dog extends Animal {

void bark() {

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

}

}

Usage:

Dog d = new Dog();

d.eat(); // from Animal

d.bark(); // from Dog

**4. Polymorphism**

Polymorphism = **many forms**  
Two types:

**A. Compile-time Polymorphism (Method Overloading) /Static polymorphism**

Multiple methods with **same name but different parameters**.

class MathUtil {

int add(int a, int b) { return a + b; }

int add(int a, int b, int c) { return a + b + c; }

}

**B. Runtime Polymorphism (Method Overriding) /Dynamic polymorphism**

Child class provides its **own implementation** of parent method.

class Animal {

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

}

class Dog extends Animal {

@Override

void sound() { System.out.println("Dog barks"); }

}

Runtime example:

Animal a = new Dog();

a.sound(); // Dog barks

This is **dynamic binding**.

**5. Abstraction**

Abstraction = **show essential part and hide implementation details.**

Two ways:

1. **Abstract class**
2. **Interface**

**A) Abstract Class**

* Can have **abstract + non-abstract** methods
* Cannot be instantiated
* Can have constructors
* Supports partial abstraction

abstract class Shape {

abstract void draw(); // abstract method

void color() {

System.out.println("Shape color");

}

}

class Circle extends Shape {

void draw() { System.out.println("Drawing circle"); }

}

**B) Interface**

* 100% abstraction (before Java 8)
* Only abstract methods (now can have default & static)
* Used for multiple inheritance

interface Vehicle {

void start();

}

class Bike implements Vehicle {

public void start() {

System.out.println("Bike starts");

}

}

**6. Association, Aggregation, Composition**

**Association**

Association represents a **general relationship** between two classes where both objects can **exist independently**.

**Aggregation**

Aggregation is a has-a relationship with weak ownership. Child object **can exist independently**

Example:  
*Department* has students — students can exist without department.

**Composition**

Composition is a **strong has-a relationship** with **strong ownership**.

Child object **cannot exist** without parent.

Example:  
*House* has rooms — rooms cannot exist without house.

| **Feature** | **Association** | **Aggregation** | **Composition** |
| --- | --- | --- | --- |
| Dependency | No dependency | Weak dependency | Strong dependency |
| Ownership | No ownership | Shared ownership | Exclusive ownership |
| Lifecycle | Independent | Independent | Dependent |

**How does memory management differ between Aggregation & Composition?**

**Answer:**

* **Aggregation**  
  Parent does **not** manage the child’s lifecycle.  
  Child can live independently.
* **Composition**  
  Parent **controls** the child’s creation and destruction.

**What is the difference between Abstraction and Encapsulation?**

| **Abstraction** | **Encapsulation** |
| --- | --- |
| Hides *implementation details* | Hides *data* |
| Achieved using **abstract class/interface** | Achieved using **private fields** |

**Can Java support multiple inheritance? Why not?**

**Answer:**  
Java **does not support multiple inheritance** using classes to avoid **diamond problem**.  
But it **supports multiple inheritance using interfaces**.

**Difference between Abstract Class and Interface?**

| **Abstract Class** | **Interface** |
| --- | --- |
| Partial abstraction | Full abstraction |
| Can have constructor | No constructor |
| Single inheritance | Multiple inheritance |

**What is Composition vs Aggregation?**

| **Aggregation** | **Composition** |
| --- | --- |
| Weak association | Strong association |
| Child can exist independently | Child cannot exist |
| Example: Department → Student | House → Rooms |

**What is a final class?**

A class that **cannot be extended**.

**What is a static method?**

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

**Why Java is not fully object-oriented?**

Because it uses **primitive data types** (int, float, etc.).

**What is the difference between class and object?**

| **Class** | **Object** |
| --- | --- |
| Blueprint | Instance |
| No memory allocated | Memory allocated |
| Logical | Physical |

**How many ways we have to create an object?**

**1. Using new keyword (Most common)**

MyClass obj = new MyClass();

**2. Using Class.forName() (Reflection)**

Used when class name is known at runtime.

MyClass obj = (MyClass) Class.forName("com.example.MyClass").newInstance();

⚠ newInstance() is deprecated. Use:

MyClass obj = MyClass.class.getDeclaredConstructor().newInstance();

**3. Using clone() method**

Cloning creates a new object from an existing one.

MyClass obj1 = new MyClass();

MyClass obj2 = (MyClass) obj1.clone();

Class must implement Cloneable.

**4. Using Deserialization (ObjectInputStream)**

Creates an object from a byte stream (even if constructor is private!).

ObjectInputStream in = new ObjectInputStream(new FileInputStream("data.obj"));

MyClass obj = (MyClass) in.readObject();

**5. Using Factory methods / newInstance() of constructors**

You can get constructors using Reflection and call newInstance():

Constructor<MyClass> constructor = MyClass.class.getConstructor();

MyClass obj = constructor.newInstance();

Factory method example:

Integer i = Integer.valueOf(10); // returns new or cached object

**Exceptions**

**What is an Exception in Java?**

An exception is an **unexpected event** that disrupts the normal execution of a program.

int x = 10 / 0; // ArithmeticException

**What is the difference between Exception and Error?**

| **Exception** | **Error** |
| --- | --- |
| Recoverable | Not recoverable |
| Application-level issue | System-level issue |
| Example: IOException | Example: OutOfMemoryError |

// Error example

public class Test {

public static void main(String[] args) {

recurse(); // causes StackOverflowError

}

static void recurse() { recurse(); }

}

**Checked vs Unchecked Exceptions**

✔ Checked → Compile-time  
✔ Unchecked → Runtime exceptions

// Checked

FileReader fr = new FileReader("abc.txt"); // Compile error

// Unchecked

String s = null;

s.length(); // NullPointerException

**Explain Exception Hierarchy.**

Throwable

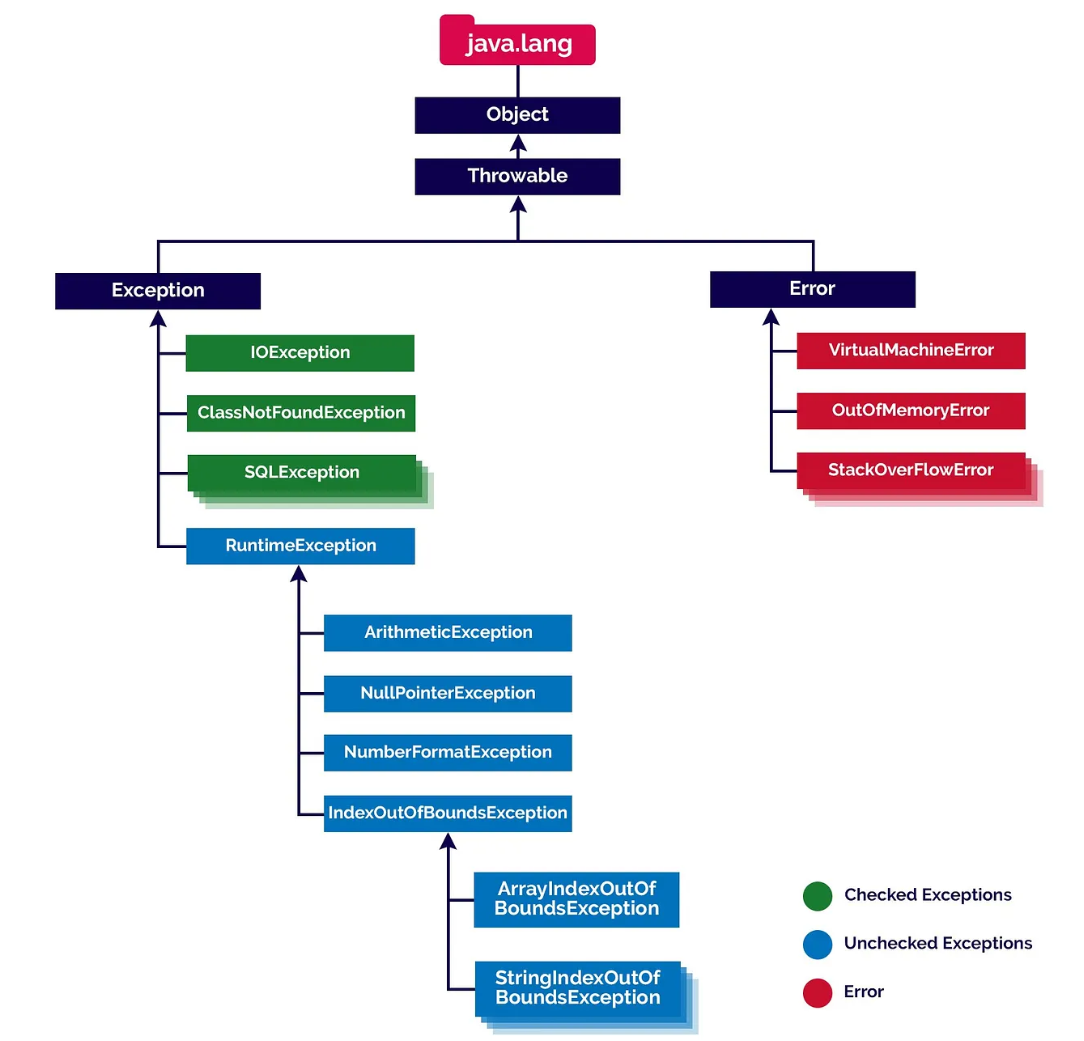
├── Error

└── Exception

├── RuntimeException

├── IOException

└── SQLException



**Difference between throw and throws**

| **throw** | **throws** |
| --- | --- |
| Used to throw exception manually | Used to declare exception |
| Object of exception is created | Method signature |
| One at a time | Multiple allowed |

void check(int age) {

if (age < 18)

throw new IllegalArgumentException("Not eligible");

}

void readFile() throws IOException {

FileReader fr = new FileReader("file.txt");

}

**try–catch–finallym**

Finally always executes.

try {

int x = 10 / 0;

} catch (ArithmeticException e) {

System.out.println("Error");

} finally {

System.out.println("Always executed");

}

**Can a try block exist without catch?**

YES, if there is a finally block.

try {

System.out.println("Inside try");

} finally {

System.out.println("Always");

}

**Multi-catch block (Java 7+)**

One catch for multiple exceptions.

try {

int x = Integer.parseInt("abc");

} catch (NumberFormatException | NullPointerException e) {

e.printStackTrace();

}

**try-with-resources**

Automatically closes resources that implement AutoCloseable.

try (FileReader fr = new FileReader("file.txt")) {

// file auto-closed

} catch (IOException e) {

e.printStackTrace();

}

**Example: multiple resources + suppressed exceptions demo**

class R implements AutoCloseable {

private final String name;

R(String name) { this.name = name; }

@Override

public void close() throws Exception {

System.out.println("closing " + name);

if ("B".equals(name)) throw new Exception("close B failed");

}

}

public static void main(String[] args) {

try (R a = new R("A"); R b = new R("B")) {

throw new Exception("work failed");

} catch (Exception e) {

System.out.println("Primary: " + e.getMessage());

for (Throwable s : e.getSuppressed())

System.out.println("Suppressed: " + s.getMessage());

}

}

Output:

closing B

closing A

Primary: work failed

Suppressed: close B failed

Note: B closed first (because it was declared second), and its close exception became suppressed.

**What is a Custom Exception?**

Used to create business-specific errors.

class InvalidAgeException extends Exception {

InvalidAgeException(String message) {

super(message);

}

}

void validate(int age) throws InvalidAgeException {

if (age < 18) throw new InvalidAgeException("Age < 18");

}

**What is Exception Chaining?**

Wrapping an exception inside another.

try {

int x = Integer.parseInt("abc");

} catch (NumberFormatException e) {

throw new RuntimeException("Number conversion failed", e);

}

**What happens if exception is not handled?**

JVM prints stack trace → program stops.

**Can a constructor throw an exception?**

YES.

class Demo {

Demo() throws IOException {

throw new IOException("Constructor error");

}

}

**What happens if finally returns a value?**

Finally overrides try/catch return value → BAD practice.

int test() {

try {

return 1;

} finally {

return 2;

}

}

Result → 2

**What are common Runtime Exceptions?**

* NullPointerException
* ArithmeticException
* IndexOutOfBoundsException
* ClassCastException
* IllegalArgumentException

**What is finalize()?**

Called before garbage collection (deprecated).

@Override

protected void finalize() {

System.out.println("Finalize called");

}

**What are Suppressed Exceptions?**

A **suppressed exception** is an exception that occurs **during the cleanup phase**, usually **inside a finally block** or inside **AutoCloseable.close()** when another exception is **already thrown** in the try block.

Exceptions suppressed during try-with-resources.

try (MyResource r = new MyResource()) {

} catch (Exception e) {

System.out.println(Arrays.toString(e.getSuppressed()));

}

**Why catching Exception is bad?**

Because it hides specific errors.

catch (Exception e) {}

**Steps JVM follows when exception occurs**

**Theory**

1. Creates exception object
2. Finds matching catch
3. If not found → propagates up
4. Program terminates

**Rethrowing an Exception**

try {

risky();

} catch (IOException e) {

throw e;

}

**final, finally, finalize**

**Theory**

| **Keyword** | **Meaning** |
| --- | --- |
| final | Constant / no override |
| finally | Cleanup block |
| finalize() | GC hook |

**Exception in static block**

static {

int x = 10/0; // ExceptionInInitializerError

}

**Exception in Thread**

Thread t = new Thread(() -> {

throw new RuntimeException("Boom!");

});

t.setUncaughtExceptionHandler((th, ex) -> {

System.out.println("Caught: " + ex);

});

t.start();

**Real-world Exception Handling Example**

public String loadConfig() {

try {

return Files.readString(Path.of("config.json"));

} catch (NoSuchFileException e) {

return "{}"; // default config

} catch (IOException e) {

throw new RuntimeException("Unable to read config", e);

}

}

**Global Exception Handling in Spring Boot**

**Code**

@RestControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<?> handleNotFound(ResourceNotFoundException e) {

return ResponseEntity.status(404).body(e.getMessage());

}

}

**Nested try-catch**

**Code**

try {

try {

int x = 10 / 0;

} catch (ArithmeticException e) {

System.out.println("Inner catch");

}

} catch (Exception e) {

System.out.println("Outer catch");

}

**Can we catch Throwable?**

Yes, but BAD. It catches Errors too.

catch (Throwable t) { }

**Custom Runtime Exception**

class InvalidInputException extends RuntimeException {

InvalidInputException(String msg) { super(msg); }

}

**How to handle checked exceptions in Streams?**

list.forEach(s -> {

try {

riskyMethod(s);

} catch (Exception e) {

throw new RuntimeException(e);

}

});

**Multiple catch order**

Smaller → Bigger

catch (IOException e) { }

catch (Exception e) { }

**return + exception in finally**

Finally overrides.

**Exception Propagation Flow**

Method A → B → C  
If C throws exception → goes upward until handled.

**What is exception propagation?**

When an exception is not handled in a method, it moves up until some method handles it or JVM terminates the program.

**Can an interface have exception in method?**

Yes.

public interface Test {

void run() throws IOException;

}

**Why do we need Exception Propagation?**

* To avoid handling every exception at every level
* Cleaner & layered architecture
* Only handle exception where you can meaningfully recover

**Empty catch block (Why bad?)**

Hides issues.

catch(Exception e) {}

**Logging Exceptions**

catch (Exception e) {

logger.error("Failed:", e);

}

**Throwing exception from finally**

Legal but VERY BAD.

**Exception in lambda**

Runnable r = () -> {

throw new RuntimeException("Boom");

};

**Exception in Constructor Chaining**

Demo() throws IOException {

this("test"); // if this() fails → exception thrown

}

**Multithreading & Concurrency**

**How many ways can we create thread?**

**1. By extending the Thread class**

A computer code with text

AI-generated content may be incorrect.

You must override run() and call start() (not run() directly).

**2 By implementing the Runnable interface**

**A screen shot of a computer code

AI-generated content may be incorrect.**

Preferred way — because Java supports **multiple inheritance of interfaces**, not classes.

**3. By using Callable and Future (since Java 5)**

A computer code with text

AI-generated content may be incorrect.

Callable can **return a value** and **throw checked exceptions**, unlike Runnable.

**What is the difference between Runnable and Callable?**

A screenshot of a computer

AI-generated content may be incorrect.

A white background with black text

AI-generated content may be incorrect.

Output: Runnable task running by Thread-0

A computer screen shot of a program code

AI-generated content may be incorrect.

Output: Callable task completed by pool-1-thread-1

 **Runnable:** Task, no return, no checked exceptions.

 **Callable:** Task, returns value, can throw checked exceptions.

 **Preferred Use:**

* Runnable → simple background jobs
* Callable → tasks needing result or exception handling

 **Execution:** Both can be executed with ExecutorService.

A screenshot of a computer program

AI-generated content may be incorrect. **What are daemon threads?**

* **Background threads** that support user threads.
* JVM **exits** when all **user threads** finish.
* Examples: Garbage Collector, Finalizer thread.

**Example:**

A computer screen shot of a computer screen

AI-generated content may be incorrect.

**What is synchronization?**

**Synchronization** is a mechanism used in multithreading to ensure that **only one thread accesses a shared resource at a time**.A screen shot of a computer

AI-generated content may be incorrect.

**What is the volatile keyword used for?**

* It **changes to a variable** are **visible across threads**.
* Prevents **thread-local caching** of the variable.

volatile boolean flag = true;

**What is ThreadLocal?**

* Each thread has its **own copy** of a variable.
* Useful for **per-thread data** (like user session, DB connection).

A screen shot of a computer code

AI-generated content may be incorrect.

**What is a deadlock?**

* Occurs when two or more threads **wait for each other’s locks**, and none can proceed.

**Example:**

class A {} // Lock object 1

class B {} // Lock object 2

public class DeadlockExample {

public static void main(String[] args) {

A a = new A(); // resource A

B b = new B(); // resource B

// ---------------- THREAD 1 -----------------

Thread t1 = new Thread(() -> {

// Thread 1 locks object 'a'

synchronized (a) {

System.out.println("Thread 1 locked A");

// Sleep to allow Thread 2 to run and lock B

try { Thread.sleep(100); } catch(Exception e) {}

// Now Thread 1 tries to lock 'b'

System.out.println("Thread 1 trying to lock B...");

synchronized (b) {

// Thread 1 can only reach here if it gets lock B

System.out.println("Thread 1 locked B");

}

}

});

// ---------------- THREAD 2 -----------------

Thread t2 = new Thread(() -> {

// Thread 2 locks object 'b'

synchronized (b) {

System.out.println("Thread 2 locked B");

// Sleep to allow Thread 1 to lock A

try { Thread.sleep(100); } catch(Exception e) {}

// Now Thread 2 tries to lock 'a'

System.out.println("Thread 2 trying to lock A...");

synchronized (a) {

// Thread 2 can only reach here if it gets lock A

System.out.println("Thread 2 locked A");

}

}

});

// Start both threads (they run at the same time)

t1.start();

t2.start();

}

}A screenshot of a computer

AI-generated content may be incorrect.

**What is notify() and notifyAll()?**

* **notify()** → Wakes up **one** waiting thread.
* **notifyAll()** → Wakes up **all** waiting threads on that object.

**What is Callable and Future?**

* Callable<T> → Like Runnable but returns a value and can throw checked exceptions.
* Future<T> → Represents the **result** of an asynchronous computation.

**Example:**

A computer code with black text

AI-generated content may be incorrect.

**What is ExecutorService?**

* Framework to **manage and control thread execution**.
* Replaces manual thread creation (new Thread(...)).

**Example:**

A close-up of a computer screen

AI-generated content may be incorrect.

**What is ForkJoinPool?**

* Used for **parallel tasks** that can be split into smaller subtasks.
* Implements **work-stealing algorithm** — idle threads "steal" work from busy threads.

**Can we start a thread twice?**

❌ **No**, it throws IllegalThredStateException.

A screenshot of a computer

AI-generated content may be incorrect.

**How can we prevent deadlock?**

**Prevention:**  
Acquire locks in **same order**  
Use **tryLock()** with timeout  
Minimize synchronized blocks

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**Pessimistic Locking — “LOCK FIRST, THEN WORK”**

**✔️ Assumption:**

**Conflicts WILL happen**, so we must *prevent them*.

**✔️ Strategy:**

A thread **locks** the record so that **no one else can modify it** until it’s done.

**✔️ Behavior:**

* First transaction locks the row
* Other transactions trying to update it must **WAIT** or **FAIL**
* Ensures safety but reduces concurrency and speed

**✔️ Example:**

Thread 1:

SELECT \* FROM account WHERE id=1 FOR UPDATE;

Thread 2:

blocked... waiting for the lock

**✔️ Used in:**

* Banking transactions
* Critical financial operations
* When conflicts are very likely

**✔️ Pros:**

* Very safe
* Prevents lost updates

**✔️ Cons:**

* Slow
* Other threads get blocked
* Deadlocks can occur

**Optimistic Locking — “WORK FIRST, CHECK LATER”**

**✔️ Assumption:**

**Conflicts are rare**, so don’t lock.

**✔️ Strategy:**

Allow multiple threads to read and modify data **freely** but detect conflict only when updating.

Uses a **version field** (or timestamp).

Example with a version number:

| **Thread** | **Value** | **Version** |
| --- | --- | --- |
| T1 | 500 | v1 |
| T2 | 500 | v1 |

Both read the same version.

**Execution:**

Thread 1 updates:

UPDATE account SET balance=600, version=version+1 WHERE id=1 AND version=1;

Success → version becomes 2.

Thread 2 tries:

UPDATE ... WHERE version=1

But now version is **2**, so update fails.

**✔️ Used in:**

* Web applications
* REST services
* Hibernate/JPA (built-in versioning)
* High concurrency systems

**✔️ Pros:**

* No blocking
* Very fast
* Scales well

**✔️ Cons:**

* If conflicts happen often, many operations will fail and retry

**What is a race condition and how do you prevent it?**

* **Race Condition:** Multiple threads access shared data simultaneously, leading to inconsistent results.
* **Prevention:**  
  ✅ Use **synchronized**, **Lock**, or **Atomic** variables.

A screenshot of a computer

AI-generated content may be incorrect.

**How does Java handle thread safety in immutable objects?**

* Immutable objects (like String, Integer) are **thread-safe** by design:
  + Their **state never changes** after creation.
  + No synchronization needed.

**Basic**

**What are the main features of Java?**

* **Simple** – Easy to learn and use.
* **Object-Oriented** – Everything is treated as an object.
* **Platform-Independent** – Write once, run anywhere (thanks to JVM).
* **Secure** – No direct memory access and has security features.
* **Robust** – Strong memory management, exception handling, and type checking.
* **Multithreaded** – Supports multiple threads running simultaneously.
* **High Performance** – With Just-In-Time (JIT) compiler.
* **Distributed** – Supports networking and remote method invocation.

### **Platform independent** → same compiled code runs everywhere.

### **Portable** → same source code can be moved and recompiled everywhere.

**Explain JDK, JRE, and JVM.**

* **JVM (Java Virtual Machine)** – Runs Java bytecode. Platform-specific.
* **JRE (Java Runtime Environment)** – JVM + libraries needed to run Java programs.
* **JDK (Java Development Kit)** – JRE + tools (compiler, debugger) to write & develop Java programs.

A diagram of a class loader

AI-generated content may be incorrect.

1)Classloader- Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built in classloader in Java.

2)Class(Method) Area:-

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

3)Heap:-It is the runtime data area in which objects are allocated.

4)Stack:- Java stack stores frames. It holds local variables and partial results and plays a part in method invocation and return. Each thread has a private JVM stack, created at the same time as thread. A new frame is created each time a method is invoked. A frame is destroyed when it’s method invocation completes.

5)Program Counter Register:-

PC(Program Counter) register contains the address of the Java Virtual Machine instruction currently being executed.

6)Native Method Stack:-It contains all the native methods used in the application.

7)Execution Engine:-It contains a virtual processor.

Interpreter: Read byte code stream then execute the instructions.

Just-In-Time(JIT)Compiler:-It is used to improve the performance. JIT compiles part of the byte code that have similar functionality at the same time and hence reduces the amount of time needed for compilation. Here the term “Compiler”refers to a translator from the instruction set of a Java Virtual Machine (JVM) to the instruction set of a specific CPU.

8)Java Native Interface(JNI)is a framework which provides an interface to communicate with another application written in another language like C, C++, assembly etc. Java uses JNI framework to send output to the console or interact with OS libraries.

**Why is Java platform-independent?**

* Java code is compiled into **bytecode**, which can run on **any OS** using the JVM.
* “Write Once, Run Anywhere” (WORA).

**Difference between == and .equals().**

* == → Compares **memory addresses** (reference).
* .equals() → Compares **actual content/value**.

**What are wrapper classes?**

* Classes that **wrap primitive types** into objects.
* Example: int → Integer, double → Double.

**What is autoboxing and unboxing?**

* **Autoboxing** – Automatic conversion of primitive → wrapper.

A close-up of a logo

AI-generated content may be incorrect.

**Unboxing** – Automatic conversion of wrapper → primitive.

A close-up of a logo

AI-generated content may be incorrect.

**What is Upcasting?**

Converting a **child class object** reference to a **parent class type**.

**Example**

class Animal {

void sound() {

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

}

}

class Dog extends Animal {

void sound() {

System.out.println("Bark");

}

void fetch() {

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

}

}

public class Test {

public static void main(String[] args) {

Animal a = new Dog(); // ✅ Upcasting

a.sound(); // Bark

// a.fetch(); ❌ Not accessible

}

}

**Key points**

* Animal a = new Dog(); → **upcasting**
* Only **parent methods** are accessible
* **Overridden methods** execute child implementation (runtime polymorphism)
* **Safe** and widely used

**Downcasting**

**What is Downcasting?**

Converting a **parent class reference** back to a **child class type**.

👉 This is **NOT automatic** and requires **explicit casting**.

**✅ Example (Safe Downcasting)**

public class Test {

public static void main(String[] args) {

Animal a = new Dog(); // Upcasting

Dog d = (Dog) a; // ✅ Downcasting

d.fetch(); // Works

}

}

**❌ Example (Unsafe Downcasting)**

Animal a = new Animal();

Dog d = (Dog) a; // ❌ Runtime error

**💥 Output**

ClassCastException

**✅ Safe way: instanceof**

if (a instanceof Dog) {

Dog d = (Dog) a;

d.fetch();

}

**What is Widening?**

Converting a **smaller data type** into a **larger data type**.

✔ Happens **automatically**  
✔ **No data loss**  
✔ Also called **implicit conversion**

**✅ Primitive widening order (Java)**

byte → short → int → long → float → double

**✅ Example**

int a = 10;

double b = a; // ✅ Widening

System.out.println(b); // 10.0

**Why widening is safe?**

* Larger type can **hold all values** of smaller type
* JVM handles conversion automatically

**Narrowing (Explicit Type Conversion)**

**What is Narrowing?**

Converting a **larger data type** into a **smaller data type**.

❌ Not automatic  
⚠️ **Possible data loss**  
✔ Requires **explicit casting**

**✅ Example**

double d = 10.75;

int i = (int) d; // ✅ Narrowing

System.out.println(i); // 10

**❌ Data loss example**

int i = 130;

byte b = (byte) i;

System.out.println(b); // -126

Why?

* byte range: -128 to 127
* Overflow occurs

**More examples**

long l = 1000L;

short s = (short) l;

float f = 12.5f;

int i = (int) f; // decimal lost

**Widening vs Narrowing (Quick Table)**

| **Feature** | **Widening** | **Narrowing** |
| --- | --- | --- |
| Conversion | Smaller → Larger | Larger → Smaller |
| Automatic | ✅ Yes | ❌ No |
| Cast needed | ❌ No | ✅ Yes |
| Data loss | ❌ No | ⚠️ Possible |
| Safety | ✅ Safe | ⚠️ Risky |

**Difference between final, finally, and finalize().**

| **Keyword** | **Meaning** |
| --- | --- |
| final | Constant variable, cannot override or inherit. |
| finally | Block used for cleanup (runs after try-catch). |
| finalize() | Method called by Garbage Collector before destroying an object. |

**What is a constructor? Can it be inherited?**

* Special method used to **initialize objects**.
* Has the same name as the class.
* **Cannot be inherited**, but can be called using super() in subclass.

A screenshot of a phone

AI-generated content may be incorrect.

**Why are strings immutable in Java?**

* **Security** – Prevents changing sensitive data.
* **Caching** – Strings can be shared in String Pool.
* **Thread-safe** – Immutable objects are safe in multi-threading.

**What are access modifiers in Java?**

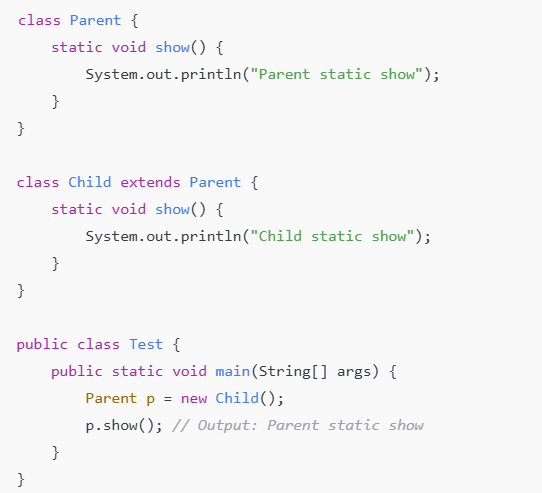
* **private** → Accessible only within the class.
* **default** (no keyword) → Accessible within the package.
* **protected** → Accessible in package + subclasses.
* **public** → Accessible everywhere.

**What is the default value of local variables?**

* **Local variables do NOT have default values.** You must initialize them before use.
* Class/instance variables have default values like 0, null, false.

**Can we override a static method?**

* **No**. Static methods belong to the class, not object. You can **hide** it, not override.



**Constructor chaining?**

**Construction chaining in Java** (more commonly called **constructor chaining**) is the process where **one constructor calls another constructor** within the **same class** or the **parent class**.

**1. Chaining within the same class (this())**

A constructor can call another constructor in the same class using **this()**.

class Example {

Example() {

this(10); // calling another constructor

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

}

Example(int x) {

System.out.println("Parameterized constructor: " + x);

}

}

**Output:**

Parameterized constructor: 10

Default constructor

**2. Chaining to the parent class constructor (super())**

A constructor can call a **parent class constructor** using **super()**.

class Parent {

Parent() {

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

}

}

class Child extends Parent {

Child() {

super(); // calls Parent constructor

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

}

}

**Can we make a constructor private?**

* **Yes**, commonly used in t creation.

**What is the purpose of the this keyword?**

* Refers to the **current object**.
* Used to access class members, call constructors, or pass the current object.

**How does JVM load and execute a Java program?**

1. **Class Loading** – JVM loads .class file using ClassLoader.
2. **Bytecode Verification** – Ensures safety & correctness.
3. **Execution** – JIT compiler converts bytecode to native code, executed in memory.

**Difference between heap and stack memory.**

| **Memory** | **Stores** | **Lifetime** | **Managed by** |
| --- | --- | --- | --- |
| Stack | Local variables & method calls | Short-lived | JVM automatically |
| Heap | Objects & instance variables | Long-lived | Garbage Collector |

**How does Java achieve platform independence?**

* Java code → compiled to **bytecode** → executed by **JVM** on any platform.

**Difference between strong, soft, weak, and phantom references.**

| **Type** | **GC Behavior** |
| --- | --- |
| Strong | Not eligible for GC |
| Soft | Collected only if memory is low |
| Weak | Collected during next GC |
| Phantom | Used for cleanup before object is removed |

Object obj = new Object(); // strong reference

SoftReference<Object> ref = new SoftReference<>(new Object());

WeakReference<Object> ref = new WeakReference<>(new Object());

PhantomReference<Object> phantomRef =

new PhantomReference<>(obj, referenceQueue);

**What happens during class loading (ClassLoader mechanism)?**

1. **Loading** – JVM finds and loads .class.
2. **Linking** – Verify, prepare (allocate memory), and resolve symbolic references.
3. **Initialization** – Execute static blocks and assign default values.

**Difference between Class.forName() and ClassLoader.loadClass().**

* Class.forName() → Loads class **and executes static blocks**.
* ClassLoader.loadClass() → Loads class **without executing static blocks**.

**What are immutable objects? How to create a custom immutable class?**

* **Immutable objects** → Object state cannot be changed after creation.
* **Steps to create**:
  1. Make class final.
  2. Make fields private and final.
  3. No setter methods.
  4. Initialize via constructor only.

**What are memory leaks in Java and how to detect them?**

* **Memory leak** → Objects not used but still referenced → memory wasted.
* **Detect** → Use tools like **VisualVM**, **Eclipse MAT**, or profilers.

**How does the transient keyword work?**

* Prevents **serialization** of a field.
* The field marked transient will **not be saved** when an object is serialized.

**What are the four pillars of OOP?**

* **Encapsulation** → Hiding data and providing access via methods.
* **Abstraction** → Showing only essential details, hiding implementation.
* **Inheritance** → Reusing code from parent class in child class.
* **Polymorphism** → One interface, multiple implementations (method overloading + overriding).

**Difference between class and object**

| **Feature** | **Class** | **Object** |
| --- | --- | --- |
| Definition | Blueprint or template | Instance of a class |
| Memory | No memory allocated | Memory is allocated |
| Example | class Car { ... } | Car c = new Car(); |

**Difference between abstract class and interface**

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| Methods | Can have abstract + concrete methods | Only abstract (Java 8+ can have default & static methods) |
| Variables | Can have instance variables | Only static & final variables |
| Inheritance | Single inheritance | Multiple inheritance possible |
| Constructor | Yes | No |

**What is method overloading?**

* **Same method name, different parameters** in the same class.
* Compile-time polymorphism.



**What is the use of the super keyword?**

* Refers to **parent class**.
* Used to:
  1. Call parent constructor → super()
  2. Access parent methods → super.method()
  3. Access parent variables → super.var

**Can an interface extend another interface?**

* **Yes**, an interface can extend one or more interfaces.

**Can an abstract class have a constructor?**

* **Yes**, used to initialize fields when subclass object is created.

**Can an interface have static methods?**

* **Yes**, from Java 8 onward.
* Called using InterfaceName.methodName().

**Can we achieve multiple inheritance in Java?**

* **No with classes** (to avoid ambiguity).
* **Yes, with interfaces** – a class can implement multiple interfaces.

**What is Serialization and Deserialization in Java with Example?**

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.

**A computer code on a white background

AI-generated content may be incorrect.**

**Mutable vs Immutable - Key Differences**

Mutable Objects (Can Change)

* State can be modified after creation.
* Examples: ArrayList, HashMap, StringBuilder, StringBuffer
* Use Case: When you need to frequently update data**A screenshot of a computer program

  AI-generated content may be incorrect.**

**Immutable Objects (Cannot Change)**

* State cannot be modified after creation.
* Examples: String, List.of(), Set.of(), Map.of(), toList() (Java 16+), Collectors.toUnmodifiableList()
* Use Case: When you need thread safety or data consistency.

**A screenshot of a computer code

AI-generated content may be incorrect.**

**What is the difference between String Buffer and String Builder?**

**StringBuffer** A StringBuffer in Java is a mutable sequence of characters that is thread-safe because its methods are synchronized.

StringBuffer sbf = new StringBuffer("Hi");

sbf.append(" Java");

System.out.println(sbf); // Hi Java

**StringBuilder**  
A StringBuilder in Java is a **mutable sequence of characters** that is **not synchronized**, making it **faster** than StringBuffer.

StringBuilder sb = new StringBuilder("Hello");

sb.append(" World");

System.out.println(sb); // Hello World

**What is the difference between collection and collections?**

**Collection**

* **Type:** **Interface**
* **Package:** java.util.Collection
* **Definition:** The root interface in the **collection hierarchy** that represents a group of objects, known as elements.
* **Purpose:** To define basic operations for data structures like **List, Set, Queue**, etc.
* **Key Implementing Classes:**
  + ArrayList (List)
  + HashSet (Set)
  + LinkedList (List, Queue)

A computer screen shot of a code

AI-generated content may be incorrect.

**Collections**

* **Type:** **Class** (final utility class)
* **Package:** java.util.Collections
* **Definition:** A **utility class** that provides static methods to operate on or return **collection objects**.
* **Purpose:** Sorting, reversing, searching, synchronized wrappers, etc.
* **Key Methods:**
  + sort(List<T> list)
  + reverse(List<?> list)
  + shuffle(List<?> list)
  + synchronizedList(List<T> list)
* **Example:**

A screen shot of a computer program

AI-generated content may be incorrect.

**What is a Hash map?**

A **HashMap** in Java is a part of the **java.util** package that stores **key–value pairs**.  
It allows **fast retrieval**, **insertion**, and **deletion** based on the **hashing technique**

A computer screen shot of a code

AI-generated content may be incorrect.

**What is mean by Immutable?**

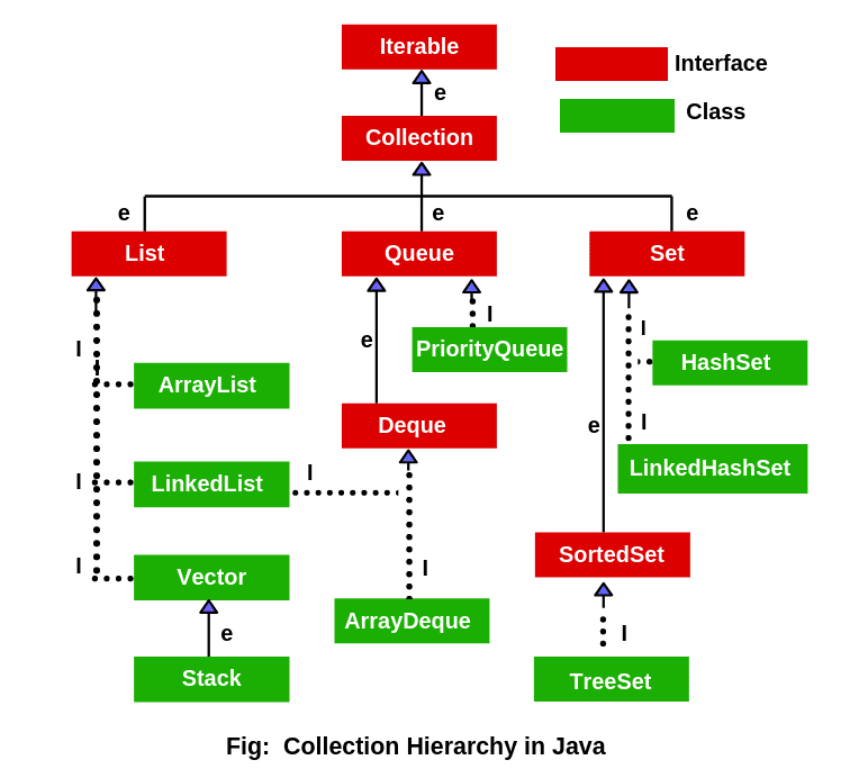
Once an object is created, its **state (data)** cannot be modified.

A close-up of a computer screen

AI-generated content may be incorrect.

**Collections Framework**

Collection is an interface used to store a group of objects, while Collections is a utility class that provides static methods to manipulate and work with collection objects.

****

A screenshot of a computer

AI-generated content may be incorrect.

**Examples:**

* ArrayList, LinkedList → List
* HashSet, TreeSet → Set
* HashMap, TreeMap → Map

A screenshot of a computer

AI-generated content may be incorrect.

**What is HashSet?**

HashSet stores elements using a **hash table**.

* No insertion order
* No sorting
* Very fast for add, remove, search

**Example**

Set<Integer> set = new HashSet<>();

set.add(3);

set.add(1);

set.add(2);

set.add(2); // duplicate

System.out.println(set); // Output order NOT guaranteed

**🔍 Output (example)**

[1, 2, 3] // OR [3, 1, 2] etc.

**TreeSet**

**✅ What is TreeSet?**

TreeSet stores elements in a **sorted order** using a **Red-Black Tree**.

* Sorted by **natural order**
* Or by a **Comparator**

**✅ Example**

Set<Integer> set = new TreeSet<>();

set.add(3);

set.add(1);

set.add(2);

System.out.println(set);

**🔍 Output**

[1, 2, 3]

**❌ Null in TreeSet**

Set<Integer> set = new TreeSet<>();

set.add(null); // ❌ NullPointerException

A screenshot of a hashtag

AI-generated content may be incorrect.

**What is HashMap?**

HashMap stores data in **key–value pairs** and is **not thread-safe** by default.

**Example**

Map<Integer, String> map = new HashMap<>();

map.put(1, "Java");

map.put(2, "Spring");

map.put(null, "API"); // allowed

map.put(3, null); // allowed

System.out.println(map);

**Key characteristics**

* Allows **1 null key**
* Allows **multiple null values**
* High performance
* Use in **single-threaded** applications

**Hashtable**

**What is Hashtable?**

Hashtable is a **legacy**, **thread-safe** map where **all methods are synchronized**.

**❌ Null restriction**

Hashtable<Integer, String> table = new Hashtable<>();

table.put(null, "Java"); // ❌ NullPointerException

table.put(1, null); // ❌ NullPointerException

**Key characteristics**

* Thread-safe by default
* No null keys or values
* Slower due to synchronization
* Rarely used in modern code

**Fail-fast vs Fail-safe**

**HashMap (Fail-fast)**

Map<Integer, String> map = new HashMap<>();

for (Integer key : map.keySet()) {

map.put(4, "New"); // ❌ ConcurrentModificationException

}

**Hashtable (Enumerator)**

* Enumerator is **not fail-fast**
* Does not throw ConcurrentModificationException

**Modern alternative to Hashtable**

Instead of Hashtable, use:

Map<K, V> map = Collections.synchronizedMap(new HashMap<>());

or (best option):

ConcurrentHashMap<K, V> map = new ConcurrentHashMap<>();

A screenshot of a phone

AI-generated content may be incorrect.

**What is a fail-fast iterator?**

* If a collection is **structurally modified** while iterating (except via iterator’s remove ()), it throws a **ConcurrentModificationException**.
* Applies to: ArrayList, HashSet, HashMap.

**Example:**

A close-up of a logo

AI-generated content may be incorrect.

**How does HashMap work internally?**

1. Stores data as **key-value pairs** in **buckets** (array of Node).
2. Key’s hashCode() → used to find bucket index.
3. If collision occurs (same bucket), stores as a **linked list** or **tree (after JDK 8)**.
4. Uses **equals()** to check key equality.

**What is the load factor in HashMap?**

* Load factor defines **when to increase bucket size**.
* Default = **0.75 (75%)**
* When the map reaches 75% of its capacity → it **rehashes** (doubles size).

**What is Comparable?**

Comparable is used when:

A class has **one natural way** of sorting.

**Method**

int compareTo(T o)

**Example**

class Employee implements Comparable<Employee> {

int id;

String name;

double salary;

Employee(int id, String name, double salary) {

this.id = id;

this.name = name;

this.salary = salary;

}

@Override

public int compareTo(Employee e) {

return this.id - e.id; // natural order by id

}

}

List<Employee> list = new ArrayList<>();

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

**Key points**

* Sorting logic is **inside the class**
* Only **one sorting sequence**

**Comparator**

**What is Comparator?**

Comparator is used when:

You need **multiple or external sorting logics**.

**✅ Method**

int compare(T o1, T o2)

**✅ Example**

Comparator<Employee> bySalary =

(e1, e2) -> Double.compare(e1.salary, e2.salary);

Comparator<Employee> byName =

(e1, e2) -> e1.name.compareTo(e2.name);

Collections.sort(list, bySalary);

Collections.sort(list, byName);

A screenshot of a computer

AI-generated content may be incorrect.

Both are used to sort objects, but they differ in **where** and **how** the comparison logic is written.

**How does LinkedHashMap maintain insertion order?**

* Uses a **doubly linked list** internally connecting entries.
* Preserves **insertion order** (or access order if configured).

**What are weak keys in WeakHashMap?**

* In WeakHashMap, keys are **weakly referenced**.
* If a key has **no strong reference**, it is **garbage collected automatically**.

**Can we store null keys in HashMap?**

✅ Yes, **only one null key** is allowed.  
Internally, it is stored in a special bucket (index = 0).

**What is a PriorityQueue?**

* A queue where elements are ordered by **priority** (natural order or custom comparator).
* Head of queue = **smallest/highest priority** element.
* Implemented using a **binary heap**.

**What are NavigableSet and NavigableMap?**

* **NavigableSet** → Extends SortedSet (methods like lower(), higher(), floor(), ceiling()).
* **NavigableMap** → Extends SortedMap (methods like lowerKey(), higherKey()).

**Examples:**  
TreeSet, TreeMap.

**What happens if two keys have the same hashcode in a HashMap?**

* They go to the **same bucket**.
* Then **equals()** method is used to differentiate between keys.
* If equals() returns true → new value replaces old one.

**How does ConcurrentHashMap achieve thread safety without locking the entire map?**

* Uses **segment-level (bucket-level) locking** (JDK 7) or **CAS (Compare-And-Swap)** operations (JDK 8).
* Allows multiple threads to read/write to different buckets **concurrently**.

A screenshot of a computer

AI-generated content may be incorrect.

**How does TreeMap maintain sorting order?**

* Sorted based on **natural order** of keys or a **Comparator** provided during creation.
* Internally uses a **Red-Black Tree**.

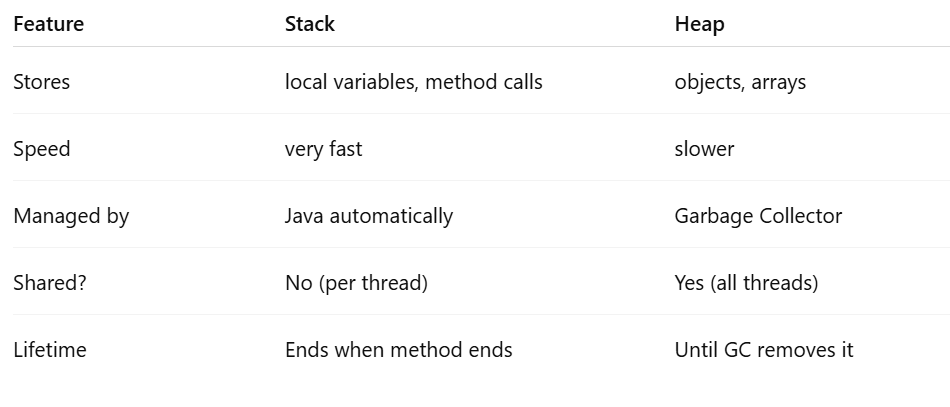
**When would you prefer LinkedHashMap over HashMap?**

Use **LinkedHashMap** when:

* You need **predictable iteration order** (insertion/access order).
* Example: Implementing **LRU Cache** or maintaining **ordered output**.

**Memory Management & JVM**

**1. What are the main memory areas in JVM?**



**What is Garbage Collection in Java? How does it work?**

Garbage Collection automatically frees memory by removing unused objects.

**GC Process:**

1. **Mark Phase** → Identify reachable objects.
2. **Sweep Phase** → Remove unreferenced objects.
3. **Compact Phase** → Rearranges objects to avoid fragmentation.

**GC Roots include:**

* Local variables
* Static variables
* Active threads
* JNI handles

**What are memory leaks in Java?**

Memory leaks happen when:

* Unused objects remain reachable.
* So, GC cannot delete them.

Common causes:

* Static collections
* Forgot to remove listeners
* Incorrect caching
* Inner classes holding outer class references

**What is a ClassLoader?**

Loads .class files into JVM.

Types:

* **Bootstrap ClassLoader**
* **Extension ClassLoader**
* **Application (System) ClassLoader**
* **Custom ClassLoader**

They follow **parent delegation model**.

**StackOverflowError vs OutOfMemoryError**

**StackOverflowError**

* Stack becomes full.
* Cause: infinite recursion or deep method nesting.

**OutOfMemoryError (OOM)**

* JVM cannot allocate memory in:
  + Heap
  + Metaspace
  + DirectBuffer
  + GC overhead limit exceeded

**What is JIT Compiler?**

JIT = **Just-In-Time Compiler**  
Part of JVM that converts **frequently executed bytecode → native machine code** for faster execution.

Optimizations include:

* Method inlining
* Escape analysis
* Loop unrolling
* Dead code elimination

**Difference between Serial, Parallel, and G1 Garbage Collectors**

**Serial GC**

Single-threaded.

* Small applications & single-core CPUs.

**Parallel GC**

Multi-threaded young generation GC.

* Focus: **Throughput**.
* Good for batch jobs.

**G1 GC (Default in Java 11+)**

* Region-based.
* Predictable pause times.
* Good for large heaps (4GB+).

**What is Class Unloading?**

JVM unloads classes when:

* Their ClassLoader becomes unreachable.
* No active references exist.

Prevents Metaspace leaks.

**Design Patterns**

**What is Singleton Pattern?**

Singleton Pattern ensures that:

1. **Only one instance** of a class exists.
2. That instance is **globally accessible**.

public class Singleton {

private static Singleton instance;

private Singleton() { } // private constructor prevents other classes from creating objects

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

**Use cases:**

* Database connection
* Logger
* Configuration manager

**what is Factory Pattern?**

Factory Pattern provides a method to create objects **without exposing the creation logic** to the client.

**Example:**

A computer code with colorful text

AI-generated content may be incorrect.

**Use cases:**

* When object creation is complex
* When you want to decouple object creation from client code

**What is Builder Pattern?**

Builder Pattern helps construct **complex objects step by step**.

**Example:**

A screenshot of a computer code

AI-generated content may be incorrect.

**Use cases:**

* When a class has many optional parameters
* For immutable objects
* Avoid telescoping constructors

**What is Prototype Pattern?**

Prototype Pattern creates new objects by **copying existing objects** instead of instantiating new ones.

**Example:**

A screen shot of a computer code

AI-generated content may be incorrect.

**Use cases:**

* When object creation is expensive
* When we need many similar objects

**What is Observer Pattern?**

Observer Pattern defines a **one-to-many relationship** between objects.

When one object changes its state → all dependents (observers) are notified.

**Example:**

A screenshot of a computer program

AI-generated content may be incorrect.

**Use cases:**

* Event listeners
* UI frameworks
* Real-time data updates

**What is Strategy Pattern?**

Strategy Pattern allows you to choose an algorithm **at runtime**.

**Example:**

A screen shot of a computer code

AI-generated content may be incorrect.

**Use cases:**

* Payment methods
* Sorting strategies
* Compression algorithms

**What is Dependency Injection?**

Dependency Injection (DI) is when objects receive their dependencies **from outside**, instead of creating them inside.

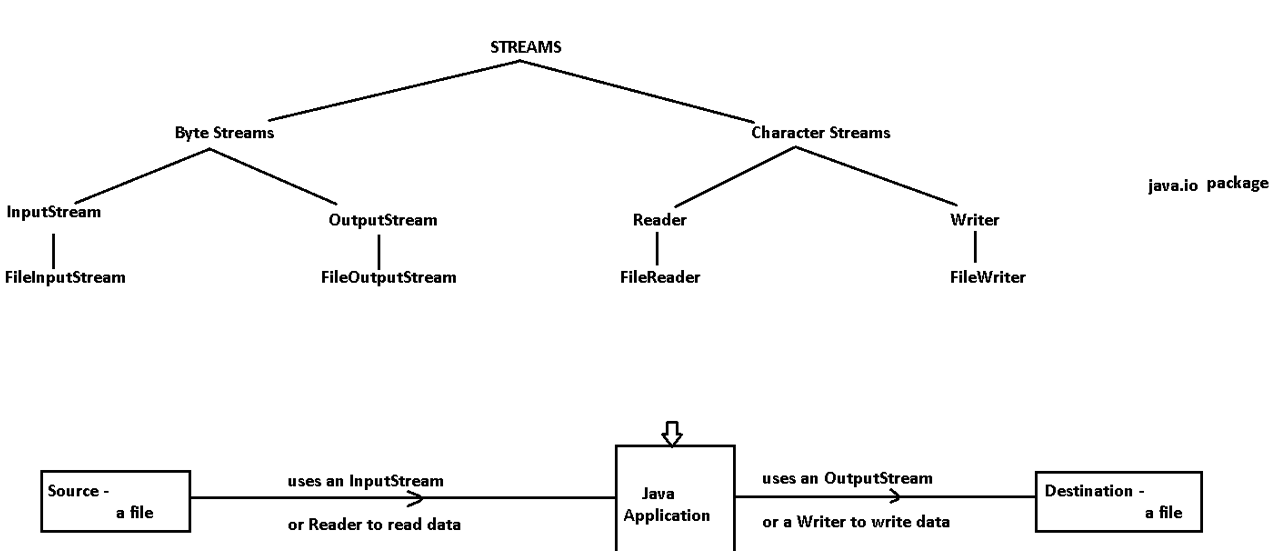
**A screenshot of a computer program

AI-generated content may be incorrect.**

**Use cases:**

* Spring Framework
* Loose coupling
* Unit testing (mocking)

**Java I/O and Serialization**



**What is Java I/O?**

**Answer:**  
Java I/O is used to **read data from a source** and **write data to a destination** such as files, keyboard, network, or memory using streams.

**What is a Stream in Java?**

**Answer:**  
A stream is a **unidirectional flow of data** between a Java program and an external source.

**1. Difference between FileReader and BufferedReader**

**FileReader**

* Reads characters directly from a file.
* Slow → performs one read operation per character.
* No buffering.
* Used for simple reading.

**BufferedReader**

* Wraps around a Reader (like FileReader).
* Reads data in **large chunks** using an internal buffer → **faster**.
* Provides advanced methods like readLine().
* Used for performance.

**Difference between byte stream and character stream**

**Byte Stream**

* Reads raw bytes (8-bit).
* Uses InputStream / OutputStream.
* Suitable for:
  + images
  + videos
  + PDFs
  + binary files

**Character Stream**

* Reads 16-bit Unicode characters.
* Uses Reader / Writer.
* Suitable for text files.

**Difference between FileInputStream and BufferedInputStream**

| **FileInputStream** | **BufferedInputStream** |
| --- | --- |
| No buffer | Uses buffer |
| Slow | Fast |
| Direct disk access | Memory access |

**What is InputStreamReader?**

**Answer:**  
A **bridge class** that converts **byte streams to character streams** and supports character encoding.

✔ Preferred over FileReader

**What is FileInputStream?**

**Answer:**  
Reads **raw bytes** from a file **one byte at a time**.

❌ Slow  
❌ No buffering

**Why do we need Serialization?**

**Answer:**

* Save object to file
* Send object over network
* Caching
* RMI / distributed systems

**What gets serialized?**

✔ Instance variables  
✔ Object references

❌ static variables  
❌ transient variables

**What is BufferedInputStream?**

**Answer:**  
A wrapper over InputStream that **reads data in chunks using buffer**, improving performance.

✔ Faster  
✔ Fewer disk accesses

**What is serialVersionUID?**   
A **unique version identifier** used to verify compatibility during deserialization.

**What is the role of serialVersionUID?**

serialVersionUID ensures that **sender and receiver** of a serialized object have **compatible class versions**.

* If UID mismatches → InvalidClassException
* Good practice: always manually define it.

private static final long serialVersionUID = 1L;

**STEP 1: Employee class (WITH serialVersionUID)**

import java.io.Serializable;

public class Employee implements Serializable {

// 🔑 serialVersionUID for version control

private static final long serialVersionUID = 1L;

int id;

String name;

public Employee(int id, String name) {

this.id = id;

this.name = name;

}

}

**✅ STEP 2: Serialize the object**

import java.io.FileOutputStream;

import java.io.ObjectOutputStream;

public class SerializeEmployee {

public static void main(String[] args) {

try {

Employee emp = new Employee(101, "Sridhar");

ObjectOutputStream oos =

new ObjectOutputStream(new FileOutputStream("employee.ser"));

oos.writeObject(emp);

oos.close();

System.out.println("Employee object serialized successfully");

} catch (Exception e) {

e.printStackTrace();

}

}

}

👉 Run this program first  
👉 It creates **employee.ser**

**✅ STEP 3: Deserialize the object**

import java.io.FileInputStream;

import java.io.ObjectInputStream;

public class DeserializeEmployee {

public static void main(String[] args) {

try {

ObjectInputStream ois =

new ObjectInputStream(new FileInputStream("employee.ser"));

Employee emp = (Employee) ois.readObject();

System.out.println("Employee Data:");

System.out.println("ID : " + emp.id);

System.out.println("Name : " + emp.name);

ois.close();

} catch (Exception e) {

e.printStackTrace();

}

}

}

👉 Run this → **deserialization succeeds**

**🔥 NOW THE IMPORTANT PART (ROLE OF serialVersionUID)**

**STEP 4: MODIFY the Employee class (version change)**

Now **change the Employee class** like this 👇  
(**Do NOT change serialVersionUID**)

import java.io.Serializable;

public class Employee implements Serializable {

private static final long serialVersionUID = 1L; // SAME UID

int id;

String name;

String department; // ✅ NEW FIELD ADDED

public Employee(int id, String name) {

this.id = id;

this.name = name;

}

}

👉 Now **run only DeserializeEmployee**

**✅ OUTPUT**

Employee Data:

ID : 101

Name : Sridhar

✔ Deserialization works  
✔ New field is ignored  
✔ **serialVersionUID saved you**

**❌ WHAT IF YOU CHANGE serialVersionUID?**

Change UID to:

private static final long serialVersionUID = 2L;

Now run **DeserializeEmployee** again.

**❌ OUTPUT**

java.io.InvalidClassException:

Employee; local class incompatible:

stream classdesc serialVersionUID = 1,

local class serialVersionUID = 2

**🧠 WHAT THIS PROVES**

* serialVersionUID is stored **inside the .ser file**
* JVM compares:
* UID in file vs UID in class
* If mismatch → ❌ exception
* If match → ✅ deserialization works

**How do you make a class non-serializable?**

**Method 1: Do not implement Serializable (simplest)**

**Method 2: Make fields transient**

transient int id;

**Method 3: Throw exception in writeObject()**

private void writeObject(ObjectOutputStream o) throws NotSerializableException {

throw new NotSerializableException();

}

Method 4: Use a non-serializable parent class

**1️⃣ Read a Text File (Using FileReader)**

import java.io.FileReader;

import java.io.IOException;

public class ReadFileUsingFileReader {

public static void main(String[] args) throws IOException {

FileReader fr = new FileReader("data.txt");

int ch;

while ((ch = fr.read()) != -1) {

System.out.print((char) ch);

}

fr.close();

}

}

📌 Reads **character by character** (slow, basic)

**2️⃣ Read a Text File (Using BufferedReader)**

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

public class ReadFileUsingBufferedReader {

public static void main(String[] args) throws IOException {

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

String line;

while ((line = br.readLine()) != null) {

System.out.println(line);

}

br.close();

}

}

✔ Fast  
✔ Uses buffer  
✔ Line-by-line reading

**3️⃣ Write to a File (Using FileWriter)**

import java.io.FileWriter;

import java.io.IOException;

public class WriteFileUsingFileWriter {

public static void main(String[] args) throws IOException {

FileWriter fw = new FileWriter("output.txt");

fw.write("Hello Java I/O\n");

fw.write("File writing example");

fw.close();

}

}

**4️⃣ Copy a File (Using Byte Streams) ⭐ Interview Favorite**

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class CopyFile {

public static void main(String[] args) throws IOException {

FileInputStream fis = new FileInputStream("source.jpg");

FileOutputStream fos = new FileOutputStream("dest.jpg");

int b;

while ((b = fis.read()) != -1) {

fos.write(b);

}

fis.close();

fos.close();

}

}

✔ Works for **binary files**

**5️⃣ Copy a File Faster (Using Buffered Streams)**

import java.io.\*;

public class CopyFileBuffered {

public static void main(String[] args) throws IOException {

BufferedInputStream bis =

new BufferedInputStream(new FileInputStream("source.jpg"));

BufferedOutputStream bos =

new BufferedOutputStream(new FileOutputStream("dest.jpg"));

int b;

while ((b = bis.read()) != -1) {

bos.write(b);

}

bis.close();

bos.close();

}

}

✔ Much faster  
✔ Best practice

**6️⃣ Read Keyboard Input (Using BufferedReader)**

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.io.IOException;

public class ReadFromKeyboard {

public static void main(String[] args) throws IOException {

BufferedReader br =

new BufferedReader(new InputStreamReader(System.in));

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

String name = br.readLine();

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

}

}

**7️⃣ Write Primitive Data (DataOutputStream)**

import java.io.\*;

public class WritePrimitiveData {

public static void main(String[] args) throws IOException {

DataOutputStream dos =

new DataOutputStream(new FileOutputStream("data.bin"));

dos.writeInt(100);

dos.writeDouble(99.5);

dos.writeBoolean(true);

dos.close();

}

}

**8️⃣ Read Primitive Data (DataInputStream)**

import java.io.\*;

public class ReadPrimitiveData {

public static void main(String[] args) throws IOException {

DataInputStream dis =

new DataInputStream(new FileInputStream("data.bin"));

System.out.println(dis.readInt());

System.out.println(dis.readDouble());

System.out.println(dis.readBoolean());

dis.close();

}

}

**9️⃣ Serialize an Object**

import java.io.\*;

class Employee implements Serializable {

int id;

String name;

Employee(int id, String name) {

this.id = id;

this.name = name;

}

}

public class SerializeObject {

public static void main(String[] args) throws IOException {

Employee e = new Employee(1, "John");

ObjectOutputStream oos =

new ObjectOutputStream(new FileOutputStream("emp.ser"));

oos.writeObject(e);

oos.close();

System.out.println("Object Serialized");

}

}

**🔟 Deserialize an Object**

import java.io.\*;

public class DeserializeObject {

public static void main(String[] args)

throws IOException, ClassNotFoundException {

ObjectInputStream ois =

new ObjectInputStream(new FileInputStream("emp.ser"));

Employee e = (Employee) ois.readObject();

System.out.println(e.id + " " + e.name);

ois.close();

}

}

**1️⃣1️⃣ Serialization with transient**

import java.io.\*;

class User implements Serializable {

String username;

transient String password;

User(String u, String p) {

username = u;

password = p;

}

}

After deserialization:

password = null

**1️⃣2️⃣ try-with-resources (Best Practice)**

import java.io.\*;

public class TryWithResourcesExample {

public static void main(String[] args) {

try (BufferedReader br =

new BufferedReader(new FileReader("data.txt"))) {

System.out.println(br.readLine());

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Tricky / Scenario-Based**

**Can we override static or private methods?**

**Static methods:** ❌ Cannot be overridden → they can only be *hidden*.  
**Private methods:** ❌ Cannot be overridden → not visible to subclasses.

**What happens if constructor throws an exception?**

* Object creation **fails**.
* The partially created object is **not returned**.
* Memory is eligible for GC.

**Can a class be immutable? How?**

Yes.

To make class immutable:

1. Make class final.
2. Make fields private and final.
3. No setters.
4. Initialize fields in constructor.
5. For mutable fields, return **defensive copies**.

**What happens if you modify a key in HashMap after insertion?**

The key becomes **unreachable**.  
HashMap can no longer find the bucket → get() returns null.

**What is pass-by-value vs pass-by-reference?**

**Java is always pass-by-value.**

* For primitives → value passed.
* For objects → reference *copy* passed.

**Can we use System.exit() in try block?**

Yes.  
JVM terminates immediately — finally block **will NOT run**.

**What happens if we start the same thread twice?**

Throws:

java.lang.IllegalThreadStateException

A thread can be started **only once**.

**How can we make a thread-safe singleton?**

Best way → **Bill Pugh Singleton**.

class Singleton {

private Singleton(){}

private static class Holder {

static final Singleton INSTANCE = new Singleton();

}

public static Singleton getInstance(){

return Holder.INSTANCE;

}

}

* Lazy loading
* Thread safe
* No synchronization overhead

**Can we synchronize static methods?**

Yes.  
static synchronized locks **Class object**, not this.

**Can we call one constructor from another?**

Yes, using this().  
Must be the **first statement**.

**What happens if you change a key’s hashCode after inserting into a HashMap?**

Same as modifying the key → Map loses track of entry.  
Key will not be found during lookup.

**an you override equals() without overriding hashCode()?**

You *can*, but you **should NEVER**.

Violates Java contract → breaks HashMap / HashSet.

**What happens if a constructor throws an exception?**

(Same as #2)  
Object **does not get created**, and constructor must handle or propagate the exception.

**Can we make an immutable class with mutable fields?**

Yes.  
Return **deep copies** of mutable fields:

public Date getDob() {

return new Date(dob.getTime());

}

**Can we serialize a singleton class?**

Yes, but serialization creates a **new object**, breaking singleton.

Fix with:

protected Object readResolve() {

return getInstance();

}

**What happens if we call wait() without holding the monitor lock?**

Throws:

java.lang.IllegalMonitorStateException

wait(), notify(), notifyAll() must be inside synchronized.

**How do you detect and solve deadlocks in Java?**

**Detect**

* Thread dump (jstack)
* VisualVM / JConsole
* Look for threads in BLOCKED state, waiting on each other.

**Solve**

* Use consistent lock ordering
* Reduce synchronized blocks
* Use tryLock() with timeout
* Avoid nested locking

**Can we start the same thread twice?**

(Repeated) → **No**. Throws IllegalThreadStateException.

**How does System.gc() work?**

* Suggests JVM to run GC
* No guarantee JVM will actually run GC
* Under the hood → calls Runtime.getRuntime().gc()

**What’s the difference between shallow and deep copy?**

**Shallow Copy**

* Copies object references only
* Nested objects are shared
* clone() default behavior

**Deep Copy**

* Copies *all* fields + nested objects
* Fully independent clone

**What happens when a thread calls join() on itself?**

Thread waits for itself forever → **deadlock** (self-deadlock).  
Thread never proceeds.

**JWT TOKEN**

**What is authentication and authorization?**

* Authentication is the process of **verifying the identity** of a user.
* Authorization is the process of **checking user permissions** after they are authenticated.A close-up of a computer code

  Description automatically generated

**What are some common approaches to implement authentication in Java applications?**

* Basic Authentication
* OAuth2
* JWT Tokens
* Session-based Authentication (e.g., Spring Security)

**Basic Authentication:.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A computer screen shot of a program

AI-generated content may be incorrect.**

**Note:** Write controller class otherwise you will get **404 not found** error.

**JWT Tokens**

**Pom.xml**

**JwtUtilityClass**

**JwtAuthenticationFilterClass**

**SecuriyConfigClass**

**AuthenticationControllerClass**

**A computer screen shot of a program

AI-generated content may be incorrect.**

**Create JWT Utility Class:**

**A computer code with text

AI-generated content may be incorrect.**

**Create a JwtAuthenticationFilter**

**A computer screen shot of a program

AI-generated content may be incorrect.**

**A computer code with text

AI-generated content may be incorrect.**

**Configure Spring Security**

**A screenshot of a computer code

AI-generated content may be incorrect.**

**A computer code with text

AI-generated content may be incorrect.**

**Create AuthController to Generate Tokens**

**A close-up of a computer code

AI-generated content may be incorrect.What is the role of Spring Security in authentication and authorization?**

* Spring Security provides configurable tools for authentication and authorization, including form-based login, role-based access control.

**How do role-based access control (RBAC) and attribute-based access control (ABAC) differ?**

* + RBAC assigns permissions based on roles (e.g., ADMIN, USER).
  + ABAC considers attributes of the user, environment, and resource (e.g., age > 18).

**What is a JWT, and why is it used?** (**JSON Web Token) (JavaScript Object Notation)**

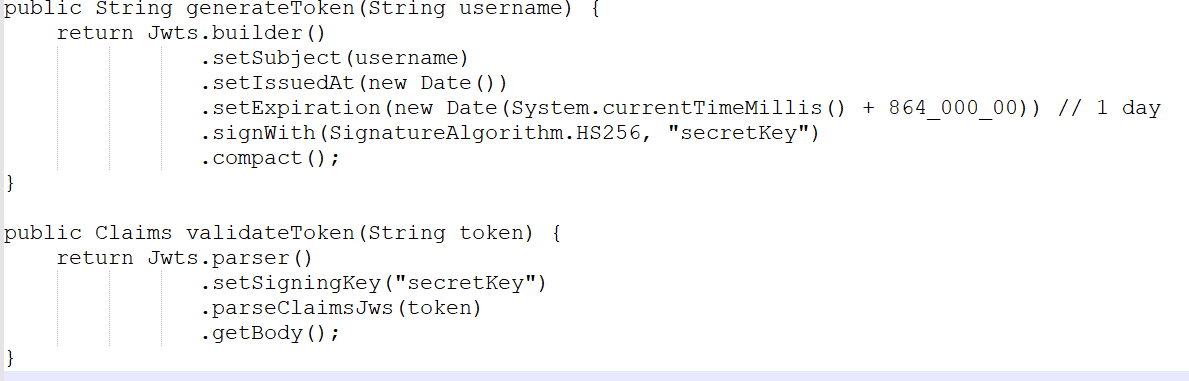
JSON Web Token (JWT) is a way to securely transmit information between parties using JSON. JWTs are commonly used for authentication and authorization. They are compact and can be sent in URLs, HTTP headers, or POST parameters.

**Explain the structure of a JWT.**

* + A JWT consists of three parts:
    1. **Header**: Metadata (e.g., type and algorithm).
    2. **Payload**: Claims (e.g., user data, roles).
    3. **Signature**: Validates token integrity.

**Example Token:** eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1c2VyMSIsImFkbWluIjp0cnVlfQ.Xu4PvWo74u6KYZJImoGfb8Uu6\_4PPbgZNrV17shWo1M

**How can you implement JWT authentication in a Java application?**



Generate Token:

1. **Jwts.builder()** → Creates a new JWT.
2. **.setSubject(username**) → Stores **username** in the token payload (claims).
3. **.setIssuedAt(new Date())** → Adds the **current timestamp** to track token issuance.
4. **.setExpiration(new Date(System.currentTimeMillis() + EXPIRATION\_TIME))** →
   * Sets an **expiration time** (current time + 1 day).
5. **.signWith(key, SignatureAlgorithm.HS256) →**
   * Signs the JWT using **HMAC SHA-256** with the secret key.
6. **.compact()** → Converts everything into a **JWT string**.

Validate Token:

**Jwts.parser()**

* Creates a **JWT parser** instance to process the token.

**.setSigningKey("secretKey")**

* Sets the **secret key** used to **validate the token's signature**.
* The same key used to **sign** the token must be used for **validation**.

**.parseClaimsJws(token)**

* **Parses the JWT token**, verifying:  
  ✅ It is **well-formed**.  
  ✅ It **hasn't been tampered with**.  
  ✅ The **signature matches** using the given secret key.

**.getBody()**

* Extracts the **claims (payload)** from the token.
* The payload contains **user-related information**, like username, roles, issued time, and expiration.

**What are claims in a JWT, and what are the differences between registered, public, and private claims?**

Claims are pieces of information (key-value pairs) stored in the token to convey details about a user or an entity.

* + 1. Registered claims are standardized (e.g., iss, sub, exp).
    2. Public claims are user-defined but must avoid collisions.
    3. Private claims are used for custom data shared between parties.

**How do you secure a JWT?**

* + Use strong secret keys.
  + Set short expiration times.
  + Implement refreshing tokens.

**How would you implement role-based access control using JWT in a Spring Boot application?**

A close up of text

Description automatically generated

**What would you do if a JWT token expires?**

Implement a refresh token mechanism to issue a new token without requiring re-authentication.

**Can a JWT be modified or tampered with? How would you prevent this?**

A JWT can be tampered with, but tampering invalidates the signature. Always verify the token's signature using the secret key.

**How would you log out a user in a JWT-based system?**

* Add the JWT to the blacklist.
* Track invalidated tokens in a database or cache.

**What are the common errors you might encounter when using JWT?**

* Expired token error.
* Signature verification failure.
* Malformed token error.

**Explain the difference between symmetric and asymmetric encryption in JWT signing.**

* Symmetric uses the same key for signing and verification (e.g., HS256).
* Asymmetric uses a public-private key pair (e.g., RS256).

**How can you handle multi-factor authentication (MFA) in a JWT-based system?**

* Add an intermediate step after verifying the password to validate a second factor (e.g., OTP) and then issue a JWT.

**How can you implement token-based authentication for APIs in Spring Boot?**

* + Use a filter to intercept requests and validate JWTs:

A screenshot of a computer code

Description automatically generated

**Session-Based vs Token-Based Authentication:**

**Session-Based Authentication**

The server creates and stores a session for an authenticated user and uses cookies to track the session.

**Token-Based Authentication**

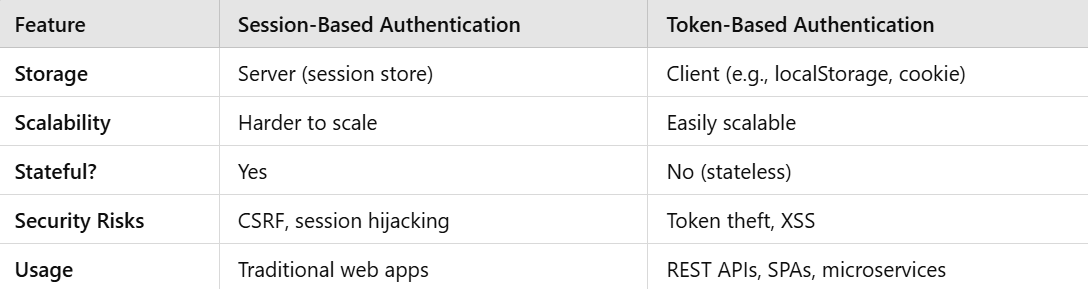
The server issues a token (typically JWT) after authentication, and the client sends the token in requests to authenticate itself.

In token-based authentication, when a user logs in with valid credentials, the server generates a **token** (commonly a JWT – JSON Web Token). This token is returned to the client.

The client stores the token (usually in browser localStorage or sessionStorage) and sends it with every request — usually in the **Authorization header**:

Authorization: Bearer <token>

The server does **not** store session data. Instead, it verifies the token on each request.  
If the token is valid, the server allows access; if not, it rejects the request.



**RESTful API**

**A close-up of a computer code

AI-generated content may be incorrect.**

**What is API?**

API (Application Programming Interface) is a set of rules and protocols that allow different software applications to communicate with each other

**What is a RESTful API?**

A **RESTful API** is an **API that follows REST (Representational State Transfer) principles** to enable communication between clients (e.g., web browsers, mobile apps) and servers over the **HTTP/HTTPS** protocol..

**What are HTTP methods in REST?**

**GET**: Retrieve a resource.

**POST**: Create a new resource.

**PUT**: Update an existing resource.

**DELETE**: Remove a resource.

**PATCH**: Partial update of a resource.

**What is the difference between PUT and POST?**

**POST:** Creates a new resource. It’s not idempotent (sending the same request multiple times can create multiple resources).

**PUT:** Updates or creates a resource at a specific URI. It’s idempotent (repeated requests have the same effect).

**What is idempotence in REST?**

An operation is idempotent if repeated execution has the same result.

Example: Repeated GET or DELETE requests don’t change the state of the server.

**How do you create a RESTful API in Java?**

By using frameworks like Spring Boot, JAX-RS (Jersey/RESTEasy), or SparkJava.

**Example in Spring Boot:**

**A computer code with black text

Description automatically generated**

**What is @RestController in Spring Boot?**

It’s a specialized version of @Controller that combines @Controller and @ResponseBody. It’s used for creating RESTful web services.

**How do you handle exceptions in a REST API?**

Use @ControllerAdvice and @ExceptionHandler for global exception handling.

**Example:**

**A close-up of a computer code

Description automatically generated**

**What is @RequestMapping vs @GetMapping in Spring Boot?**

@RequestMapping is a versatile annotation that maps HTTP methods and paths.

@GetMapping is a shorthand for @RequestMapping(method = RequestMethod.GET).

**How do you secure a REST API in Java?**

Use Spring Security to secure endpoints. Example:

**A close-up of a computer code

Description automatically generated**

Use JWT for token-based security.

**How do you handle pagination in a REST API?**

Pass page and size as query parameters: Example: /api/resources?page=0&size=10

Response includes metadata for total pages, current page, etc.

**How do you test REST APIs in Java?**

Use tools like Postman or Curl for manual testing.

Use MockMvc or RestAssured for automated testing. Example with MockMvc:

**java**

**A computer code with black text

Description automatically generated**

**What is the difference between synchronous and asynchronous REST APIs?**

**Synchronous**: The client waits for the server to respond before proceeding.

**Asynchronous**: The client can continue other tasks while the response is processed later.

**Example of asynchronous REST API in Spring Boot:**

**A computer code with black text

Description automatically generated with medium confidence**

**How do you ensure REST API performance?**

Use caching (e.g., @Cacheable in Spring).

Optimize database queries.

Use pagination for large datasets.

Compress responses (e.g., GZIP).

**What are REST API best practices?**

Use meaningful URIs (/users instead of /getAllUsers).

Use HTTP status codes correctly (200 OK, 404 Not Found, 401 Unauthorized).

Document APIs using Swagger/OpenAPI.

**What is OpenAPI/Swagger?**

A tool for designing, documenting, and consuming REST APIs. It automatically generates documentation from code using annotations like @ApiOperation.

**Status Codes:**

**200 OK –** Successful GET, PUT, PATCH, DELETE requests.

**201 Created –** Successful resource creation via POST.

**204 No Content –** Successful DELETE or update without a response body.

**400 Bad Request** – Malformed requests or invalid input.

**401 Unauthorized –** Missing or invalid authentication token.

**403 Forbidden** – Insufficient permissions**.**

**404 Not Found –** Resource does not exist.

**409 Conflict –** Resource state conflicts (e.g., duplicate data).

**500 Internal Server Error –** Unexpected backend error.

| **Status** | **Controller Called?** | **Why** |
| --- | --- | --- |
| 200 | ✔ Yes | Successful request |
| 201 | ✔ Yes | Resource created |
| 204 | ✔ Yes | Success, no body |
| 400 | ❌ Sometimes | Validation fails before controller |
| 401 | ❌ No | Security check fails |
| 403 | ❌ No | Authorization fails |
| 404 | ❌ No | No controller mapping found |
| 409 | ✔ Yes | Business conflict inside controller |
| 500 | ✔ / ❌ | Depends where the error occurs |

**Java 8 Features**

🡪 Java 8 was designed to improve developer productivity, code readability, and performance.

**What features do you know or use in Java 8?**

Here you can list down all the key features of Java 8 like,

* Functional Interface
* Lambda Expression
* Stream API
* Completable Future
* Java DateTime API
* Method Reference
* Comparable and Comparator
* Optional Class

**Functional Interface**

A screenshot of a computer

Description automatically generated

**What is Functional Interface in Java 8?**

A **functional interface** in Java is an interface that contains exactly **one abstract method** but can have multiple **default** and **static** methods. It is primarily used to enable **lambda expressions**.

**TYPES OF FUNCTIONAL INTERFACES:**

**1.Consumer**

**2. Supplier**

**3.Function**

**4.Predicate**

**All are present in java.util.function; package.**

**1.Consumer:**

**🡪** It represents an operation, that accept a single input parameter and returns no result.

A computer screen shot of a code

AI-generated content may be incorrect.

**A computer screen shot of a computer code

AI-generated content may be incorrect.**

**A computer code with text

AI-generated content may be incorrect.**

**2.Supplier:**

🡪 It represents the supplier of the result. Accept no input parameter but produce a result.

A computer screen shot of a computer code

AI-generated content may be incorrect.

A computer code with black text

AI-generated content may be incorrect.

**3.Function**

**🡪** Represents function, that accept one argument process and produces results.

A close-up of a computer screen

AI-generated content may be incorrect.

A computer code with text

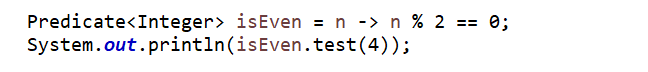
AI-generated content may be incorrect.

**4.Predicate**

**🡪** It represents function, that accept one argument and return the Boolean.

**A white background with black text

AI-generated content may be incorrect.**

****

**Lambda Expression**

**What is Lambda Expression?**

A lambda expression in Java is a short block of code that acts like a method, but without need of a name.

It eliminates boilerplate code when implementing functional interfaces.

A screenshot of a computer

AI-generated content may be incorrect.

**Can we extend a functional interface from another functional interface?**

Yes, we can extend but if you extend that your functional interface will not act as a functional interface because it will find multiple abstract methods inside that.

**What are the advantages of Lambda Expressio n?**

* + Less boilerplate code
  + Improved readability
  + Better Performance

**What are the disadvantages of Lambda expressions?**

* Hard to use without an IDE
* Complex to debug

**How are functional interfaces and Lambda Expressions related?**

Functional interfaces in Java are interfaces that only contain one abstract method.

* Lambda expressions provide a simple way to implement functional interfaces.
* Lambda expressions can be used wherever functional interfaces are needed.

// Functional interface  
interface MyFunctionalInterface {  
 void myMethod();  
}  
  
public class Main {  
 public static void main(String[] args) {  
 // Lambda expression for implemention of the functional interface  
 MyFunctionalInterface myLambda = () -> System.out.println("Hello Lambda!");  
   
 // calling method, using lambda expression  
 myLambda.myMethod();  
 }  
}

**How do you define a custom functional interface and use it with a lambda expression?**

**A computer code with text

AI-generated content may be incorrect.**

**Can lambda expressions access local variables?**→ Yes, but only effectively final variables (cannot be modified after assignment).

**Can lambda expressions access instance and static variables?**→ Yes, unlike local variables, they can modify instance and static variables.

**Stream API**

**What is Stream API in Java 8?**

* Stream API is used to process collections of objects.
* It will improve program's performance by allowing you to avoid unnecessary loops and iterations.
* Streams can be used for filtering, collecting, printing, and converting from one data structure to another, etc.

**What are the stream methods you used in your project?**

* filter
* forEach
* sorted
* map
* flatMap
* reduce
* groupingBy
* collect

**How is Stream different from Collections?**  
→ Collections store data and allow modifications, while Streams process data and are **immutable**.

**How do you create a Stream in Java?**

**A close-up of a computer code

AI-generated content may be incorrect.**

**Can a Stream be reused?**  
→ No, a Stream cannot be reused after a terminal operation.

**What is map () in Stream API?**→ Transforms each element in a stream.



String::toUpperCase is shortcut for s -> s.toUpperCase()

**System.out::println means:**

It takes each element from the stream or list and prints it.

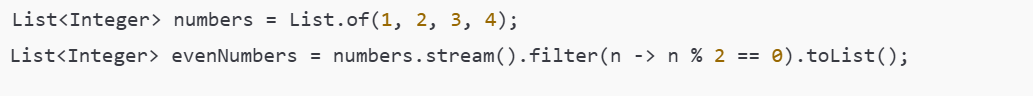
List<Integer> l=List.*of*(1,2,3);

List<Integer> l2=Arrays.*asList*(1,2,3);

List.of() → fully unmodifiable, no nulls

Arrays.asList() → fixed-size but elements modifiable

**What is filter () in Stream API?**→ Filters elements based on a condition.

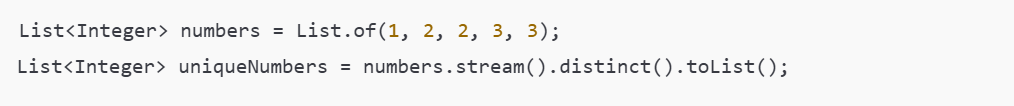


**What is flatMap () in Java Streams?**  
→ Flattens multiple streams into a single stream.

A close up of text

AI-generated content may be incorrect.

**What is distinct () in Java Streams?**  
→ Removes duplicate elements.



**What does sorted () do in Stream API?**  
→ Sorts elements naturally or using a comparator.





A **Comparator** is an interface in Java used to define **custom sorting logic** for objects.

**What is collect () in Java Streams?**  
→ Collects stream elements into a collection.

**Convert to List**

List<Integer> list = stream.collect(Collectors.toList());

**2️⃣ Convert to Set**

Set<Integer> set = stream.collect(Collectors.toSet());

**3️⃣ Convert to Map**

Map<Integer, String> map =

employees.stream()

.collect(Collectors.toMap(

Employee::getId,

Employee::getName

));

**4️⃣ Joining Strings**

String result = names.stream()

.collect(Collectors.joining(", "));

Output:

"Sridhar, Raju, Mani"

**5️⃣ Grouping**

Map<String, List<Employee>> map =

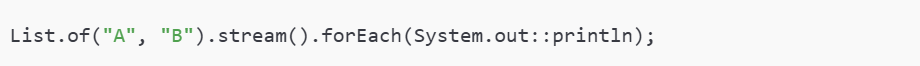
employees.stream()

.collect(Collectors.groupingBy(Employee::getDepartment));

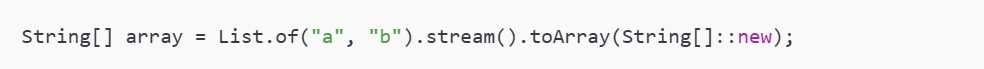
**6️⃣ Counting**

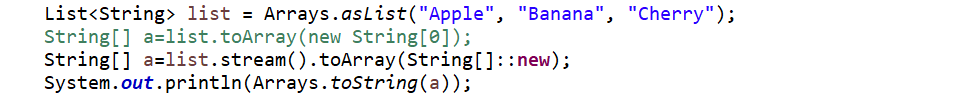
long count = stream.collect(Collectors.counting());

**What does forEach() do?**  
→ Iterates over each element.



**How do you convert a Stream to an array?**

****

****

**How do you convert an IntStream to a List?**

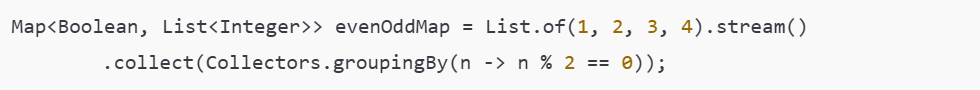
****

IntStream contains **primitive int**

boxed() converts int → Integer

collect(toList()) builds a List

**What is groupingBy() in Stream API**?  
→ Groups elements based on a classifier function.

****

**What is partitioningBy() in Java Streams?**→ Similar to groupingBy() but creates only two partitions (true and false).

**How do you find the max or min element using Stream?**

****

**How do you remove duplicates from a List using Stream API?**

****

**How do you limit the number of elements in a Stream?**→ Use limit(n).

****

**What is reduce () in Stream API?**→ Performs aggregation like sum, min, max.

****

**When should you not use Streams?**

* When modifying a collection (e.g., adding/removing elements).
* When the operation is stateful (e.g., keeping track of previous values).
* When debugging is required (since streams make debugging harder).

**Completable Future**

* Runs tasks asynchronously (non-blocking execution).
* Performs dependent tasks (chain multiple async operations).
* Handles errors (exception handling in async tasks).
* Combines multiple futures (run multiple tasks in parallel and combine results)

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screen shot of a computer code

AI-generated content may be incorrect.**

**Java DateTime API**

Java 8 introduced a new Date-Time API (java.time package) to solve the problems of the old java.util.Date and java.util.Calendar, which were mutable, error-prone, and not thread-safe.

A screenshot of a computer

AI-generated content may be incorrect.

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer code

AI-generated content may be incorrect.**

**A computer screen shot of a code

AI-generated content may be incorrect.**

**Method Reference**

A **method reference** is a shorthand syntax in Java 8 that allows you to refer to a method **without calling it**.. It improves code readability and reusability.

**More concise** than lambda expressions

**Less boilerplate**, removing unnecessary parameters

**Comparable and Comparator**

**Differentiate Between Comparable and Comparator in Java.**

Java provides two interfaces for configuring objects using class data members:

* Comparable
* Comparator

A screenshot of a computer

AI-generated content may be incorrect.

A computer code on a white background

AI-generated content may be incorrect.

**When to use map and flatMap?**

map (): It is used where we have to map the elements of a particular collection to a specific function, and then we need to return the stream that contains the updated results.

Example: Multiply all the elements of a list by 3 and return the updated list.

flatMap(): It is used where we have to transform or flatten the string, as we can't flatten our string using map().

Example: Get the first Character of all the String present in a List of Strings and return the result in form of a stream.

A computer code with black text

AI-generated content may be incorrect.

A screen shot of a computer code

AI-generated content may be incorrect.

**What is Optional Class in Java 8?**

In Java 8, Optional Class is a container object.

* The Optional class used to represent a value that may be present or may not be.
* This class helps in avoiding null pointer exceptions by providing methods to check the presence of a value before accessing it.
* This helps null values handling more effectively.

**Example:**

import java.util.Optional;

class Address {

private String email;

public Address(String email) {

this.email = email;

}

public String getEmail() {

return email;

}

}

class User {

private String name;

private Address address; // can be null

public User(String name, Address address) {

this.name = name;

this.address = address;

}

public Optional<Address> getAddress() {

return Optional.ofNullable(address);

}

public String getName() {

return name;

}

}

public class OptionalDemo {

public static void main(String[] args) {

// 1. Optional.of()

Optional<String> opt1 = Optional.of("Java");

System.out.println("of(): " + opt1.get());

// 2. Optional.ofNullable()

Optional<String> opt2 = Optional.ofNullable(null);

System.out.println("ofNullable(): " + opt2.orElse("Value is null"));

// 3. Optional.empty()

Optional<String> opt3 = Optional.empty();

System.out.println("empty(): " + opt3.orElse("Empty Optional"));

// 4. ifPresent()

opt1.ifPresent(v -> System.out.println("ifPresent(): " + v));

// 5. map()

Optional<Integer> length =

opt1.map(String::length);

System.out.println("map(): Length = " + length.get());

// 6. orElse()

String value1 = opt2.orElse("Default value");

System.out.println("orElse(): " + value1);

// 7. orElseGet()

String value2 = opt2.orElseGet(() -> "Generated Default");

System.out.println("orElseGet(): " + value2);

// 8. orElseThrow()

try {

opt3.orElseThrow(() -> new RuntimeException("Value is missing!"));

} catch (Exception e) {

System.out.println("orElseThrow(): " + e.getMessage());

}

// 9. Optional Chaining (User → Address → Email)

User user1 = new User("Ravi", new Address("ravi@example.com"));

User user2 = new User("John", null); // no address

String email1 = Optional.ofNullable(user1)

.flatMap(User::getAddress)

.map(Address::getEmail)

.orElse("Email not found");

System.out.println("Chaining (user1): " + email1);

String email2 = Optional.ofNullable(user2)

.flatMap(User::getAddress)

.map(Address::getEmail)

.orElse("Email not found");

System.out.println("Chaining (user2): " + email2);

// 10. Using Optional to avoid NullPointerException

Optional<String> safeNull = Optional.ofNullable(null);

safeNull.ifPresent(v -> System.out.println("Will not print this"));

}

}

**What is Date-Time API in Java 8?**

The Date-Time API in Java 8 provides a set of classes for date-time conversions, including timelines and advanced programming.

* It imports the **java.time** package, and this package contains **LocalDate, LocalTime, LocalDateTime, ZonedDateTime,** and other classes.
* This API provides better robustness, consistency and thread safety compared to legacy Date and Calendar classes.

**What is Optional equals() method in Java?**

In Java, the **equals()** method of the Optional class is used to compare two Optional objects for equality.

* It returns true if both the Optional objects contain the same value.
* And it returns false if both does not contain the same value.

**Illustration**:  
public class Main   
{  
 public static void main(String args[])   
{  
 // Creating Optional objects  
 Optional<String> opt1 = Optional.of("Sweta");  
 Optional<String> opt2 = Optional.of("Sweta");  
 Optional<String> opt3 = Optional.of("Dash");  
  
 // Comparing Optional objects  
 System.out.println(opt1.equals(opt2)); // true  
 System.out.println(opt1.equals(opt3)); // false  
 }  
}

**What is ArrayList forEach() method in Java?**

In Java, the forEach() method is used to iterate over each ArrayList element.

* It performs specified operations for each element.
* It simplifies iteration and shortens the code.
* It takes a Consumer as a parameter, which represents the action to be performed on each element.

ArrayList<Integer> numbers = new ArrayList<>();  
numbers.add(1);  
numbers.add(2);  
numbers.add(3);  
  
numbers.forEach(num -> System.out.println(num));

**How to find duplicate elements in a Stream in Java?**

**Count occurrence of a given character in a string using Stream API in Java.**

**How to get Slice of a Stream in Java?**

**How to Reverse elements of a Parallel Stream in Java?**

**Write a Program to Iterate over a Stream with Indices in Java 8.**

**What is method reference in Java 8?**

Method reference is a concise way to use a lambda expression for calling a method directly. It simplifies the code by providing a shorthand notation. are four types of method references that are listed below:

* Static Method Reference
* Instance Method Reference of a particular object
* Referencing an instance method of an unspecified object belonging to a specific class.
* Constructor Reference.

**Example:**

numList.stream().filter(n -> n > 5).sorted().forEach(System.out::println);

**What is MetaSpace in Java 8?**

* **Metaspace** is the memory area in Java 8+ used to store **class metadata** (information about classes, methods, fields, bytecode, etc.)

**Hibernate**

Use spring-data-jpa dependency

Hibernate is an ORM (Object-Relational Mapping) framework for Java that simplifies database interactions

**Why is Hibernate better than JDBC?**

* Reduces boilerplate code by mapping Java objects to database tables.
* Database-independent; no need to write database-specific SQL.
* Supports caching for better performance.

**What is a Session in Hibernate?**

* A Session is a short-lived object used to interact with the database.
* It is not thread-safe and is used to perform CRUD operations.
* Operations are executed when the transaction is committed or flushed.

**List and describe the Hibernate framework’s essential interfaces.**

1. **Session:** Handles database operations.
2. **SessionFactory:** Creates Session objects; is thread safe.
3. **Transaction:** Manages transactions.
4. **Query:** Executes HQL/SQL queries.

**What is the Hibernate Configuration File?**

* It specifies database connection properties, dialect, entity mappings, and caching settings.
* Example: hibernate.cfg.xml or hibernate.properties.

**What is an Entity in Hibernate?**

* A class representing a database table.
* Annotated with @Entity and fields are mapped to table columns.
* Example:

java

@Entity

public class Employee {

@Id

private int id;

private String name;

}

**What is ORM (Object-Relational Mapping)?**

* Maps Java objects to database tables.
* Makes database operations more object-oriented and simpler.

**Difference Between Session and SessionFactory**

| **Session** | **SessionFactory** |
| --- | --- |
| Short-lived | Long-lived |
| Not thread-safe | Thread-safe |
| Performs CRUD | Creates Session objects |

**What is HQL?**

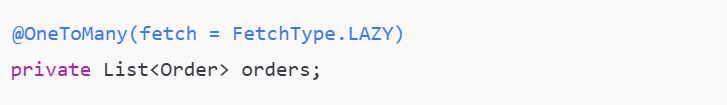
* Hibernate Query Language is an object-oriented query language.
* Works with entity attributes instead of table columns.
* Example: from Employee where department = 'HR'.

**Difference Between get() and load()**

| **get()** | **load()** |
| --- | --- |
| Fetches immediately | Returns a proxy, fetches lazily. |
| Returns null if not found | Throws exception if not found. |

**What is Lazy Loading?**

* Data is fetched only when accessed.
* Saves memory and improves performance.
* Example: @OneToMany(fetch = FetchType.LAZY)



**What is the purpose of application.properties in a Spring Boot application?**

It is used to configure application settings, including Hibernate, database connections, logging, and more.

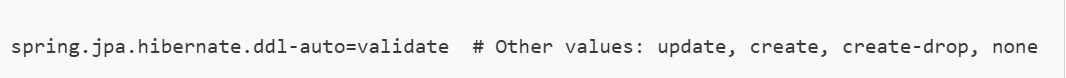
**How do you configure Hibernate dialect in application.properties?**

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

**How do you enable Hibernate’s SQL logging?**

spring.jpa.show-sql=true

**How do you configure Hibernate to validate or update the schema?**

****

**How do you configure the database connection pool in application.properties?**

**A screenshot of a computer code

Description automatically generated**

**How do you configure Hibernate batch size in application.properties?**

****

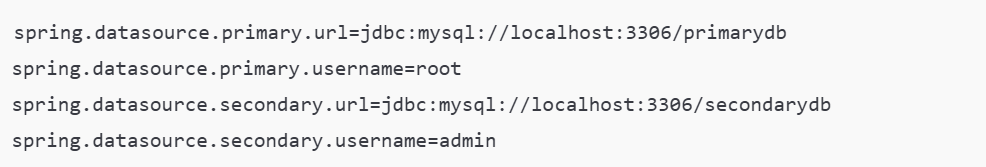
**How do you configure the connection timeout for the database?**

****

**What is the difference between spring.datasource and spring.jpa properties?**

* spring.datasource: Configures database connection details like URL, username, password, and driver.
* spring.jpa: Configures Hibernate-specific properties like SQL logging, dialect, and schema management.

**How do you configure multiple data sources in application.properties?**

****

**What are the advantages of Hibernate over other ORM tools?**

* **Database Independent:** Works with multiple databases using a dialect.
* **Reduces Boilerplate Code:** Simplifies persistence with object-oriented methods.
* **Caching Support:** Improves performance with first- and second-level caching.
* **HQL Support:** Object-oriented query language makes queries simpler.
* **Lazy Loading:** Loads data only when needed, saving resources.
* **Automatic Schema Generation:** Creates and manages tables automatically.

**What are the different states of an object in Hibernate?**

1. **Transient:** Object exists only in memory and is not associated with a database or Session.
2. **Persistent:** Object is associated with a Session and is synchronized with the database.
3. **Detached:** Object was once persistent but is now disconnected from the Session.

**What is the difference between save () and persist ()?**

| **save ()** |  | **persist ()** |
| --- | --- | --- |
| Returns the generated ID. |  | Does not return the ID. |
| Can be used outside a transaction. |  | Must be used within a transaction. |
| Insert the object immediately. |  | Defers insert until flush/commit. |

**What is the difference between merge () and update ()?**

Hibernate provides two methods to reattach a **detached entity**:  
**update()** and **merge()**, but they work very differently.

| **merge ()** | **update ()** |
| --- | --- |
| Does not throw an exception if the object is already in the Session. | Throws an exception if the object is already in the Session. |
| Returns a new persistent instance. | Does not return a new instance. |

**What is the difference between first-level cache and second-level cache in Hibernate?**

| **First-Level Cache** | **Second-Level Cache** |
| --- | --- |
| Enabled by default. | Must be explicitly enabled. |
| Specific to a Session. | Shared across Sessions. |
| Exists for the duration of a Session. | Exists for the duration of the SessionFactory. |

**What are the annotations used in Hibernate?**

1. **@Entity:** Marks a class as a database entity.
2. **@Table:** Specifies the table name.
3. **@Id:** Marks the primary key field.
4. **@GeneratedValue:** Configures how the primary key is generated.
5. **@Column:** Maps a field to a table column.
6. **@OneToOne, @OneToMany, @ManyToOne, @ManyToMany:** Define relationships.
7. **@JoinColumn:** Specifies the join column for relationships.
8. **@Transient:** Excludes a field from persistence.

**How does Hibernate handle database transactions?**

* Hibernate uses **ACID transactions** to ensure data consistency.
* Transactions are managed by the Transaction interface in Hibernate.
* Operations like save(), update(), and delete() are performed within a transaction.
* Commit (transaction.commit()) saves changes; rollback (transaction.rollback()) undoes changes.
* In Spring Boot, transactions are often managed with @Transactional.

**How would you handle versioning in Hibernate?**

* Versioning is used to manage concurrent updates to data.
* Add a @Version annotation to a field (e.g., int, long, or Timestamp) in your entity.
* Hibernate increments the version number with each update to detect conflicts.

A screen shot of a computer code

Description automatically generated

**How do you use composite keys in Hibernate?**

* Composite keys are created using multiple fields as a primary key.
* Use the @IdClass or @EmbeddedId annotations.

**A computer screen shot of a code

Description automatically generated**

**Example with @EmbeddedId:**

A screenshot of a computer code

Description automatically generated

**Options:**

1. **validate:** Validates schema but makes no changes.
2. **update:** Updates the schema without dropping existing data.
3. **create:** Drops and recreates the schema every time.
4. **create-drop:** Drops the schema at the end of the session.
5. **none:** Disables schema management.

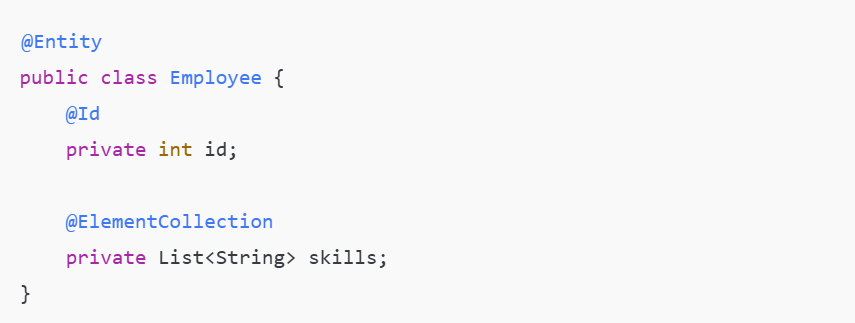
**Example:**

spring.jpa.hibernate.ddl-auto=update

**How do you map a collection of elements in Hibernate?**

* Use @ElementCollection for a collection of simple types or embeddable objects.
* Use @OneToMany or @ManyToMany for relationships with other entities.

**Example with @ElementCollection:**



**Example with @OneToMany:**

A computer code on a white background

Description automatically generated

**What are the common properties in the Hibernate configuration file?**

1. **Database connection settings:**

hibernate.connection.url=jdbc:mysql://localhost:3306/mydb

hibernate.connection.username=root

hibernate.connection.password=password

hibernate.connection.driver\_class=com.mysql.cj.jdbc.Driver

1. **Hibernate dialect:**

hibernate.dialect=org.hibernate.dialect.MySQLDialect

1. **Schema management:**

hibernate.hbm2ddl.auto=update

1. **Caching settings:**

hibernate.cache.use\_second\_level\_cache=true

hibernate.cache.provider\_class=org.hibernate.cache.EhCacheProvider

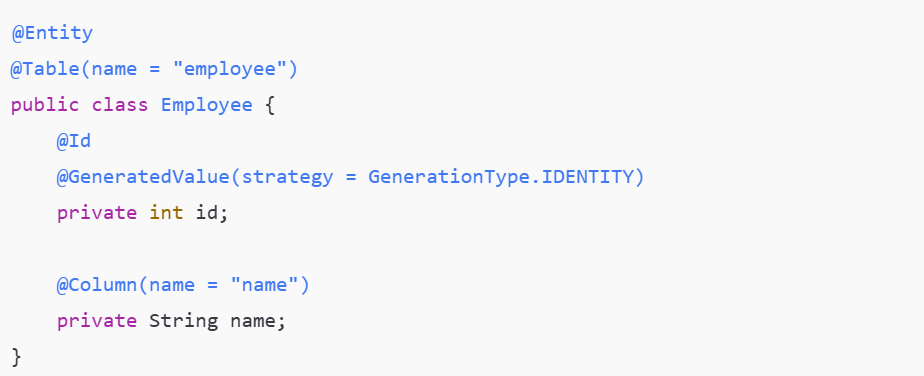
1. **SQL logging:**

hibernate.show\_sql=true

hibernate.format\_sql=true

**How do you configure Hibernate with annotations?**

* Add Hibernate annotations directly to your entity classes:



**What is the difference between native SQL and HQL in Hibernate?**

| **Native SQL** | **HQL** |
| --- | --- |
| Uses plain SQL queries. | Object-oriented query language. |
| Works with database tables. | Works with entity objects. |
| Database-specific. | Database-independent. |

**What are proxies in Hibernate?**

* A proxy is a placeholder object created by Hibernate to support **lazy loading**.
* The proxy initializes the actual object when needed, reducing memory and performance costs.

**Explain the concept of Interceptor and Event Listeners in Hibernate.**

* **Interceptor:** Allows custom logic to be executed during Hibernate operations like save, delete, or update. Implement Interceptor interface to define custom behavior.
* **Event Listener:** A more advanced feature that hooks into Hibernate's lifecycle events, like pre-insert or post-update.

**Example of Interceptor:**

public class MyInterceptor extends EmptyInterceptor {

@Override

public boolean onSave(Object entity, Serializable id, Object[] state, String[] propertyNames, Type[] types) {

System.out.println("Entity saved: " + entity);

return false;

}

}

**2. What is Hibernate and how does it relate to JPA?**

* **Hibernate** = ORM Framework.
* **JPA** = Specification (interface).
* Hibernate is a **JPA implementation**.

**JPA → Specification**

“JPA is a *standard/specification* that defines how ORM should work in Java.”

**Hibernate → Implementation**

“Hibernate is an *ORM framework* that implements JPA and performs actual database operations.”

**Spring Data JPA → Abstraction Layer**

“Spring Data JPA is a *framework on top of JPA* that reduces boilerplate and generates repository code automatically.”

**Simple Analogy**

| **Concept** | **Analogy** |
| --- | --- |
| **JPA** | Rulebook |
| **Hibernate** | The player who follows the rulebook |
| **Spring Data JPA** | Coach who makes your work easier and automates tasks |

**What is @GeneratedValue and its strategies?**

@GeneratedValue is a JPA annotation used to **auto-generate primary key values** instead of manually assigning them

@GeneratedValue(strategy = GenerationType.IDENTITY)

Strategies:

 **AUTO:** Hibernate chooses the best strategy.

 **IDENTITY:** Uses DB auto-increment columns (like MySQL).

 **SEQUENCE:** Uses database sequences (Oracle/PostgreSQL).

 **TABLE:** Uses a table to generate IDs.

**What is @Transient used for?**

Fields ignored by JPA.

@Transient is a JPA annotation used to **mark a field that should NOT be persisted** to the database.

@Transient

private int temp;

**How do you map enums in JPA?**

In JPA, you map enums using the @Enumerated annotation. It tells JPA how to store an enum value in the database

**EnumType.ORDINAL (stores numbers)**

Stores the **enum index (0, 1, 2…)** in the database.

public enum Status {

ACTIVE,

INACTIVE,

PENDING

}

@Entity

public class User {

@Enumerated(EnumType.ORDINAL)

private Status status;

}

**✔ Pros**

* Very compact, small DB size.

**❌ Cons (Major)**

* **Dangerous in real projects.**  
  If you change the order of enum values or add new ones, all existing data becomes invalid.

**Never recommended for production.**

**EnumType.STRING (stores text)**

Stores the **enum name ("ACTIVE", "INACTIVE", "PENDING")** in the database.

@Entity

public class User {

@Enumerated(EnumType.STRING)

private Status status;

}

**✔ Pros**

* Safe and readable
* Adding new enum values does **not** break existing rows
* Best for real-world apps

**❌ Cons**

* Slightly larger column storage size (negligible)

**This is the recommended strategy.**

**🟩 ASSOCIATIONS & RELATIONSHIPS**

**✅ 11. Association types**

| **Annotation** | **Meaning** |
| --- | --- |
| @OneToOne | One object ↔ One object |
| @OneToMany | One ↔ Many |
| @ManyToOne | Many ↔ One |
| @ManyToMany | Many ↔ Many |

**What is owning vs inverse side?**

**How to Choose the Owning Side in JPA**

Choosing the owning side is important because **Hibernate only saves relationship changes from the owning side**.

Below are simple rules for each relationship type.

**Many-to-One (MOST COMMON)**

**✔ The Many side is ALWAYS the owning side**

This is because the **foreign key is always on the Many side**.

Example:

* One Customer → Many Orders
* FK customer\_id is in Order table
* So **Order is the owning side**

👉 **Rule:**  
@ManyToOne is the owning side  
@OneToMany(mappedBy = ...) is the inverse side

This is the easiest and most common rule.

**One-to-One**

This depends on where the **foreign key** is placed.

**✔ If FK is in table A → A is the owning side**

**✔ If FK is in table B → B is the owning side**

Example:

@Entity

class User {

@OneToOne

@JoinColumn(name = "profile\_id") // FK is here → owning side

private Profile profile;

}

Profile side becomes inverse:

@OneToOne(mappedBy = "profile")

private User user;

👉 **Rule:**  
Owning side = **side with @JoinColumn**  
Inverse side = side with mappedBy

**Many-to-Many**

Here, JPA creates a **join table**.

**✔ You may choose either side as owning side**

But you must choose **only one**, NOT both.

**Best practice:**

* Choose the entity that is **more frequently used** to maintain relationships.
* Example: Student–Course  
  Usually *Student* owns, Course is inverse.

Owning side:

@ManyToMany

@JoinTable(

name = "student\_course",

joinColumns = @JoinColumn(name = "student\_id"),

inverseJoinColumns = @JoinColumn(name = "course\_id")

)

private List<Course> courses;

Inverse side:

@ManyToMany(mappedBy = "courses")

private List<Student> students;

👉 **Rule:**  
One side defines @JoinTable, the other uses mappedBy.

**What is mappedBy and when used?**

mappedBy defines the **inverse** side.

@OneToMany(mappedBy="user")

private List<Order> orders;

**What is cascade?**

**Cascade** means if you perform an operation on one entity (parent), the same operation is automatically performed on its related entities (children).

Example:

If you delete a customer and want to automatically delete all their Orders, you enable cascade.

Cascade lets operations propagate automatically.

cascade = CascadeType.ALL

Types:

* ALL
* PERSIST
* MERGE
* REMOVE
* REFRESH
* DETACH

Use when child lifecycle depends on parent.

**orphanRemoval vs cascade delete**

* **orphanRemoval** deletes child when removed from collection.
* **cascade delete** deletes child when parent is deleted.

**CascadeType.REMOVE (Cascade Delete)**

**When you delete the parent → Hibernate deletes the children.**

**Example:**

@OneToMany(cascade = CascadeType.REMOVE)

private List<Order> orders;

entityManager.remove(customer);

All orders of that customer are deleted.

**✔ Purpose**

Propagate **parent deletion** to children.

**✔ When to use**

When child **cannot exist without parent** (composition).

**BUT:**

CascadeType.REMOVE **does NOT delete a child** when it is simply removed from the list.

**orphanRemoval = true**

This removes children that become **orphan objects** — meaning:

If a child is removed from the relationship **in memory**, it is deleted from the database automatically.

**Example:**

@OneToMany(mappedBy = "customer", orphanRemoval = true)

private List<Order> orders;

Now:

customer.getOrders().remove(order);

👉 Hibernate will DELETE that single order row from DB  
(even if the parent is not deleted)

**✔ Purpose**

Delete children that no longer have a parent in the relationship.

**✔ When to use**

For relationships where child **must NOT exist without being linked**.

Example:

* OrderItems inside an Order
* Addresses inside a User profile
* CartItems inside a Cart

**🧪 Example to show difference clearly**

customer.getOrders().remove(order);

* With **cascade = REMOVE** → ❌ Nothing happens
* With **orphanRemoval = true** → ✅ Order will be deleted

entityManager.remove(customer);

* With **cascade = REMOVE** → ✅ Orders deleted
* With **orphanRemoval = true** → ❌ Does not automatically cascade unless REMOVE is also used  
  (though usually both are used together)

**16. How to map join table in many-to-many?**

@ManyToMany

@JoinTable(...)

**Unidirectional Relationship**

**✔ Only one entity knows about the relationship**

Meaning:  
You can navigate from **A → B**, but **NOT from B → A**.

**Example: Unidirectional Many-to-One**

@Entity

class Order {

@ManyToOne

private Customer customer; // navigate only Order → Customer

}

Here:

* From Order you can find the Customer
* But in Customer there is **no list of orders**

**✔ Simpler mapping**

**✔ Less code**

**❌ Cannot navigate backwards (Customer → Orders)**

**Bidirectional Relationship**

**✔ Both entities know each other**

You can navigate **A → B** and **B → A**.

**Example: Bidirectional One-to-Many / Many-to-One**

@Entity

class Order {

@ManyToOne

@JoinColumn(name = "customer\_id")

private Customer customer; // owning side

}

@Entity

class Customer {

@OneToMany(mappedBy = "customer")

private List<Order> orders; // inverse side

}

Now:

* From Order → Customer
* From Customer → Orders

**✔ Convenient navigation both ways**

**✔ Needed for some business logic**

**❌ Must manage owning vs inverse side**

**❌ More chance of infinite JSON recursion (requires @JsonIgnore)**

**FETCHING & LOADING**

**EAGER vs LAZY**

* LAZY → loads when accessed
* EAGER → loads immediately

Prefer **LAZY** unless absolutely needed.

A screenshot of a computer

AI-generated content may be incorrect.

**N+1 Select Problem**

Cause:

The N+1 Select Problem happens when Hibernate runs **1 query to load the parent records** and then runs **another query for each parent** to load its child records.

Solution:

* fetch join
* EntityGraph
* BatchSize
* Second level cache

How to Fix N+1 Problem?

✔ 1. Use JOIN FETCH in JPQL / HQL

@Query("SELECT c FROM Customer c LEFT JOIN FETCH c.orders")

List<Customer> findAllWithOrders();

Now Hibernate makes only 1 query:

SELECT c.\*, o.\*

FROM customer c

LEFT JOIN order o ON o.customer\_id = c.id;

✔ 2. Use Entity Graph

@EntityGraph(attributePaths = "orders")

List<Customer> findAll();

✔ 3. Use Batch Fetching

@BatchSize(size = 20)

private List<Order> orders;

✔ 4. Change FetchType (rare)

Sometimes set OneToMany → EAGER (but not recommended generally)

**QUERIES**

**Parameterized JPQL**

query.setParameter("id", id);

Prevents SQL injection.

**createQuery() vs createNativeQuery()**

createQuery() is used for JPQL/HQL, which works with entity names and is database independent.

createNativeQuery() is used to run raw SQL, using table and column names directly and allows DB-specific features.

**🟥 TRANSACTIONS & SESSION**

**Hibernate Session vs EntityManager**

**EntityManager (JPA)**

* Part of **JPA specification**
* Database-agnostic (standard API)
* Used in modern Spring Boot apps
* Works on top of Hibernate (or any JPA provider)

entityManager.persist(entity);

**Session (Hibernate)**

* Hibernate’s **native API**
* Not part of JPA standard
* Has extra Hibernate-specific features not available in EntityManager

**flush()**

flush() forces Hibernate to **synchronize the in-memory changes with the database** immediately.

In simple words:

flush() sends all pending INSERTs, UPDATEs, and DELETEs to the database.

But it **does NOT commit** the transaction.

**persist() vs merge()**

persist() is used to save a **new** entity and makes it managed.  
merge() is used to update a **detached** entity; it returns a new managed instance while the original stays detached.

persist() = insert only  
merge() = insert or update depending on entity state.

**detach() & remove()**

detach() removes an entity from the persistence context, meaning Hibernate stops tracking it—no SQL runs.

remove() marks the entity for deletion; Hibernate executes a DELETE in the database on flush/commit.

**Optimistic vs Pessimistic Locking**

**Optimistic** → @Version  
**Pessimistic** → DB-level locks

**Optimistic locking** uses a version field and does not lock the database row.  
It detects conflicts only when the transaction commits.  
Best for low-contention, high-concurrency systems.

**Pessimistic locking** locks the row immediately using database locks (FOR UPDATE).  
Other transactions must wait or fail.  
Best for high-contention systems like banking.

**Hibernate Performance Tips**

* Use fetch join
* Use batch size
* Enable JDBC batching
* Avoid EAGER
* Use projections for heavy queries

**JDBC Batching**

Enable:

hibernate.jdbc.batch\_size=30

**🟫 ADVANCED**

**Handling LOBs**

Use:

* @Lob
* Stream-based access

**Multi-tenancy in Hibernate**

Supports:

* Separate DB
* Separate schema
* Partitioned tables

**Connection pooling**

Hibernate uses:

* HikariCP (default in Spring Boot)

**Soft delete**

Use:  
@SQLDelete + @Where(clause = "deleted=false")

**Hibernate + Spring**

@Transactional boundaries control session/transaction.

**Custom Types**

Use:

* AttributeConverter
* UserType

**Cascading detached entities**

Hibernate may reattach or throw exceptions depending on cascade.

**Bytecode enhancement**

Improves:

* Lazy loading
* Dirty checking

**Bulk update/delete consequences**

Bypass persistence context → must call clear() manually.

**Integration tests**

Use:

* Spring Boot Test
* @DataJpaTest
* H2 database

**Using H2**

Add dependency + configure datasource.

**Migration tools**

Use:

* **Flyway**
* **Liquibase**

**ANNOTATIONS**

**@SpringBootApplication**

This annotation is used to mark the main class of a Spring Boot application. It encapsulates @SpringBootConfiguration, @EnableAutoConfiguration, and @ComponentScan annotations with their default attributes.

**@EnableAutoConfiguration**

It automatically creates, and registers beans based on both the included jar files in the classpath and the beans defined by us.

**@ComponentScan**

@ComponentScan is an annotation used in the Spring Framework for auto-detecting and registering Spring-managed components (e.g. beans, controllers, services, repositories, etc.) within a specified package or set of packages.

**@Configuration**

The @Configuration annotation in Spring marks a class as a configuration class that provides bean definitions.

**Stereotype Annotations**

1. **Component**

without having to write any explicit code, Spring will:

Scan our application for classes annotated with @Component

Instantiate them and inject any specified dependencies into them.

1. Service
2. RestController/Controller
3. Repository

**Spring Core Annotations**

1. Configuration
2. Bean
3. Autowired
4. Qualifier
5. primary
6. Lazy
7. Value
8. PropertySource
9. ConfigurationProperties
10. Profile
11. Scope

**@Configuration:**

The @Configuration annotation in Spring marks a class as a configuration class that provides bean definitions.

**@Bean**

It indicates that a method produces a bean to be managed by the Spring Container. It is usually declared in Configuration class to create Spring bean definitions.

**@Qualifier**

The @Qualifier annotation is used to resolve the autowiring conflict, when there are multiple beans of same type.

**@Primary**

It indicates that a bean should be given preference when multiple beans are candidates to be autowired to a single-valued dependency.

**@Lazy**

It indicates that a bean should be lazily initialized, meaning it will only be created when it's first requested, rather than during the application context initialization.

**@Value**

@Value is a core annotation in Spring that is primarily used for assigning default values to variables and method parameters.

**@PropertySource**

It is used to provide properties file to Spring Environment.

**@ConfigurationProperties**

to map or bind the . properties or yml configuration values to Java objects. (with prefix)

**@Profile**

to resolve the challenge of controlling which parts of our application should be active under a particular set of conditions or environments.

**@Scope**

It indicates the lifecycle of an instance, such as singleton or prototype.

**REST API ANNOTATIONS**

1. RestController
2. Controller
3. RequestMapping
4. GetMapping
5. PostMapping
6. PutMapping
7. DeleteMapping
8. RequestBody
9. PathVariable
10. RequestParam
11. Controller Advice & ExceptionHandler

**@Controller**

@Controller is used to define a Spring MVC controller that processes HTTP requests and typically returns a view (like a JSP, Thymeleaf, or HTML page).

Methods in a @Controller class usually return the name of a view or a ModelAndView object. The returned view name is resolved by a view resolver to render the HTML page or template.

**@RestController**

@RestController is a specialized version of @Controller that is used for creating RESTful web services. It combines @Controller and @ResponseBody, meaning that the return values of methods are directly written to the HTTP response body rather than being resolved as a view name.

Methods in a @RestController class typically return objects, which are automatically serialized into JSON or XML (based on the content type requested) and sent in the HTTP response body.

**@RequestMapping**

This is an annotation used to map web requests to specific handler methods or classes. It can be applied at the class level or method level in a Spring controller.

**@ControllerAdvice**

It is used to define global exception handlers, model attribute handlers, and binding handlers that apply to all controllers or a subset of controllers in your application.

**@ExceptionHandler**

It is used to handle exceptions thrown by a specific controller method or across a set of controllers. It provides a way to define methods that should be called when a certain type of exception is thrown.

**JUnit**

**Unit testing** is a software testing technique where individual components or units of a software application are tested in isolation to ensure they work as expected.

**Junit:**

* Unit testing framework for java and it’s framework.
* Junit5 is the next generation of JUnit.
* The goal is to create an up-to-date foundation for developer-side testing on the JVM. This includes focusing on Java 8 and above, as well as enabling many different styles of testing.

**AssertJ:**

* AssertJ is a Java library that provides a rich set of assertions and truly helpful error messages.
* It improves test code readability and is designed to be super easy to use within your favorite IDE.

**Mockito**

* Mockito is mocking framework for Java
* It lets you write beautiful tests with a clean & simple API.
* The tests are very readable, and they produce clean verification errors.
* Mockito 3.x requires Java 8 or above.

**What is the difference between @Test and @Before annotations in JUnit?**

* @Test: Marks a method as a test method to be run by the JUnit framework.
* @Before: Marks a method to be executed before each test case. It's typically used to set up the test environment (e.g., initializing objects).

**What is the purpose of @After annotation in JUnit?**

* @After is used to define a method that runs after each test case. It is generally used for cleanup, like closing resources or resetting values.

**Explain the @BeforeClass and @AfterClass annotations.**

* @BeforeClass: This annotation is used to mark a method that should be executed once before any of the test methods in the class are run.
* @AfterClass: This annotation marks a method to be executed once after all test methods in the class have been run.

**What is a test suite in JUnit?**

* A test suite is a collection of test cases that can be run together. In JUnit, you can use @RunWith and @Suite.SuiteClasses to define a suite of tests.

**What are assertions in JUnit?**

Assertions are used to check the expected result against the actual result of the test case. Common assertions include:

* + assertEquals(expected, actual)
  + assertTrue(condition)
  + assertFalse(condition)
  + assertNotNull(object)
  + assertNull(object)

**How do you handle exceptions in JUnit tests?**

In JUnit, you can test for exceptions using the expected attribute of the @Test annotation or use try-catch blocks:

* + @Test(expected = Exception.class)
  + Alternatively, use assertThrows in JUnit 5 for more control.

**What is the difference between JUnit 4 and JUnit 5?**

* JUnit 4 uses annotations like @Test, @Before, and @After, while JUnit 5 introduces new annotations (@BeforeEach, @AfterEach, etc.) and supports Lambda expressions for more flexible tests.
* JUnit 5 also introduces the concept of extensions (similar to JUnit 4's Rules) and allows for better integration with modern IDEs and build tools.

**What is @ParameterizedTest in JUnit 5?**

* @ParameterizedTest allows you to run a test multiple times with different inputs. It is used when you want to test the same logic under various conditions or inputs.

**What is the use of @Disabled annotation?**

* The @Disabled annotation is used to disable a test method or class. The test will not run, and the framework will report it as skipped.

**How do you test private methods in JUnit?**

Private methods can be tested in two ways:

* + **Reflection**: You can use Java reflection to access and invoke private methods.
  + **Refactor to package-private or protected**: This makes it easier to test the method within the same package or subclass.

**What is mocking in JUnit?**

* Mocking is a technique used to simulate the behavior of real objects to test specific parts of a program in isolation. Frameworks like **Mockito** are often used in combination with JUnit for creating mock objects and verifying interactions with dependencies.

**What is a test double?**

* A test double is an object that takes the place of a real object in the context of a unit test. Types of test doubles include:
  + **Mock**: A mock object simulates the behavior of real objects and verifies interactions.
  + **Stub**: A stub is a simple implementation of an interface that returns predefined responses.
  + **Spy**: A spy records the calls made to it and can return values.

**What is the use of @Timeout in JUnit 5?**

* The @Timeout annotation is used to specify that a test should complete within a given time limit. If the test takes longer than the specified time, it will fail.

**Explain the difference between assertEquals() and assertSame() in JUnit.**

* + assertEquals(expected, actual): Checks if the expected value is equal to the actual value based on the equals() method.
  + assertSame(expected, actual): Checks if the expected and actual objects are the same (i.e., they refer to the same memory location).

**What is the role of @TestInstance in JUnit 5?**

* @TestInstance specifies the lifecycle of test instances. By default, JUnit 5 creates a new instance of the test class for each test method. Using @TestInstance(Lifecycle.PER\_CLASS) makes JUnit create only one instance of the test class for all test methods.

**How do you run JUnit tests from the command line?**

* You can run JUnit tests from the command line using build tools like **Maven** or **Gradle**:
  + **Maven**: mvn test
  + **Gradle**: gradle test

**What are assumptions in JUnit?**

* Assumptions are used to specify conditions that must be true for a test to execute. If an assumption fails, the test is ignored (skipped). For example, Assume.assumeTrue(condition) will skip the test if the condition is false.

**What is the @EnableJUnit4 annotation in JUnit 5?**

* @EnableJUnit4 is used to run JUnit 4 tests in a JUnit 5 environment, allowing backward compatibility with existing JUnit 4 tests.

**How can you test the performance of a method in JUnit?**

* You can test the performance of a method using @Test(timeout = <time\_in\_ms>) in JUnit 4, or by using a custom performance extension or tool in JUnit 5.

**==============================================================================**

**What is spring framework and its advantages?**

It is an open-source framework for building Java-based applications. It provides tools and features to simplify enterprise development.

**Key Features:**

1. **Dependency Injection (DI)** – Manages object creation and dependencies.
2. **Aspect-Oriented Programming (AOP)** – Separates cross-cutting concerns like logging and security from the business logic, making the code more modular and easier to maintain.
3. **Spring MVC** – A framework for building web applications.
4. **Spring Boot** – Simplifies app setup with pre-configured templates.
5. **Spring Security** – Handles authentication and authorization.
6. **Data Access** – Simplifies working with databases and integrates with technologies like JDBC, Hibernate, and JPA.

**Advantages:**

* **Loose Coupling** – Makes components independent and easier to maintain.
* **Testability** – Encourages writing testable code.
* **Comprehensive** – Offers built-in support for many tasks (data access, security, etc.).
* **Flexibility** – Works with various platforms and technologies.

**Where does custom exception fits in?**

Custom exceptions in a Spring application are used to handle specific error conditions that aren’t covered by standard exceptions. They help you manage application-specific errors and provide meaningful feedback**.**

**A screen shot of a computer

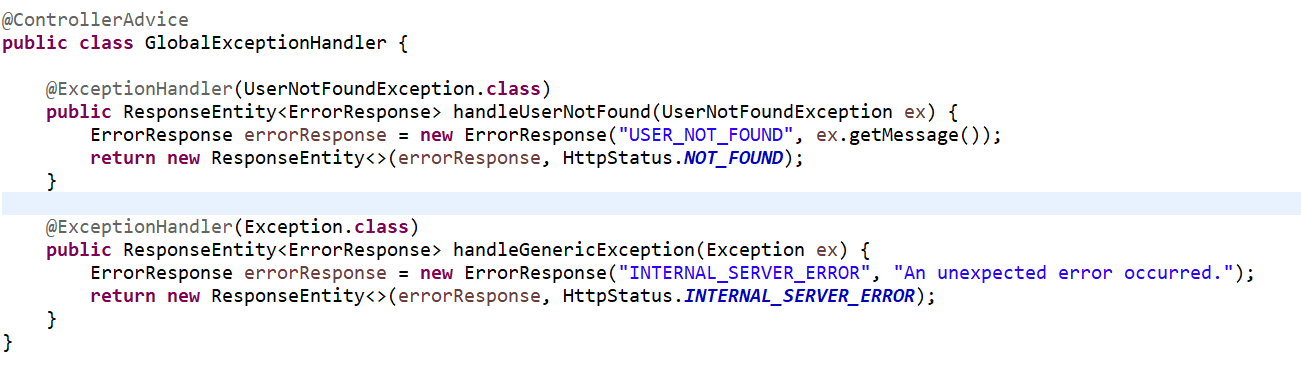
AI-generated content may be incorrect.**

**A close-up of a computer screen

AI-generated content may be incorrect.**

**Handling Custom Exceptions with @ControllerAdvice**

In a Spring application, you can use the @ControllerAdvice annotation to handle custom exceptions globally and return a meaningful response to the client.



**What is unit testing?**

**Unit testing** is a software testing technique where individual components or units of a software application are tested in isolation to ensure they work as expected.

**What is Purpose of RestController?**

The **@RestController** annotation in Spring is used to define a RESTful web service controller. It is a convenience annotation that combines two important annotations in Spring:

1. **@Controller**: Marks the class as a Spring MVC controller that can handle HTTP requests.
2. **@ResponseBody**: Indicates that the return values of methods in the class should be automatically converted to JSON or XML and sent as the HTTP response body, rather than being rendered as a view.

**How do you achieve Dependency Injection in Spring?**

**Dependency Injection (DI)** in Spring is a core concept that allows you to inject dependencies (such as services, repositories, or components) into a class, rather than creating them directly within the class.

Spring provides several ways to achieve dependency injection, but the most common methods are **Constructor Injection**, **Setter Injection**, and **Field Injection**.

1. **Constructor Injection**



2. **Setter Injection**:A screenshot of a computer program

AI-generated content may be incorrect.

3. **Field Injection**:

A screenshot of a computer program

AI-generated content may be incorrect.

**What is the purpose of SpringBootApplication?**

The purpose of a **Spring Boot application** is to provide a **quick, easy, and efficient way to create stand-alone, production-ready applications** with minimal configuration. It is part of the **Spring Framework** and simplifies the process of setting up and running Spring applications.

 **Quick Setup**: You can get started with minimal configuration.

 **Embedded Server**: It includes a built-in web server (like Tomcat), so no need for external setup.

 **Auto-Configuration**: Automatically configures your application based on the dependencies you add.

**How do you customize the serialization?**

use transient keyword

**What is the best strategy of sorting elements in ArrayList?**

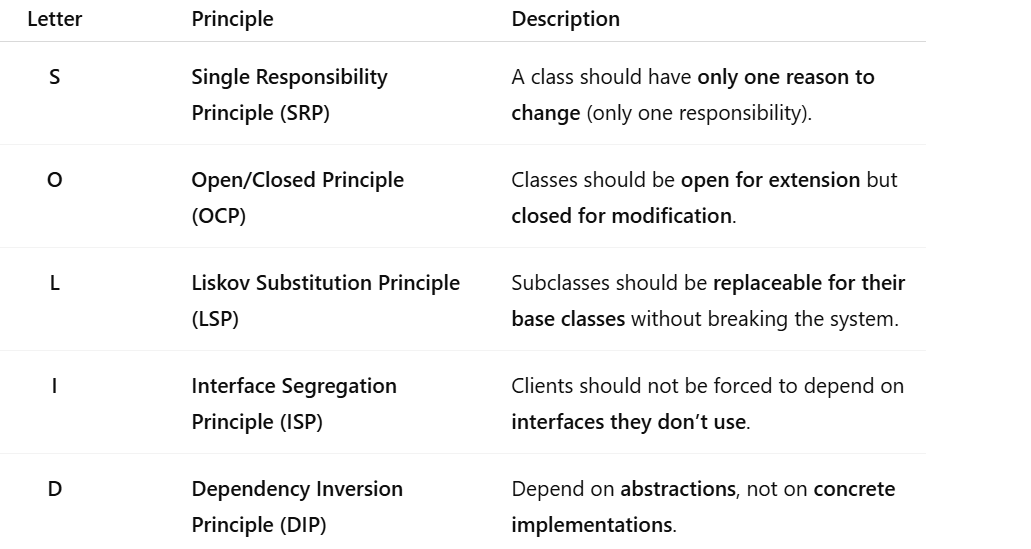
Collections.sort(list);

Collections.sort(list, Comparator.comparingInt(person -> person.age));

**what is RestControllerAdvice? And the purpose?**

@RestControllerAdvice is a specialized version of @ControllerAdvice in Spring, specifically used in RESTful web services. It is a convenience annotation that combines **@ControllerAdvice** and **@ResponseBody**. It allows you to handle exceptions globally and return appropriate responses, without needing to manually handle exceptions in every individual controller.

**What is SOLID in Java?**

****

1. **Single Responsibility Principle (SRP)**

A class should do **only one thing**.

**2.Open/Closed Principle (OCP)**

A screenshot of a computer code

AI-generated content may be incorrect.

A screenshot of a computer code

AI-generated content may be incorrect.

1. **Liskov Substitution Principle (LSP)**

**Bad Example**

**A close-up of a computer screen

AI-generated content may be incorrect.**

**A white background with text

AI-generated content may be incorrect.**

1. **Interface Segregation Principle (ISP)**

Don’t force a class to implement methods it doesn’t need.

**A screenshot of a computer program

AI-generated content may be incorrect.**

1. **Dependency Inversion Principle (DIP)**

Depend on abstractions, not concrete classes.

**A computer code with text

AI-generated content may be incorrect.**

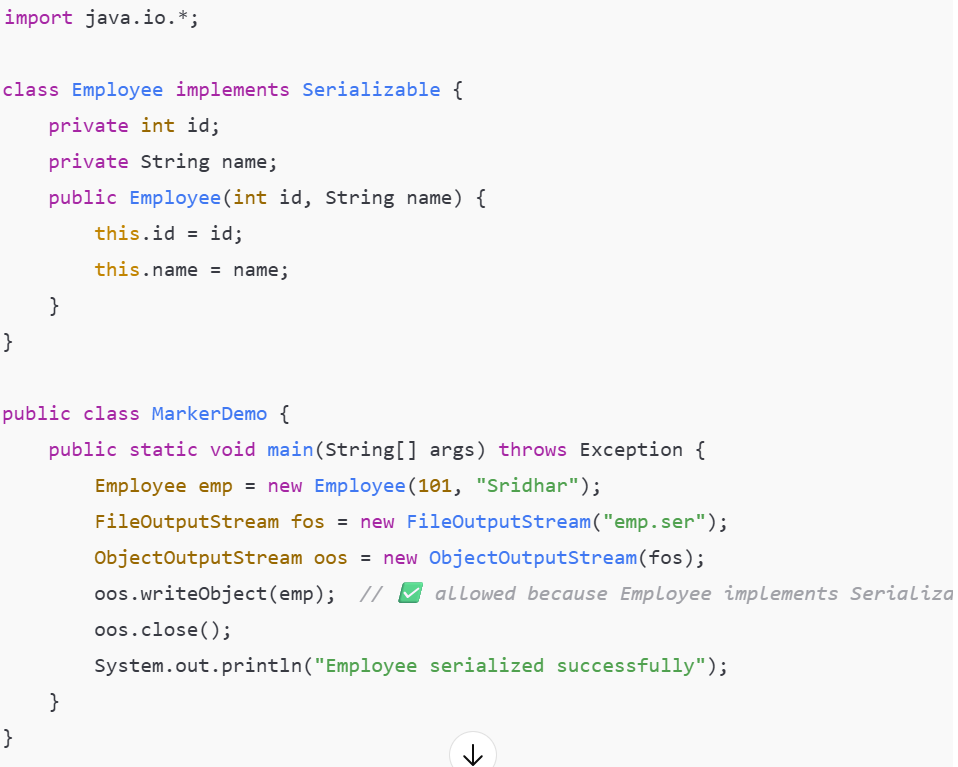
**A screenshot of a computer program

AI-generated content may be incorrect.**

| **Principle** | **Meaning** |
| --- | --- |
| **S** | **Do one thing only** |
| **O** | **Add new code without modifying old code** |
| **L** | **Subclasses must behave like parent class** |
| **I** | **Use small specific interfaces** |
| **D** | **Depend on abstractions, not concrete classes** |

**What is marker Interface?**

A **Marker Interface** is an **interface with no methods or fields**.  
It is used to **mark or tag a class** to indicate some special behavior to the JVM or framework.



**What is Hibernate session factory, Session transaction**

SessionFactory is a **thread-safe, heavyweight object** in Hibernate that **creates Session objects**.  
It is **initialized once** per application and acts as a **factory for sessions**.

A close up of a text

AI-generated content may be incorrect.

1. **Hibernate Session**

Session is a **lightweight, non-thread-safe object** that represents a **single unit of work with the database**.

A computer code with black text

AI-generated content may be incorrect.

**Hibernate Transaction**

Transaction in Hibernate is an object used to **manage database operations as a single unit of work**, ensuring **ACID properties**.

A screen shot of a computer code

AI-generated content may be incorrect.

**Q1: What is the difference between @Bean and @Component?**  
**A:** @Bean declares a bean in Java config; @Component marks a class for component scanning and automatic bean creation.

**Q2: Explain the flow of a request in Spring MVC.**  
**A:** Browser → DispatcherServlet → Controller → Service → Repository → Controller → View → Browser.

**Q3: What is the difference between Spring Framework and Spring Boot?**  
**A:** Spring needs manual configuration; Spring Boot is opinionated, auto-configures, and has embedded server support.

**Q4: What is the purpose of @Qualifier?**  
**A:** Selects which bean to inject when multiple beans of the same type exist.

**Q5: What is a DispatcherServlet?**  
**A:** Front controller in Spring MVC that handles all HTTP requests and routes them to controllers.

**Q6: What is ApplicationContext?**  
**A:** Spring container that manages beans, configurations, and dependency injection.

**Q7: What is lazy loading and eager loading?**  
**A:** Lazy → bean loaded only when needed; Eager → bean loaded at application startup.

**Java Basics / OOP**

**Q8: If a variable is final, how can it be initialized?**  
**A:** At declaration, in a constructor, or in an initializer block.

**Q9: How can we use an abstract method?**  
**A:** By overriding it in a subclass; declared in abstract class or interface.

**Q10: What is the difference between this() and super() in constructors?**  
**A:** this() calls the current class constructor;

super() calls the parent class constructor.

**Q11: What is synchronization in Java?**  
**A:** **Synchronization in Java** is a mechanism that ensures **only one thread can access a shared resource at a time**, preventing **thread interference** and **data inconsistency** in multithreading.

**Collections / Utilities**

**How can you print all data in a HashMap?**

A close-up of a computer code

AI-generated content may be incorrect.

**SPRING CORE**

**1. What is the Spring Framework and what problems does it solve?**

Spring is a Java framework for building loosely coupled enterprise-level applications.

It solves:

* Tight coupling → through IoC/DI
* Complex configurations → through annotations/Java config
* Boilerplate code → simplifies JDBC, transactions, security
* Integration with ORM, messaging, web, etc.

**2. What are the main modules of Spring Framework?**

* **Spring Core**
* **Spring Beans**
* **Spring Context**
* **Spring AOP**
* **Spring JDBC**
* **Spring ORM**
* **Spring MVC**
* **Spring Test**
* **Spring WebFlux**
* **Spring Expression Language (SpEL)**

**3. What is Inversion of Control (IoC)?**

IoC means objects do **not** create their dependencies; instead, Spring **injects** them.

**4. What is Dependency Injection (DI)?**

DI is IoC in practice: dependencies are provided from outside, not created within.

**5. Types of Dependency Injection**

* **Constructor Injection**
* **Setter Injection**
* **Field Injection** (not recommended)

**6. What is a Spring Bean?**

A Spring-managed object created inside the IoC container.

**7. How are beans defined in Spring?**

Using:

* XML configuration
* Annotations (@Component)
* Java-based config (@Bean)

**8. Different ways to configure Beans**

1. **XML**
2. **Annotation-based**
3. **Java Config**

**9. Difference between @Component, @Service, @Repository, @Controller**

All are **stereotype annotations**, but:

* @Component → generic bean
* @Service → business logic
* @Repository → DAO layer (adds exception translation)
* @Controller → MVC controller

**10. What is ApplicationContext?**

The Spring IoC container that manages beans, their lifecycle, dependencies, and configurations.

**🔹 BEAN LIFECYCLE & SCOPES**

**11. Spring Bean lifecycle**

1. Instantiate
2. Populate properties
3. Call BeanNameAware
4. Call BeanFactoryAware
5. Call BeanPostProcessor (before init)
6. Call init-method / @PostConstruct
7. Bean ready
8. Call destroy-method / @PreDestroy at context shutdown

**12. Bean Scopes**

* singleton (default)
* prototype
* request
* session
* application
* websocket

**13. Singleton vs Prototype**

* **singleton** → one instance per container
* **prototype** → new instance every request

**14. How to initialize & destroy beans?**

Using:

* @PostConstruct & @PreDestroy
* initMethod & destroyMethod
* XML (init-method, destroy-method)

**15. Purpose of @PostConstruct and @PreDestroy**

* Run code after bean creation
* Run cleanup before bean is destroyed

**16. Difference between BeanFactory and ApplicationContext**

| **BeanFactory** | **ApplicationContext** |
| --- | --- |
| Simple container | Advanced container |
| Lazy initialization | Eager initialization |
| No AOP/events | Supports AOP, events |

**17. Common dependency injection methods**

* Constructor injection (preferred)
* Setter injection

**18. How to create a lazy-initialized bean?**

@Lazy

@Bean

public MyBean bean() { ... }

**19. What happens if bean not found?**

Spring throws:

NoSuchBeanDefinitionException

**20. Use of @Lazy**

Delays bean creation until first use.

**🔹 CONFIGURATION & WIRING**

**21. Configuration approaches**

* XML
* Annotation-based
* Java Config (@Configuration)

**22. Purpose of @Configuration**

Marks a class as configuration source.

**23. Purpose of @Bean**

Creates a bean inside Java config class.

**24. What is component scanning?**

Automatically detects beans annotated with @Component, @Service, etc.

**25. Use of @ComponentScan**

Tells Spring where to search for components.

**26. What is autowiring?**

Automatic dependency injection by type.

Types:

* byType
* byName
* constructor
* no

**27. Use of @Autowired**

Injects a dependency automatically.

**28. Difference between @Autowired and @Inject**

* @Autowired → Spring
* @Inject → JSR-330 (Java standard)  
  Both work similarly.

**29. How to resolve multiple beans of same type?**

Use:

* @Qualifier
* @Primary

**30. Purpose of @Qualifier**

Identifies which bean to inject when multiple beans exist.

**🔹 AOP (Aspect Oriented Programming)**

**31. What is AOP and why use it?**

AOP handles **cross-cutting concerns**: logging, transactions, security.

**32. Key Concepts**

* **Aspect** → modular cross-cutting code
* **JoinPoint** → where aspect applies (method call)
* **Advice** → what to run
* **Pointcut** → where advice applies (expression)

**33. Types of Advice**

* Before
* After
* AfterReturning
* AfterThrowing
* Around

**34. What is a Pointcut expression?**

Expression that defines where advice should run.

**35. What is a proxy in AOP?**

A wrapper object that intercepts method calls.

**36. JDK vs CGLIB proxies**

| **JDK proxy** | **CGLIB proxy** |
| --- | --- |
| For interfaces | For classes |
| Java built-in | Bytecode manipulation |
| Cannot intercept final methods | Cannot intercept final methods |

**37. How to enable AOP?**

@EnableAspectJAutoProxy

**38. How does Spring manage cross-cutting concerns?**

By wrapping beans in proxies and applying advice logic.

**🔹 SPRING CONTEXT & RESOURCES**

**39. How does Spring load bean definitions?**

Via:

* XML
* Annotations
* Java config
* Classpath scanning

**40. ApplicationContext vs WebApplicationContext**

WebApplicationContext is for web apps and supports:

* servlet context
* web scopes
* MVC components

**41. Access properties from .properties file**

Using:

* @Value
* Environment
* @PropertySource

**42. Use of @PropertySource and @Value**

Load properties file and inject values.

**43. What is Environment?**

Provides access to:

* property values
* active profiles

**44. How do profiles work (@Profile)?**

Enable beans for specific environments:

@Profile("dev")

**🔹 TRANSACTION MANAGEMENT**

**45. What is transaction management?**

Ensuring operations run atomically and consistently.

**46. Programmatic vs Declarative**

* Programmatic → manual transaction code
* Declarative → @Transactional

**47. Use of @Transactional**

Marks a method/class as transactional.

**48. Propagation Levels**

* REQUIRED
* REQUIRES\_NEW
* SUPPORTS
* NEVER
* NOT\_SUPPORTED
* MANDATORY
* NESTED

**49. Isolation Levels**

* DEFAULT
* READ\_COMMITTED
* READ\_UNCOMMITTED
* REPEATABLE\_READ
* SERIALIZABLE

Ensure DB consistency.

**50. What is rollback?**

Undo changes in case of exception.

**51. Rollback for specific exceptions**

@Transactional(rollbackFor = CustomException.class)

**52. Can @Transactional be used on private methods?**

❌ No  
Because Spring uses proxies → private methods are not intercepted.

**🔹 EXCEPTION HANDLING & EVENTS**

**53. Global exception handling**

Use:

@ControllerAdvice

@ExceptionHandler

**54. Use of @ControllerAdvice**

Global exception handler for controllers.

**55. What are Spring Events?**

Publish-subscribe mechanism inside Spring.

**56. Publish/Listen custom events**

Publisher: applicationEventPublisher.publishEvent()  
Listener: @EventListener

**🔹 MISC / ADVANCED**

**57. ApplicationContext vs BeanFactory**

See #16.

**58. Spring Core vs Spring Boot**

| **Spring Core** | **Spring Boot** |
| --- | --- |
| Manual config | Auto-config |
| Complex setup | Opinionated defaults |
| No embedded server | Embedded Tomcat |

**59. Spring integration with Hibernate/JPA**

Uses:

* LocalSessionFactoryBean
* JpaTransactionManager
* HibernateTemplate

**60. What is circular dependency?**

Two beans depend on each other.

**Fix with:**

* Constructor → fails
* Setter injection → allowed
* @Lazy injection
* Redesign

**61. @Primary vs @Qualifier**

* @Primary → default bean
* @Qualifier → choose specific bean

**62. Inject values from environment variables**

Using:

* @Value
* Environment.getProperty()

**63. What is FactoryBean?**

A special bean that **returns an object created by the factory**, not itself.

**64. How does Spring achieve loose coupling?**

Through IoC + Dependency Injection.

**65. How does Spring handle singleton beans in multithreaded environments?**

Singleton beans must be:

* stateless
* thread-safe manually

Spring does **not** create per-thread singletons.

**66. Common pitfalls in Spring configuration**

* Circular dependencies
* Mixing XML + Java config incorrectly
* Incorrect bean scope
* Using field injection
* Transactional not working on private/final methods
* Forgetting component scanning

**SPRING DATA JPA**

**1. What is Spring Data JPA?**

Spring Data JPA is a Spring module that simplifies working with JPA by providing ready-made repository patterns, reducing boilerplate code, and enabling derived queries, pagination, and auditing.

**2. How is Spring Data JPA related to JPA and Hibernate?**

* **JPA** → Specification (interfaces)
* **Hibernate** → JPA implementation
* **Spring Data JPA** → Built on top of JPA; adds automatic repository implementations and query generation.

**3. Benefits of using Spring Data JPA over plain Hibernate**

* No need for boilerplate DAO code
* Auto-implemented repositories
* Derived queries from method names
* Built-in pagination and sorting
* Simple transaction management
* Easy integration with Spring Boot

**4. Required dependencies in Spring Boot**

Just one starter:

spring-boot-starter-data-jpa

(Plus a database driver.)

**5. Role of JpaRepository**

Provides CRUD + paging + sorting + JPA-specific operations for entities.

**6. CrudRepository vs PagingAndSortingRepository vs JpaRepository**

| **Interface** | **Features** |
| --- | --- |
| CrudRepository | CRUD operations |
| PagingAndSortingRepository | CRUD + paging + sorting |
| JpaRepository | All above + JPA-specific methods (flush, batch operations) |

**7. How Spring Data JPA reduces boilerplate**

**1. Auto-generates Repository Implementations**

Normally, you write:

* DAO classes
* EntityManager code
* CRUD methods
* Queries
* Transactions

With Spring Data JPA:

public interface UserRepository extends JpaRepository<User, Long> {}

You get:

* save()
* findById()
* findAll()
* deleteById()
* Paging, sorting

…all without writing any implementation.

**2. Derived Query Methods (Method Name Queries)**

Instead of writing:

@Query("SELECT u FROM User u WHERE u.email = :email")

User findByEmail(String email);

You can simply write:

User findByEmail(String email);

Spring auto-generates the query from method names.

**3. Built-in Paging and Sorting**

Without writing SQL, you get:

Page<User> findByRole(String role, Pageable pageable);

Handling limit/offset manually is not needed.

**4. Simplifies Transactions**

You don’t have to manage:

EntityTransaction tx = em.getTransaction();

tx.begin();

em.persist(user);

tx.commit();

Spring handles transactions using:

@Transactional

public void createUser(User user) { ... }

**5. Eliminates Boilerplate EntityManager Code**

You avoid:

* Creating EntityManager
* Creating queries manually
* Handling persistence context
* Mapping results

**8. Purpose of @Repository**

Marks repository classes, enables:

* Exception translation to Spring’s DataAccessException

**9. How does Spring Data JPA implement repository interfaces?**

Uses **Dynamic proxies** to generate implementations at runtime.

**10. Naming conventions for derived queries**

Starts with:

* findBy
* readBy
* getBy
* countBy
* existsBy  
  Then field names in CamelCase.

**🔹 ENTITY & MAPPING**

**11. What is an entity in JPA?**

A Java class mapped to a database table.

**12. How to define an entity class?**

@Entity

public class User {}

**13. Annotations for primary keys**

* @Id
* @GeneratedValue
* @EmbeddedId
* @IdClass

**14. Mapping relationships**

Use:

* @OneToOne
* @OneToMany
* @ManyToOne
* @ManyToMany

**15. Handling bidirectional relationships**

One side is **owning side**, other is **inverse** using mappedBy.

**16. Purpose of mappedBy**

Defines the **inverse side** and indicates that the FK belongs to the other entity.

**17. Cascading operations**

Cascading means **propagating an operation from one entity to its associated entities automatically**.

Example:  
If you save an **Order**, its associated **OrderItems** will also be saved automatically if cascading is enabled.

**⭐ Common Cascade Types (Most Asked in Interviews)**

| **Cascade Type** | **Meaning** |
| --- | --- |
| CascadeType.PERSIST | Saves child when parent is saved |
| CascadeType.MERGE | Merges child when parent is merged |
| CascadeType.REMOVE | Deletes child when parent is deleted |
| CascadeType.REFRESH | Refreshes child when parent is refreshed |
| CascadeType.DETACH | Detaches child when parent is detached |
| CascadeType.ALL | Applies **all** above operations |

**⭐ Example**

**Parent → Child Relationship**

@Entity

public class Order {

@Id

private Long id;

@OneToMany(mappedBy = "order", cascade = CascadeType.ALL)

private List<OrderItem> items;

}

**Child Entity**

@Entity

public class OrderItem {

@Id

private Long id;

@ManyToOne

private Order order;

}

**Usage**

Order order = new Order();

OrderItem item1 = new OrderItem();

OrderItem item2 = new OrderItem();

order.setItems(List.of(item1, item2));

orderRepository.save(order); // This saves order + both items

**⭐ Most Common Real-World Use Cases**

**1️⃣ Save parent → automatically save children**

Order → OrderItems  
User → Addresses

**2️⃣ Delete parent → automatically delete children**

If configured with CascadeType.REMOVE.

**3️⃣ Merge parent → also merge child**

During updates.

**⚠️ Important Interview Tips**

**🚫 Never use CascadeType.REMOVE on ManyToMany**

It may delete shared records.

**🚫 Cascade is not the same as orphanRemoval**

orphanRemoval = deletes children that are removed from the collection.

@OneToMany(mappedBy = "order", cascade = CascadeType.ALL, orphanRemoval = true)

**18. Cascade vs orphanRemoval**

* **Cascade**: propagate operations
* **orphanRemoval**: remove child when removed from parent collection

**19. @Embeddable and @Embedded**

Used for value types stored in same table as entity.

**20. Inheritance strategies**

* SINGLE\_TABLE
* JOINED
* TABLE\_PER\_CLASS

**🔹 REPOSITORY LAYER**

**21. Purpose of repository interfaces**

Provide CRUD abstractions without writing implementation.

**22. Creating custom query methods**

Write method names like:

findByEmail(String email)

**23. Naming strategy for derived queries**

Spring parses method names to create queries based on entity properties.

**24. How derived queries are resolved**

Uses reflection + parser to generate JPQL at runtime.

**25. Using @Query**

@Query("SELECT u FROM User u WHERE u.email = ?1")

**26. Native SQL queries**

@Query(value="SELECT \* FROM users", nativeQuery=true)

**What is @Query in Spring Data JPA?**

@Query is an annotation used in Spring Data JPA to write **custom queries** inside repository interfaces.

You use it when:

* Method name queries become complex
* You need custom joins
* You want full control over the query

Example:

@Query("SELECT u FROM User u WHERE u.email = :email")

User findByEmail(String email);

By default, @Query expects **JPQL** (Java Persistence Query Language).

**⭐ What is nativeQuery?**

nativeQuery is an attribute of @Query.

When you set nativeQuery = true, it means:  
➡️ You are writing **raw SQL**, NOT JPQL  
➡️ You refer to database tables and column names  
➡️ The query runs directly on the database engine

Example:

@Query(value = "SELECT \* FROM users WHERE email = :email", nativeQuery = true)

User findByEmailNative(String email);

**28. Purpose of @Modifying**

Required for update/delete JPQL queries.

**29. Purpose of @Transactional**

@Transactional is used to **declare a transactional boundary** around a method or class.

If you are transferring money between accounts:

* deduct from A
* add to B

If the second fails, the first must be rolled back.

@Transactional ensures that.

@Service

public class UserService {

@Transactional

public void saveUserData(User user, Address addr) {

userRepo.save(user);

addressRepo.save(addr);

}

}

If saving the address fails → user save is rolled back.

**30. Can repository methods return Optional?**

Yes — recommended for nullable results.

**🔹 PAGING & SORTING**

**31. Implement pagination**

Use Pageable parameter in methods.

**32. Pageable & PageRequest**

PageRequest.of(page, size) → creates pageable instance.

**33. Page vs Slice**

* Page → total pages count
* Slice → only next/previous available

**34. Sorting**

Use:

Sort.by("name")

**35. Combine pagination + sorting**

PageRequest.of(page, size, Sort.by("salary"))

**🔹 TRANSACTIONS**

**36. Transaction management**

Uses Spring’s @Transactional on service/repository methods.

**37. How to enable transactions**

@EnableTransactionManagement or implicitly enabled by Spring Boot.

**38. Default propagation**

Propagation.REQUIRED

**39. Read-only vs read-write**

* readOnly → optimizes performance, avoids dirty checks
* read-write → full transaction

**40. Can @Transactional be used at both levels?**

Yes, but recommended at **service layer**.

**🔹 QUERY METHODS**

**41. Methods using names**

e.g., findByName, deleteByEmail.

**42. Derived query keywords**

* findBy
* countBy
* existsBy
* deleteBy
* orderBy

**43. LIKE queries**

findByNameContaining(String name)

**44. Multiple conditions**

findByNameAndAge(String name, int age)

**45. Limit results**

findTop3ByOrderBySalaryDesc()

**46. IN / BETWEEN / NULL**

findByAgeIn(List<Integer> ages)

findBySalaryBetween(1000, 5000)

findByEmailIsNull()

**47. What is a projection?**

DTO-like view of selected fields only.

**48. Types of projections**

* Interface-based
* DTO class
* Dynamic projections

**49. Using SpEL**

@Query("select u from User u where u.role = :#{#role}")

**50. Calling stored procedures**

Use @Procedure or @Query.

**🔹 CUSTOM IMPLEMENTATION**

**51. Add custom methods**

Create:

* repository interface
* custom interface
* implementation class

**52. Manual method implementation**

Use EntityManager inside custom repository.

**53. @Repository purpose**

Exception translation + bean registration.

**54. Extend custom functionality**

Create base repository and extend it.

**55. Base repository interface**

Define shared operations all repositories can use.

**🔹 ENTITY MANAGER & PERSISTENCE CONTEXT**

**56. EntityManager role**

Manages persistence operations: persist, merge, remove, flush.

**57. EntityManager vs Session**

EntityManager = JPA abstraction  
Session = Hibernate-specific

**58. Persistence context**

Cache of managed entities within a transaction.

**59. Entity states**

* New
* Managed
* Detached
* Removed

**60. persist() vs merge()**

* persist → inserts new entity, makes managed
* merge → copies state into managed entity

**61. Detaching an entity**

entityManager.detach(entity);

**62. save() on detached entity**

Hibernate performs update or merge based on state.

**🔹 AUDITING & LISTENERS**

**63. What is auditing?**

Automatically tracking created/updated timestamps and users.

**64. Enable auditing**

@EnableJpaAuditing

**65. @CreatedDate & @LastModifiedDate**

Automatically populate timestamps.

**66. @EntityListeners**

Used with auditing or custom listeners.

**67. Auto-populating createdBy and updatedBy**

Use AuditorAware interface.

**🔹 PERFORMANCE & OPTIMIZATION**

**68. Common performance issues**

* N+1 queries
* EAGER loading
* No indexing
* Poor joins
* Too many flush operations

**69. N+1 select problem**

Occurs due to lazy loading in loops.

**70. Avoid N+1**

* Fetch joins
* EntityGraphs
* Batch fetching

**71. Fetch join vs entity graph**

* Fetch join → JPQL level
* Entity graph → annotation level, reusable

**72. Using @EntityGraph**

@EntityGraph(attributePaths = {"roles"})

**73. Enable second-level cache**

Use provider (Ehcache, Redis, Hazelcast) + config.

**74. Improve performance with batch fetching**

@BatchSize(size=20)

**75. Using projections**

Fetch only necessary fields → improves performance.

**📁 Project Structure**

src/

└── main/java/com/example/demo/

├── entity/

│ ├── Order.java

│ └── OrderItem.java

├── repository/

│ ├── OrderRepository.java

│ └── OrderItemRepository.java

├── service/OrderService.java

├── controller/OrderController.java

└── DemoApplication.java

**🍀 1. ENTITY: Order (Parent)**

Demonstrates:  
✔ Cascading  
✔ One-to-Many  
✔ orphanRemoval

package com.example.demo.entity;

import jakarta.persistence.\*;

import java.util.\*;

@Entity

public class Order {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String customer;

@OneToMany(

mappedBy = "order",

cascade = CascadeType.ALL,

orphanRemoval = true // also interview important

)

private List<OrderItem> items = new ArrayList<>();

public void addItem(OrderItem item) {

items.add(item);

item.setOrder(this);

}

public void removeItem(OrderItem item) {

items.remove(item);

item.setOrder(null);

}

// getters and setters

}

**🍀 2. ENTITY: OrderItem (Child)**

package com.example.demo.entity;

import jakarta.persistence.\*;

@Entity

public class OrderItem {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String product;

private int quantity;

@ManyToOne

@JoinColumn(name = "order\_id")

private Order order;

// getters and setters

}

**📘 3. REPOSITORY: OrderRepository**

Covers:  
✔ JPQL @Query  
✔ nativeQuery  
✔ paging  
✔ join query  
✔ update using @Modifying  
✔ delete using @Modifying

package com.example.demo.repository;

import com.example.demo.entity.Order;

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.data.jpa.repository.\*;

import org.springframework.data.repository.query.Param;

import org.springframework.stereotype.Repository;

import java.util.List;

@Repository

public interface OrderRepository extends JpaRepository<Order, Long> {

// ---- JPQL Query ----

@Query("SELECT o FROM Order o WHERE o.customer = :customer")

List<Order> findOrdersByCustomer(@Param("customer") String customer);

// ---- Native Query ----

@Query(value = "SELECT \* FROM orders WHERE customer = :customer", nativeQuery = true)

List<Order> findOrdersNative(@Param("customer") String customer);

// ---- Paging + Sorting ----

Page<Order> findByCustomer(String customer, Pageable pageable);

// ---- Update Query ----

@Modifying

@Transactional

@Query("UPDATE Order o SET o.customer = :newName WHERE o.id = :id")

int updateCustomerName(@Param("newName") String newName, @Param("id") Long id);

// ---- Delete Query ----

@Modifying

@Transactional

@Query("DELETE FROM Order o WHERE o.customer = :customer")

int deleteByCustomer(@Param("customer") String customer);

}

**📘 4. REPOSITORY: OrderItemRepository**

package com.example.demo.repository;

import com.example.demo.entity.OrderItem;

import org.springframework.data.jpa.repository.JpaRepository;

public interface OrderItemRepository extends JpaRepository<OrderItem, Long> {

}

**🛠️ 5. SERVICE LAYER (with @Transactional)**

Demonstrates:  
✔ @Transactional  
✔ cascading saves  
✔ rollback behavior  
✔ multiple repository operations

package com.example.demo.service;

import com.example.demo.entity.Order;

import com.example.demo.entity.OrderItem;

import com.example.demo.repository.OrderRepository;

import jakarta.transaction.Transactional;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

@Service

public class OrderService {

@Autowired

private OrderRepository orderRepository;

@Transactional

public Order createOrder() {

Order order = new Order();

order.setCustomer("Sridhar");

OrderItem item1 = new OrderItem();

item1.setProduct("Laptop");

item1.setQuantity(1);

OrderItem item2 = new OrderItem();

item2.setProduct("Mouse");

item2.setQuantity(2);

order.addItem(item1);

order.addItem(item2);

// because of CascadeType.ALL → both items will be saved automatically

return orderRepository.save(order);

}

@Transactional

public int changeCustomerName(Long id, String newName) {

return orderRepository.updateCustomerName(newName, id);

}

}

**🌐 6. CONTROLLER Layer**

Exposes REST APIs

package com.example.demo.controller;

import com.example.demo.entity.Order;

import com.example.demo.service.OrderService;

import com.example.demo.repository.OrderRepository;

import org.springframework.beans.factory.annotation.Autowired;

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

import java.util.List;

@RestController

@RequestMapping("/orders")

public class OrderController {

@Autowired

private OrderService service;

@Autowired

private OrderRepository repository;

@PostMapping("/create")

public Order createOrder() {

return service.createOrder();

}

@GetMapping("/customer/{name}")

public List<Order> getOrdersByCustomer(@PathVariable String name) {

return repository.findOrdersByCustomer(name);

}

@PutMapping("/{id}/rename/{newName}")

public int renameCustomer(@PathVariable Long id, @PathVariable String newName) {

return service.changeCustomerName(id, newName);

}

@DeleteMapping("/customer/{name}")

public int deleteByCustomer(@PathVariable String name) {

return repository.deleteByCustomer(name);

}

}

**🚀 7. Main Application Class**

package com.example.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class DemoApplication {

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}

**SPRING BOOT**

**1. What is Spring Boot?**

A framework that simplifies Spring development by providing auto-configuration, embedded servers, and production-ready features.

**2. How is Spring Boot different from Spring Framework?**

Spring Framework requires manual configuration; Spring Boot provides auto-configuration and starters to eliminate boilerplate.

**3. Advantages of Spring Boot**

* Auto-configuration
* Embedded servers
* Simplified dependency management

**4. Problems Spring Boot solves**

Removes complex XML configs, eliminates boilerplate, simplifies server setup, and automates dependency management.

**5. Role of spring-boot-starter dependencies**

Provide ready-made dependency bundles for specific features.

**6. Common Spring Boot Starters**

* spring-boot-starter-web
* spring-boot-starter-data-jpa
* spring-boot-starter-test
* spring-boot-starter-security

**7. Purpose of spring-boot-starter-parent**

Provides dependency management and default plugin versions.

**8. How Spring Boot simplifies dependency management**

A screenshot of a computer

AI-generated content may be incorrect.

**9. Default embedded server**

Tomcat.

**🔹 Application Setup**

**13. Difference of @SpringBootApplication and @EnableAutoConfiguration**

@SpringBootApplication includes scanning and configuration;

@EnableAutoConfiguration only handles auto-config.

**14. How Boot app starts**

SpringApplication.run() creates ApplicationContext and performs auto-config.

**15. Function of run()**

Loads context, starts embedded server.

**16. Standalone Boot vs WAR**

Boot runs with embedded server; WAR deploys to external container.

**1. What is a JAR?**

**JAR = Java ARchive**

Used for packaging a **standalone Java application**.

Contains compiled .class files, resources  
Can run independently using Java:

java -jar app.jar

**In Spring Boot**

A **JAR** contains an **embedded server** (Tomcat/Jetty/Undertow).

Your application **runs itself** — it does NOT need an external Tomcat server.

Example:

java -jar myapp.jar

**Most Spring Boot applications prefer JAR** because it is **simple, portable, easy to deploy**.

**What is a WAR?**

**WAR = Web Application ARchive**

Used for packaging **web applications** that must run on an **external servlet container** such as:

✔ Apache Tomcat  
✔ JBoss  
✔ WebLogic  
✔ WildFly

WAR files are deployed inside a server’s webapps/ folder.

**You do NOT run a WAR file directly.**  
The server runs it.

Example deploy path:

/usr/local/tomcat/webapps/myapp.war

**🔥 KEY DIFFERENCE (Easy Table)**

| **Feature** | **JAR** | **WAR** |
| --- | --- | --- |
| Contains embedded server (Tomcat) | ✔ Yes | ❌ No |
| Needs external server? | ❌ No | ✔ Yes |
| Deployment type | Standalone | Deploy on server |
| Common in Spring Boot | ✔ Yes | Sometimes |
| Start command | java -jar file.jar | Start via server |

**🔹 Auto-Configuration**

**21. What is auto-configuration?**

Automatically configures beans based on classpath and properties.

**22. How it works internally**

Loads auto-config classes using @EnableAutoConfiguration + condition checks.

**23. Role of @EnableAutoConfiguration**

Tells Spring Boot to enable auto-configuration mechanisms.

**26. Conditional annotations**

They help Spring:

* Load beans only when required
* Reduce unnecessary configurations
* Allow flexible auto-configuration

@ConditionalOnClass, @ConditionalOnBean, @ConditionalOnMissingClass @ConditionalOnMissingBean activate beans based on conditions.

**@ConditionalOnClass**

Loads bean **only if a specific class is present on the classpath**.

@ConditionalOnClass(name = "com.mysql.jdbc.Driver")

@Configuration

public class MySqlConfig {

}

Spring loads this config only if MySQL driver exists.

**2️⃣ @ConditionalOnMissingClass**

Opposite of above:

@ConditionalOnMissingClass("com.mysql.jdbc.Driver")

public class DefaultConfig { }

**3️⃣ @ConditionalOnBean**

Loads bean only if another bean exists.

@ConditionalOnBean(DataSource.class)

public class JdbcConfig { }

**4️⃣ @ConditionalOnMissingBean**

Loads bean only if bean is **not already defined**.

@Bean

@ConditionalOnMissingBean

public ObjectMapper objectMapper() {

return new ObjectMapper();

}

This prevents overriding user custom beans.

**27. Disable specific auto-config**

@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)

public class MyApp {}

@EnableAutoConfiguration(exclude = DataSourceAutoConfiguration.class)

**30. Custom auto-configuration**

Create config class + register it in AutoConfiguration.imports.

**🔹 Configuration & Properties**

**Externalized Configuration** in Spring Boot means:  
➡️ You keep your **application settings outside the code**,  
➡️ Changing configurations **does NOT require code changes**,  
➡️ Spring loads them automatically from multiple places

**32. Property sources**

* application.properties/yml
* OS env variables
* command-line args
* application-{profile}.yml

**33. Property resolution order**

Command-line > env vars > application.yml > default properties.

**What @ConfigurationProperties Does?**

✔ Maps configuration values to a POJO  
✔ Supports nested properties  
✔ Supports validation  
✔ Reduces repeated @Value usage  
✔ Helps in writing clean auto-configurations  
✔ Allows default values

**38. @Value vs @ConfigurationProperties**

* @Value → single property
* @ConfigurationProperties → group of properties

**@ConfigurationProperties (Recommended)**

* Maps multiple properties

Type-safe (int, boolean, List, Map)

* Supports nested objects
* Easier to test
* Cleaner (no multiple @Value)

**@Value (Not recommended for large configs)**

* One property at a time
* No grouping
* No type conversion for lists/maps
* Harder to maintain

@Value is a Spring annotation used to **inject a single property value** directly into a field, parameter, or method.

**application.properties**

app.title=My Spring App

**In Java:**

@Value("${app.title}")

private String title;

**39. Encrypt sensitive configs**

Use third-party tools like Jasypt.

**40. Environment-specific config**

Use application-dev.yml, application-prod.yml.

**🔹 Profiles & Environment**

**41. What are profiles?**

**Profiles** in Spring Boot allow you to load **different configurations for different environments** like:

* **dev** (development)
* **test** (testing)
* **prod** (production)
* **stage**
* **qa**

Profiles decide:

✔ Which properties file should load  
✔ Which beans should be active  
✔ Which configuration classes should run  
✔ Which components should be available in a specific environment

**42. How to activate**

spring.profiles.active=dev

**43. Activate from command line**

--spring.profiles.active=qa

**44. active vs include**

* active → selects profiles
* include → merges additional profiles

| **Property** | **Purpose** |
| --- | --- |
| **spring.profiles.active** | Sets the primary profile for environment |
| **spring.profiles.include** | Adds additional secondary profiles |

**What is Bean Resolution?**

When Spring starts your application, it:

1. **Scans** for beans
2. **Registers** them in the application context
3. **Resolves dependencies** → decides *which bean to inject* when a class needs one

Example:

public class MyController {

private final UserService userService;

}

Spring must decide:

* Which UserService bean to inject?
* Does it exist?
* Is there more than one?
* Is it primary?
* Is there a qualifier?

This decision-making process is called **Bean Resolution**.

**🧠 Spring’s Bean Resolution Process**

Spring resolves a bean using the following order:

**1️⃣ Type Matching (Primary Check)**

If a class needs:

UserService userService;

Spring first looks for a **bean of type UserService**.

If exactly **one** bean exists → inject it.

**2️⃣ If multiple beans of same type exist, Resolution Rules apply**

Example:

@Service

public class UserServiceImpl1 implements UserService {}

@Service

public class UserServiceImpl2 implements UserService {}

Now Spring sees **two beans of type UserService**, so injection fails:

❌ Ambiguous dependency error

NoUniqueBeanDefinitionException

**3️⃣ @Primary (Choose Default Bean)**

@Primary

@Service

public class UserServiceImpl1 implements UserService {}

Spring will inject this bean by default.

**4️⃣ @Qualifier (Choose Specific Bean)**

@Autowired

@Qualifier("userServiceImpl2")

private UserService service;

This tells Spring explicitly which bean to inject.

**5️⃣ Bean Name Matching (Last Option)**

If bean name matches parameter:

@Autowired

private UserServiceImpl1 userServiceImpl1;

Spring will inject the bean with the same name.

**6️⃣ Constructor Injection (Preferred and Automatically Resolved)**

public MyController(UserService userService) { }

Spring automatically resolves it.

If there are multiple beans, constructor injection helps readability.

**7️⃣ @Conditional beans (Load based on conditions)**

If a bean is annotated with:

@ConditionalOnMissingBean

@ConditionalOnClass

@ConditionalOnProperty

Spring may **skip** or **enable** the bean, affecting resolution.

**46. Set default profile**

spring.profiles.default=dev

**47. Inject env variables**

Use @Value("${ENV\_VAR}") or Environment object.

**48. Log active profile**

Print env.getActiveProfiles() on startup.

**49. Different DB config per profile**

Use multiple datasource configs in profile-based YAML files.

**50. Multiple profile files**

Yes: application-dev.yml, application-qa.yml, etc.

**🔹 Starters & Dependencies**

**51. What are starters?**

Predefined dependency bundles for common features.

**52. How they simplify management**

Remove need for manual version management.

**53. web vs webflux**

* web: blocking MVC
* webflux: reactive, non-blocking

**54. starter-data-jpa**

Adds Hibernate + Spring Data JPA.

**55. starter-test**

Includes JUnit, Mockito, Spring Test, AssertJ.

**56. Custom starter**

Yes — create module + auto-config + starter pom.

**57. Exclude a dependency**

<exclusion>

**58. Multiple versions of same library**

Dependency conflict → use BOM to resolve.

**60. Resolve conflicts**

Use dependency management block/BOM.

**🔹 Embedded Server**

**61. How Boot embeds Tomcat**

Uses Tomcat as a dependency with Spring boot starter web.

**62. Change server**

Include Jetty/Undertow starter and exclude Tomcat.

A screenshot of a computer program

AI-generated content may be incorrect.

**63. Change port**

server.port=8081

**64. Configure SSL**

Add keystore, set server.ssl.\* properties.

server.port=8443

server.ssl.enabled=true

server.ssl.key-store=classpath:server.p12

server.ssl.key-store-password=yourpassword

server.ssl.key-store-type=PKCS12

server.ssl.key-alias=myserver

**65. Set context path**

server.servlet.context-path=/app

**66. Two apps on same port**

One fails with port binding error.

**68. Purpose of context-path**

Defines application root URL.

**🔹 Logging**

**1. What logging framework does Spring Boot use by default?**

Spring Boot uses:

* **SLF4J** → Logging API (you write logs with this)
* **Logback** → Default logging implementation

When you write:

private static final Logger log = LoggerFactory.getLogger(MyClass.class);

SLF4J forwards it to Logback.

**Why Logback?**

✔ Fast  
✔ Powerful  
✔ Supports rolling logs  
✔ Supports XML config  
✔ Well-integrated with Spring Boot

**2. What are the logging levels in Spring Boot?**

| **Level** | **Description** |
| --- | --- |
| TRACE | Very detailed |
| DEBUG | Debugging information |
| INFO | Application flow |
| WARN | Potential issue |
| ERROR | Serious issue |

**Default level: INFO**

**🟦 3. How do you change logging level in Spring Boot?**

**application.properties**

logging.level.root=INFO

logging.level.com.example=DEBUG

logging.level.org.springframework.web=ERROR

**application.yml**

logging:

level:

root: INFO

com.example: DEBUG

Spring Boot automatically applies these at startup.

**🟦 4. How to write logs in Spring Boot?**

private static final Logger log = LoggerFactory.getLogger(MyService.class);

log.info("Order created");

log.debug("Order id = {}", id);

log.error("Failed to process order", exception);

With Lombok:

@Slf4j

public class MyService {

log.info("message");

}

**🟦 5. How to enable file logging?**

logging.file.name=app.log

OR

logging.file.path=logs/

Spring Boot:

✔ Creates file  
✔ Enables rotation (default 10MB)  
✔ Writes both console + file log (if configured)

**🟦 6. How to customize log format?**

logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n

**🟦 7. What is logback-spring.xml? Why use it?**

logback-spring.xml gives **full control** over logging.

Location:

src/main/resources/logback-spring.xml

Example:

<configuration>

<appender name="FILE" class="ch.qos.logback.core.FileAppender">

<file>logs/app.log</file>

<encoder>

<pattern>%d %level %logger{36} - %msg%n</pattern>

</encoder>

</appender>

<root level="INFO">

<appender-ref ref="FILE"/>

</root>

</configuration>

Features that properties cannot do:

✔ Rolling logs  
✔ Different appenders  
✔ Async logging  
✔ JSON logs  
✔ Conditional logging

**🟦 8. How to log in JSON format (Microservice requirement)?**

Add dependency:

<dependency>

<groupId>net.logstash.logback</groupId>

<artifactId>logstash-logback-encoder</artifactId>

<version>7.4</version>

</dependency>

Then configure:

<appender name="JSON" class="ch.qos.logback.core.FileAppender">

<file>logs/app-json.log</file>

<encoder class="net.logstash.logback.encoder.LogstashEncoder" />

</appender>

Used in:

* ELK (Elasticsearch + Logstash + Kibana)
* Splunk
* Graylog

**🟦 9. Logging configuration for different environments (profiles)**

**application-dev.properties**

logging.level.com.example=DEBUG

logging.file.name=dev.log

**application-prod.properties**

logging.level.com.example=ERROR

logging.file.path=/var/log/myapp

Spring loads based on:

spring.profiles.active=prod

**🟦 10. What is Actuator’s logging endpoint?**

If actuator is enabled:

GET /actuator/loggers

View all loggers & levels.

Change level at runtime:

POST /actuator/loggers/com.example

{

"configuredLevel": "DEBUG"

}

This is useful for:

✔ Production debugging  
✔ Changing log level without restarting app

**11. How do you log request & response in Spring Boot?**

Options:

**1. Using Filters:**

@Component

public class RequestFilter extends OncePerRequestFilter {

@Override

protected void doFilterInternal(HttpServletRequest req, HttpServletResponse res, FilterChain chain) {

log.info("Request URI: {}", req.getRequestURI());

chain.doFilter(req, res);

}

}

**2. Using Spring Boot properties:**

logging.level.org.springframework.web.filter.CommonsRequestLoggingFilter=DEBUG

**12. What is Logback rolling policy?**

Used to rotate log files automatically.

<rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">

<fileNamePattern>logs/app.%d{yyyy-MM-dd}.%i.log</fileNamePattern>

<maxFileSize>10MB</maxFileSize>

<maxHistory>7</maxHistory>

</rollingPolicy>

Benefits:  
✔ Prevents huge log files  
✔ Keeps logs clean  
✔ Auto-archives older logs

**13. Difference: logging.level vs logging.pattern?**

| **Property** | **Purpose** |
| --- | --- |
| logging.level | Changes log levels (INFO, DEBUG) |
| logging.pattern | Customizes how logs look |

**🔹 Actuator & Monitoring**

**1. What is Spring Boot Actuator?**

Actuator is a Spring Boot module that provides **production-ready monitoring features**, such as:

* Application health
* Environment properties
* Beans & mappings
* Log levels
* HTTP request metrics
* System info

**Why it is used?**

✔ Monitoring  
✔ Troubleshooting  
✔ Health checks for load balancers (AWS, Azure, Kubernetes)  
✔ Exposing metrics for Prometheus / Grafana

**2. How do you enable Actuator?**

Add dependency:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

Enable endpoints:

management.endpoints.web.exposure.include=\*

Default port: same as application.

**3. What are common Actuator endpoints?**

| **Endpoint** | **Purpose** |
| --- | --- |
| /actuator/health | App health |
| /actuator/metrics | JVM & custom metrics |
| /actuator/loggers | Live log level config |
| /actuator/beans | Bean details |
| /actuator/mappings | REST endpoint mappings |
| /actuator/env | Environment variables |
| /actuator/threaddump | JVM threads |
| /actuator/info | App info details |

**4. What is the /actuator/health endpoint?**

Used by:

* Kubernetes readiness & liveness probes
* AWS ELB
* API gateways

Example:

/actuator/health

Output:

{

"status": "UP"

}

You can also enable **database health**, **Redis health**, etc.

**5. How do you create a custom health indicator?**

Example:

@Component

public class MyServiceHealth implements HealthIndicator {

@Override

public Health health() {

if (checkService()) {

return Health.up().build();

}

return Health.down().withDetail("reason", "Service unavailable").build();

}

}

Output appears in /actuator/health.

**6. What is /actuator/metrics?**

Shows JVM + system metrics:

* CPU usage
* Thread count
* GC metrics
* HTTP request metrics
* DB connection metrics
* Cache metrics

Example:

/actuator/metrics/jvm.memory.used

/actuator/metrics/http.server.requests

**7. How do you integrate Actuator with Prometheus?**

Add:

<dependency>

<groupId>io.micrometer</groupId>

<artifactId>micrometer-registry-prometheus</artifactId>

</dependency>

Enable endpoint:

management.endpoints.web.exposure.include=prometheus

Now metrics available at:

/actuator/prometheus

Grafana can read these metrics.

**8. What is Micrometer?**

Micrometer is a **metrics facade** used by Spring Boot.

It supports:

* Prometheus
* New Relic
* Datadog
* Graphite
* CloudWatch

Micrometer collects metrics and sends them to any monitoring tool.

Actuator exposes the metrics, Micrometer exports them.

**9. How do you change log levels live using Actuator?**

Check loggers:

GET /actuator/loggers

Change log level at runtime:

POST /actuator/loggers/com.example

{

"configuredLevel": "DEBUG"

}

This is very useful in production debugging.

**10. How do you secure Actuator endpoints?**

Option 1 — Use Spring Security:

management.endpoint.health.show-details=always

management.endpoints.web.exposure.include=health,info,metrics

Then secure with roles.

Option 2 — Use custom security rules for "/actuator/\*\*".

**11. What is the /actuator/info endpoint? How do you customize it?**

Add in properties:

management.info.env.enabled=true

info.app.name=Order Service

info.app.version=1.0

info.build.time=2025-01-01

Output:

{

"app": {

"name": "Order Service",

"version": "1.0"

}

}

Useful for DevOps / CI-CD.

**12. What is a readiness and liveness probe? (Kubernetes)**

Kubernetes checks Actuator endpoints like:

**Liveness Probe → Is app running?**

/actuator/health/liveness

**Readiness Probe → Is app ready to accept traffic?**

/actuator/health/readiness

These endpoints prevent traffic from going to unhealthy pods.

**13. What is /actuator/threaddump?**

Shows:

* JVM threads
* Deadlocks
* Running thread states

Useful for diagnosing:

✔ High CPU usage  
✔ Blocking threads  
✔ Thread leaks

**14. What is endpoint filtering?**

To expose selected endpoints:

management.endpoints.web.exposure.include=health,metrics,loggers

Exclude sensitive ones:

management.endpoints.web.exposure.exclude=env,beans

**What is Spring MVC and how does it differ from Spring Core?**  
Spring MVC is the web framework from the Spring ecosystem for building web applications.

Spring Core is the module that provides IoC/DI and basic container services.

Spring MVC builds on Spring Core and adds HTTP/web-specific components (DispatcherServlet, HandlerMapping, ViewResolver, controllers).

**What are the main components of Spring MVC architecture?**  
**Main Components of Spring MVC Architecture**

Spring MVC has **7 core components**:

1. **DispatcherServlet**
2. **HandlerMapping**
3. **Controller** (or @Controller classes)
4. **Model and ModelAndView**
5. **ViewResolver**
6. **View**
7. **Front Controller Pattern**

Let’s explain each.

**DispatcherServlet (Front Controller)**

* The heart of Spring MVC
* Receives every incoming HTTP request
* Delegates to other components

It follows the **Front Controller Pattern**.

Spring Boot auto-configures DispatcherServlet.

**HandlerMapping**

Responsible for finding:

Which controller method should handle the request?  
 Which URL matches which method?

Example:

@GetMapping("/users")

public String getUsers() { ... }

HandlerMapping maps "/users" to this method.

**Controller**

These are your classes annotated with:

* @Controller
* @RestController

Controllers:

Process the request  
Interact with services  
Prepare data for the view (or return JSON)

Example:

@Controller

public class UserController {

@GetMapping("/home")

public String home(Model model) {

model.addAttribute("msg", "Welcome!");

return "home";

}

}

**Model and ModelAndView**

Used to pass data from Controller → View

Example:

model.addAttribute("user", user);

Or:

return new ModelAndView("home", "msg", "Welcome!");

Model holds data.  
View name decides which page should render.

**5️⃣ ViewResolver**

Decides **which view should be rendered**.

For example, if controller returns:

return "home";

ViewResolver decides:

/WEB-INF/views/home.jsp  
OR Thymeleaf template /templates/home.html

Types of view resolvers:

* InternalResourceViewResolver
* ThymeleafViewResolver
* BeanNameViewResolver

**6️⃣ View**

The actual UI:

JSP  
 Thymeleaf  
 HTML  
 FreeMarker

The view receives the model data and displays it.

**Front Controller Pattern**

Spring MVC uses this pattern:

All requests go to **one servlet** (DispatcherServlet)  
 It processes the request and forwards to controllers  
 Unified handling improves maintainability

**Spring MVC Request Flow**

Client → DispatcherServlet

→ HandlerMapping

→ Controller

→ Business Logic/Service

→ Return Model + View name

→ ViewResolver

→ View (HTML/JSP/Thymeleaf)

→ Response sent to client

**1. @Controller**

@Controller is used for **traditional MVC web applications** where the controller returns **views (HTML pages)**.

**Key Points**

Returns **view templates** (like JSP, Thymeleaf, HTML)  
You must use @ResponseBody if you want to return JSON  
Used for server-side rendering

**Example**

@Controller

public class MyController {

@GetMapping("/home")

public String home() {

return "home"; // returns view name -> home.html or home.jsp

}

@GetMapping("/data")

@ResponseBody

public String getData() {

return "Hello"; // returns raw text/json

}

}

**What are the different ways to handle requests in Spring MVC?**

**Traditional MVC controllers**

* @Controller methods returning ModelAndView or view names.

**REST-style controllers**

* @RestController methods returning domain objects (JSON/XML) serialized by message converters.

**Using @ResponseBody**

* @RequestMapping/@GetMapping methods returning raw data (String/JSON).

**Using ResponseEntity<T>**

* For full control of status, headers, and body.

**🔹 Request Handling**

**What is @PathVariable and how is it used?**  
@PathVariable binds URI template variables to method parameters.

Example:

@GetMapping("/users/{id}")

public User get(@PathVariable Long id) { ... }

**What is @RequestParam and how is it different from @PathVariable?**  
@RequestParam binds query parameters or form data (e.g., ?page=1), while @PathVariable binds values embedded in the URI path.

Example:

@GetMapping("/search")

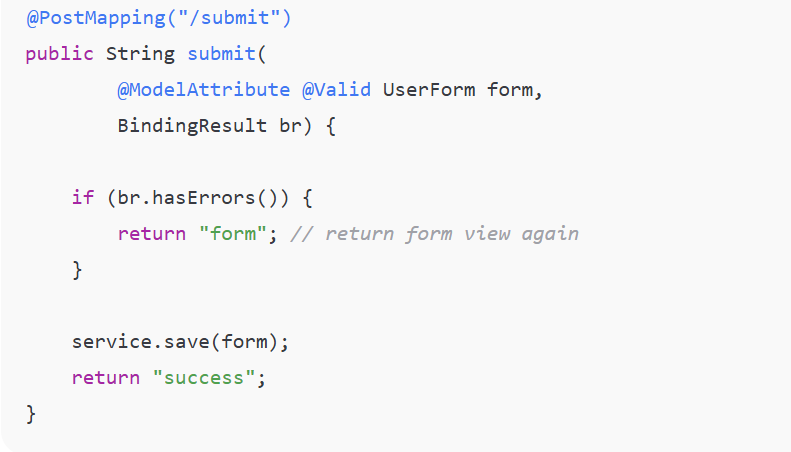
public List<Item> search(@RequestParam String q) { ... }

**How do you handle query parameters in Spring MVC?**  
In Spring MVC, query parameters are primarily handled using @RequestParam.  
You can also bind them into a Map, bind them into a POJO using @ModelAttribute, or access them through HttpServletRequest if needed.

**What is @RequestBody and when do you use it?**  
@RequestBody binds HTTP request body to a Java object using HttpMessageConverter (e.g., JSON -> POJO using Jackson). Used for JSON/XML payloads in REST controllers.

**What is @ResponseBody and how does it differ from returning a view?**  
@ResponseBody tells Spring to write the method return value directly to HTTP response body with a proper HttpMessageConverter (JSON/XML). Returning a view name instructs Spring to resolve and render a template.

**How do you handle form data in Spring MVC?**  
Form data in Spring MVC is mainly handled using @ModelAttribute, which binds form fields to a Java model object. You can use @Valid and BindingResult to perform validation and capture errors. For small forms, you may use @RequestParam, and for file uploads, MultipartFile is used.



**What is the difference between @ModelAttribute and @RequestBody?**  
@ModelAttribute binds form URL-encoded parameters and populates a POJO (and is used for view-based apps); @RequestBody binds the raw request body (e.g., JSON) to a POJO.

**How do you handle file uploads in Spring MVC?**  
Enable MultipartResolver (Spring Boot config auto-enables), accept MultipartFile in controller:

@PostMapping("/upload")

public String upload(@RequestParam("file") MultipartFile file) throws IOException {

String fileName = file.getOriginalFilename();

byte[] bytes = file.getBytes();

// Save to local disk

Path path = Paths.get("uploads/" + fileName);

Files.write(path, bytes);

return "success";

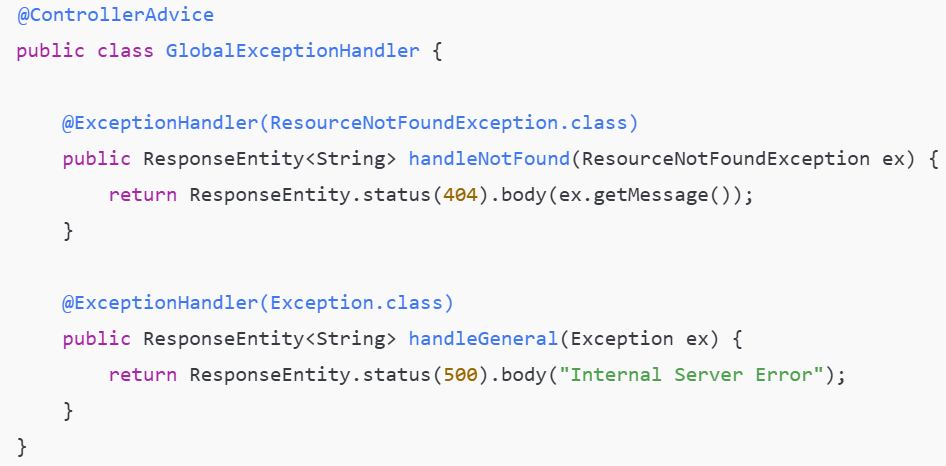
}

**What is MultipartFile in Spring?**  
Abstraction representing an uploaded file. Provides getInputStream(), getBytes(), transferTo() etc.

**How can you restrict allowed HTTP methods for a handler method?**  
Use @GetMapping, @PostMapping, or @RequestMapping(method = RequestMethod.POST) to restrict accepted methods. Container returns 405 for unsupported methods.

**🔹 Exception Handling**

**How do you handle exceptions globally in Spring MVC?**  
Use @ControllerAdvice with methods annotated @ExceptionHandler (ExceptionType.class) to centralize exception-to-response mapping. You can also extend ResponseEntityExceptionHandler for common cases.



**What is @ExceptionHandler and how is it used?**  
@ExceptionHandler marks a method that handles exceptions thrown from controller methods. It can return ResponseEntity or model and view.

**What is @ControllerAdvice?**  
This allows you to apply common logic **globally across all controllers** in a Spring MVC application.

**How do you handle different response statuses for exceptions?**  
Return ResponseEntity with appropriate HTTP status, or annotate exception handler methods with @ResponseStatus.

**How do you return custom error responses for REST APIs?**  
In @ControllerAdvice create @ExceptionHandler methods that produce structured error objects (timestamp, status, message, errors) and return ResponseEntity<ErrorDto>.

**How can you handle validation errors gracefully?**  
Intercept MethodArgumentNotValidException in an @ExceptionHandler, extract field errors, and return a structured error response listing field-specific messages.

**What is the difference between ResponseEntityExceptionHandler and @ExceptionHandler?**  
**1. @ExceptionHandler — Method-level exception handling**

@ExceptionHandler is used inside a controller or inside a @ControllerAdvice class to handle **specific exceptions**.

**✔ Key Characteristics**

* You write your own method to handle exceptions
* Can be applied **locally** (per controller) or **globally** (via @ControllerAdvice)
* Complete freedom to customize response

**2. ResponseEntityExceptionHandler — Predefined centralized handler**

ResponseEntityExceptionHandler is a **Spring-provided base class** containing many built-in exception-handling methods for Spring MVC, especially for **REST APIs**.

You extend it in your @ControllerAdvice class to override Spring's default behaviors.

**Example:**

@ControllerAdvice

public class RestExceptionHandler extends ResponseEntityExceptionHandler {

@Override

protected ResponseEntity<Object> handleMethodArgumentNotValid(

MethodArgumentNotValidException ex,

HttpHeaders headers,

HttpStatus status,

WebRequest request) {

return ResponseEntity.badRequest().body("Validation failed");

}

}

**How do you handle 404 errors in Spring Boot REST APIs?**  
In Spring Boot REST APIs, 404 errors can be handled by defining a global exception handler using @ControllerAdvice for custom "resource not found" errors, or by overriding handleNoHandlerFoundException in ResponseEntityExceptionHandler to customize the response for unmapped URLs

**How do you log exceptions in Spring MVC?**  
**1. Logging exceptions using @ControllerAdvice (Most Common, Best Practice)**

Use a global exception handler and log the exception inside @ExceptionHandler.

**Example:**

@ControllerAdvice

public class GlobalExceptionHandler {

private static final Logger log = LoggerFactory.getLogger(GlobalExceptionHandler.class);

@ExceptionHandler(Exception.class)

public ResponseEntity<String> handleException(Exception ex) {

log.error("Error occurred:", ex); // Logs full stack trace

return ResponseEntity.status(500).body("Internal Server Error");

}

}

✔ Logs **all exceptions**  
✔ Clean controllers  
✔ Centralized logging

**✅ 2. Logging inside a @ExceptionHandler at Controller level**

Local logging for specific controllers only:

@Controller

public class UserController {

private static final Logger log = LoggerFactory.getLogger(UserController.class);

@ExceptionHandler(NumberFormatException.class)

public String handleNumberFormat(NumberFormatException ex) {

log.warn("User entered invalid number", ex);

return "error";

}

}

**Can you use multiple @ControllerAdvice classes?**

Yes, Spring allows multiple @ControllerAdvice classes. They can be scoped independently, ordered using @Order, and used to separate different types of global exception handling or cross-cutting concerns.

**Request & Response Customization**

**How do you customize HTTP headers in Spring MVC?**  
@GetMapping("/download")

public ResponseEntity<String> download() {

HttpHeaders headers = new HttpHeaders();

headers.add("Content-Type", "text/plain");

headers.add("X-Custom-Header", "MyValue");

return new ResponseEntity<>("File Content", headers, HttpStatus.OK);

}

**How do you set response status codes in Spring REST?**  
Use ResponseEntity.status(HttpStatus.CREATED) or annotate controller methods with @ResponseStatus(HttpStatus.NO\_CONTENT).

**What is ResponseEntity and why is it useful?**  
**What is ResponseEntity?**

ResponseEntity is a Spring class that represents an **HTTP response**, including:

* **Body**
* **Status code**
* **Headers**

It gives you **full control** over the entire HTTP response when building REST APIs.

public class ResponseEntity<T>

It wraps your response data (T) with metadata like headers and status.

**✅ Why is ResponseEntity useful?**

**1. You can set custom HTTP status codes**

return ResponseEntity.status(HttpStatus.CREATED).body("Created");

**2. You can set headers**

return ResponseEntity.ok()

.header("X-Custom-Header", "Value")

.body("Hello");

**3. You can return any type of body**

JSON, text, files, binary data—anything.

**🟦 Practical Example**

@GetMapping("/user")

public ResponseEntity<User> getUser() {

User user = new User("Sridhar", 30);

return ResponseEntity

.ok()

.header("X-App-Version", "1.0")

.body(user);

}

This sends:

* **200 OK**
* custom header
* JSON body of User

**How do you return different media types (JSON, XML) from the same endpoint?**  
You can return multiple media types from the same endpoint using Spring’s content negotiation. Annotate the method with produces = {"application/json", "application/xml"} and Spring will automatically return JSON or XML based on the request’s Accept header

@GetMapping(value = "/user", produces = {MediaType.APPLICATION\_JSON\_VALUE, MediaType.APPLICATION\_XML\_VALUE})

public User getUser() {

return new User("Sridhar", 30);

}

**How do you implement content negotiation in Spring Boot?**  
Spring Boot auto-configures content negotiation based on Accept header, path extensions, or query parameter. Configure ContentNegotiationConfigurer for custom behavior.

**How can you serialize and deserialize custom objects?**  
Provide custom Jackson Serializer/Deserializer or use @JsonSerialize/@JsonDeserialize, or register Module with ObjectMapper.

A screenshot of a computer code

AI-generated content may be incorrect.

A screenshot of a computer code

AI-generated content may be incorrect.

**What is @JsonIgnore and @JsonProperty in Spring Boot?**  
@JsonIgnore is a Jackson annotation used to **exclude a field from JSON serialization and deserialization**.

**✔ Meaning:**

* Field **will NOT appear** in JSON output
* Field **will NOT be read** from JSON input

@JsonProperty will **customize the JSON field name** OR **include fields that are normally ignored**.

**How do you handle circular references in JSON serialization?**

Circular references happen when two objects reference each other, causing **infinite loops** during JSON serialization.

public class User {

@JsonManagedReference // forward part

private Address address;

}

public class Address {

@JsonBackReference // backward part

private User user;

}

✔ Result:

@JsonManagedReference → included in JSON

@JsonBackReference → ignored to prevent recursion

**How do you handle large JSON responses efficiently?**  
Stream responses using Jackson's streaming API, return ResponseEntity<StreamingResponseBody> or use server-side pagination to avoid large payloads.

**How do you compress REST API responses in Spring Boot?**  
Enable server compression in properties: server.compression.enabled=true and configure mime types and minimum size.

**Filters, Interceptors, and CORS**

**1. What is a Filter?**

A **Filter** is part of the **Servlet API**, running before the Spring framework.

**Key characteristics:**

* Works at **very low level**
* Sees *every* request (even static files)
* Good for:
  + Logging request metadata
  + Security checks before hitting Spring
  + Compression
  + CORS
  + Sanitizing input
* Cannot access controller method or parameters

**Example:**

@Component

public class MyFilter implements Filter {

@Override

public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)

throws IOException, ServletException {

System.out.println("Filter: request coming");

chain.doFilter(req, res);

System.out.println("Filter: response leaving");

}

}

**🟦 2. What is an Interceptor?**

An **Interceptor** is part of **Spring MVC**.  
It runs during Spring’s request handling lifecycle.

**Key characteristics:**

* Executes:
  + Before controller (preHandle)
  + After controller but before view (postHandle)
  + After complete processing (afterCompletion)
* Can see:
  + Handler (controller object)
  + Handler method
  + ModelAndView
  + Exceptions

**Example:**

public class MyInterceptor implements HandlerInterceptor {

@Override

public boolean preHandle(HttpServletRequest req, HttpServletResponse res, Object handler) {

System.out.println("Interceptor: before controller");

return true;

}

@Override

public void postHandle(HttpServletRequest req, HttpServletResponse res, Object handler, ModelAndView mv) {

System.out.println("Interceptor: after controller");

}

@Override

public void afterCompletion(HttpServletRequest req, HttpServletResponse res, Object handler, Exception ex) {

System.out.println("Interceptor: after view rendering");

}

}

**How do you implement a Filter in Spring Boot?**  
**1. Implement a Filter using @Component (Most Common & Easiest)**

Just implement javax.servlet.Filter (or jakarta.servlet.Filter in newer versions) and annotate it with @Component.

**Example:**

@Component

public class MyFilter implements Filter {

@Override

public void doFilter(

ServletRequest request,

ServletResponse response,

FilterChain chain) throws IOException, ServletException {

System.out.println("Filter: Request received");

// Continue the request

chain.doFilter(request, response);

System.out.println("Filter: Response returned");

}

}

✔ Automatically detected by Spring Boot  
✔ No configuration needed  
✔ Executes for all requests

**✅ 2. Register a Filter Manually using FilterRegistrationBean**

Use this when you need:

* Filter ordering
* URL pattern mapping
* Conditional enabling

**Example:**

@Configuration

public class FilterConfig {

@Bean

public FilterRegistrationBean<MyFilter> registerMyFilter() {

FilterRegistrationBean<MyFilter> reg = new FilterRegistrationBean<>();

reg.setFilter(new MyFilter());

reg.addUrlPatterns("/api/\*"); // apply only for specific URLs

reg.setOrder(1); // order of execution

return reg;

}

}

Control where the filter applies  
Control filter order  
Useful when multiple filters exist

**Example Filter with Logging**

@Component

public class LoggingFilter implements Filter {

@Override

public void doFilter(

ServletRequest request,

ServletResponse response,

FilterChain chain) throws IOException, ServletException {

HttpServletRequest req = (HttpServletRequest) request;

System.out.println("Incoming request: " + req.getRequestURI());

chain.doFilter(request, response);

System.out.println("Outgoing response");

}

}

**When is preHandle, postHandle, and afterCompletion called?**

* preHandle: before controller method invocation.
* postHandle: after controller returns but before view rendering.
* afterCompletion: after view rendered and request completed (useful for cleanup).

**How do you configure global CORS settings in Spring Boot?**  
**1. Global CORS Using WebMvcConfigurer**

You define a configuration class that applies CORS to *all controllers*.

**Example:**

@Configuration

public class GlobalCorsConfig implements WebMvcConfigurer {

@Override

public void addCorsMappings(CorsRegistry registry) {

registry.addMapping("/\*\*") // apply to all paths

.allowedOrigins("http://localhost:3000") // allowed origins

.allowedMethods("GET", "POST", "PUT", "DELETE")

.allowedHeaders("\*")

.allowCredentials(true);

}

}

✔ Works for all REST endpoints  
✔ Full flexibility  
✔ Most used in real projects

**✅ 2. Global CORS Using application.properties (Spring Boot 3.x / Spring 6)**

Spring now supports simple CORS config via properties:

spring.web.cors.allowed-origins=http://localhost:3000

spring.web.cors.allowed-methods=GET,POST,PUT,DELETE

spring.web.cors.allowed-headers=\*

spring.web.cors.allow-credentials=true

spring.web.cors.max-age=3600

✔ Easiest way  
✔ No Java code needed  
❌ Less flexible than WebMvcConfigurer

**✅ 3. Global CORS for Spring Security (IMPORTANT)**

If you use Spring Security, MVC CORS settings alone **won’t work**.  
You must also enable CORS in the security config:

**Spring Boot 3.x / Spring Security 6:**

@Configuration

public class SecurityConfig {

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.cors(withDefaults()) // enable CORS

.csrf(csrf -> csrf.disable());

return http.build();

}

}

If needed, register a CORS config source:

@Bean

public CorsConfigurationSource corsConfigurationSource() {

CorsConfiguration config = new CorsConfiguration();

config.addAllowedOrigin("http://localhost:3000");

config.addAllowedMethod("\*");

config.addAllowedHeader("\*");

config.setAllowCredentials(true);

UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();

source.registerCorsConfiguration("/\*\*", config);

return source;

}

✔ Needed if security is enabled  
✔ Prevents "CORS blocked" issues

**Global CORS vs. @CrossOrigin**

| **Type** | **Scope** |
| --- | --- |
| **Global CORS** | Entire application |
| @CrossOrigin | Only specific controller or method |

**How do you configure CORS for a single endpoint?**  
Use @CrossOrigin(origins="...") on controller/handler method.

A screen shot of a computer code

AI-generated content may be incorrect.

**What is the difference between @CrossOrigin and global CORS config?**  
@CrossOrigin applies to specific controllers/methods; global config applies app-wide and is centralized.

**How do you log request and response payloads in Spring MVC?**

Use a Filter or OncePerRequestFilter that wraps HttpServletRequest and HttpServletResponse to log the input/output streams. Alternatively use ContentCachingRequestWrapper and ContentCachingResponseWrapper. This allows you to safely read the body for logging. Interceptors can log metadata but cannot read the body unless wrapped earlier.

**How do you handle authentication and authorization for REST APIs?**  
Authentication verifies who the user is, and authorization checks what they are allowed to access. In Spring Boot REST APIs, this is typically implemented using JWT tokens and Spring Security filters. You configure security rules in SecurityFilterChain, validate tokens using custom filters, and restrict access using hasRole, @PreAuthorize, or permission-based rules. Other supported methods include OAuth2, Basic Auth, and API keys.

**How do you prevent CSRF attacks in RESTful services?**  
RESTful services usually disable CSRF because they are stateless and rely on tokens (JWT, OAuth, API keys) instead of cookies. Token-based authentication is not automatically sent by the browser, so CSRF attacks cannot succeed. If cookies must be used, configure SameSite, HttpOnly, Secure flags, and use proper CORS rules.

**CSRF AND CORS(Cross Site Request Forgery & Cross Origin Resource Sharing)**

**CSRF** protects your server from **malicious requests using the user's own credentials**,  
while **CORS** controls **which external domains are allowed to access your API** from a browser.

**🔹 Pagination & Sorting**

**How do you implement pagination in REST APIs with Spring Data JPA?**  
Expose endpoints that accept Pageable parameters and return Page<T>:

@GetMapping("/users")

public Page<User> list(Pageable pageable) { return userRepository.findAll(pageable); }

**How do you implement sorting in REST APIs?**  
Add Sort to PageRequest or accept sort query param (?sort=lastName,asc).

**What is Pageable and Page in Spring Data?**  
Pageable describes pagination parameters (page, size, sort). Page<T> is a result wrapper containing content and pagination metadata (totalPages, totalElements).

**How do you return metadata in REST responses?**  
Return the Page<T> directly (it contains metadata) or map Page to a custom DTO with content and meta fields (recommended for API standardization).

**How do you handle large datasets efficiently in REST APIs?**  
Use pagination, streaming, cursor-based pagination, filtering, and proper indexing on DB.

**🔹 Advanced Topics**

**How do you implement HATEOAS links?**  
HATEOAS = Hypermedia as the Engine of Application State.

It means your REST responses include **links** that tell clients what actions they can perform next.

A computer code with text

AI-generated content may be incorrect.

A computer screen shot of a computer code

AI-generated content may be incorrect.

**2. Create Your Domain Model**

(Regular POJO—no need to add links here)

public class Product {

private Long id;

private String name;

// getters and setters

}

**✅ 3. Create RepresentationModel**

This wraps your response + links.

import org.springframework.hateoas.RepresentationModel;

public class ProductModel extends RepresentationModel<ProductModel> {

private Long id;

private String name;

public ProductModel(Long id, String name) {

this.id = id;

this.name = name;

}

// getters

}

**✅ 4. Create Assembler to Build Links**

This class converts your model → HATEOAS model.

import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.\*;

import org.springframework.hateoas.server.RepresentationModelAssembler;

import org.springframework.stereotype.Component;

@Component

public class ProductModelAssembler implements RepresentationModelAssembler<Product, ProductModel> {

@Override

public ProductModel toModel(Product entity) {

ProductModel model = new ProductModel(entity.getId(), entity.getName());

model.add(linkTo(methodOn(ProductController.class)

.getProduct(entity.getId()))

.withSelfRel());

model.add(linkTo(methodOn(ProductController.class)

.getAllProducts())

.withRel("all-products"));

return model;

}

}

**✅ 5. Use It Inside Controller**

@RestController

@RequestMapping("/products")

public class ProductController {

@Autowired

private ProductService service;

@Autowired

private ProductModelAssembler assembler;

@GetMapping("/{id}")

public ProductModel getProduct(@PathVariable Long id) {

Product p = service.getById(id);

return assembler.toModel(p);

}

@GetMapping

public CollectionModel<ProductModel> getAllProducts() {

List<Product> productList = service.getAll();

List<ProductModel> modelList = productList.stream()

.map(assembler::toModel)

.collect(Collectors.toList());

return CollectionModel.of(modelList,

linkTo(methodOn(ProductController.class).getAllProducts()).withSelfRel());

}

}

**🟢 Example JSON Response**

When you GET /products/10:

{

"id": 10,

"name": "Laptop",

"\_links": {

"self": {

"href": "http://localhost:8080/products/10"

},

"all-products": {

"href": "http://localhost:8080/products"

}

}

}

**How do you implement rate limiting in REST APIs?**  
Use gateway-level limiters (API Gateway), a filter with in-memory or distributed counters (Redis), or libraries like Bucket4j.

**How do you implement caching for REST endpoints?**  
Use Spring Cache (@Cacheable) or cache HTTP responses (via reverse proxy/CDN). For REST, ensure correct cache keys and TTLs.

**How do you handle API throttling?**  
Implement throttling at gateway, use token bucket algorithms or rate-limiter libraries.

**How do you document REST APIs using Swagger/OpenAPI?**  
Use springdoc-openapi or springfox (springdoc recommended). Annotate controllers/models and expose /v3/api-docs and Swagger UI.

**How do you implement global logging for REST requests/responses?**  
Use filters/interceptors for logging and correlation IDs. Use structured logging (JSON) and centralize logs with ELK stack.

**How do you secure REST APIs using Spring Security?**  
Configure HttpSecurity to require authentication, define stateless sessions for JWT, use OAuth2 resource server for token validation, and use method-level security for fine-grained access.

**How do you integrate REST APIs with JWT or OAuth2 authentication?**  
For JWT: implement authentication filter to parse/validate token and set SecurityContext. For OAuth2: configure resource server (spring-boot-starter-oauth2-resource-server) or client for social login.

**🔐 SPRING SECURITY — DETAILED ANSWERS**

**🔹 Basics & Fundamentals**

**What is Spring Security and why is it used?**  
Spring Security is a powerful and highly customizable authentication and access-control framework for Java apps. It handles authentication, authorization, password hashing, session management, CSRF protection, and integrates with other security standards (OAuth2, JWT).

**What are the main features of Spring Security?**  
Authentication, authorization, method security, CSRF protection, session management, security event logging, password hashing, OAuth2/OpenID Connect support, and integration with LDAP/AD.

**What is the difference between authentication and authorization?**

* Authentication verifies an identity (who you are).
* Authorization determines what actions/resources the authenticated user is allowed to access.

**What is the default authentication mechanism in Spring Security?**  
Historically form-based login (with default login page) and HTTP Basic. In recent Spring Security versions, you configure a SecurityFilterChain explicitly.

**What is the difference between HTTP Basic, Form Login, and OAuth2 login?**

* HTTP Basic: sends credentials in Authorization header (base64) — simple, mostly for APIs/testing.
* Form Login: username/password posted via HTML form, uses session cookie once authenticated.
* OAuth2 login: delegates authentication to external providers (Google, Facebook) and handles OAuth2 flows.

**How does Spring Security integrate with Spring Boot?**  
Add spring-boot-starter-security and configure security via SecurityFilterChain beans or WebSecurityConfigurerAdapter (deprecated in newer versions). Spring Boot auto-configures security defaults.

**What is a SecurityFilterChain?**  
An ordered chain of Filters that apply security decisions to each request. You configure it via HttpSecurity in a SecurityFilterChain bean.

**What is the role of UsernamePasswordAuthenticationFilter?**  
Filter that handles form login by processing username/password submissions, attempts authentication via AuthenticationManager, and establishes the security context.

**What is the purpose of AuthenticationManager?**  
Core Spring Security interface that processes Authentication requests. It delegates to AuthenticationProviders for actual verification.

**What is the difference between AuthenticationProvider and UserDetailsService?**

* AuthenticationProvider performs authentication logic and returns an Authentication (e.g., checks password, tokens).
* UserDetailsService is a DAO-like service that loads user-specific data (UserDetails) by username; commonly used by authentication providers.

**🔹 Authentication**

**How do you configure in-memory authentication?**  
Example (Java config):

@Bean

public UserDetailsService users() {

UserDetails user = User.withDefaultPasswordEncoder()

.username("user").password("pass").roles("USER").build();

return new InMemoryUserDetailsManager(user);

}

Or configure via security.user.\* properties for simple cases.

**How do you configure JDBC authentication?**  
Provide a DataSource and configure JdbcUserDetailsManager or JdbcDaoImpl with appropriate queries or use default schema.

**How do you implement custom authentication in Spring Security?**  
Implement AuthenticationProvider or a custom Filter that produces Authentication tokens. Wire it into the AuthenticationManager.

**How does UserDetails and UserDetailsService work?**  
UserDetails represents user info (username, password, authorities, account status). UserDetailsService loads UserDetails by username. AuthenticationProvider calls it to retrieve user and validate credentials.

**What is the purpose of PasswordEncoder?**  
Encodes and verifies passwords (hashing). Use secure algorithms like BCryptPasswordEncoder.

**What are common implementations of PasswordEncoder?**  
BCryptPasswordEncoder, SCryptPasswordEncoder, Pbkdf2PasswordEncoder, Argon2PasswordEncoder.

**How do you hash passwords using BCrypt?**  
new BCryptPasswordEncoder().encode(rawPassword);

verify with .matches(raw, encoded).

**How do you implement multi-factor authentication?**  
Use layered authentication: primary (password) then trigger second factor (TOTP via authenticator apps or SMS); implement via additional Filter that validates a second token before granting final Authentication.

**How do you implement form-based login?**  
Configure http.formLogin() in HttpSecurity, set login page, success/failure handlers, and parameters.

**How do you handle login failures and success redirects?**  
Configure success and failure handlers (AuthenticationSuccessHandler, AuthenticationFailureHandler) to control redirects/responses.

**🔹 Authorization & Roles**

**How do you restrict access to URLs based on roles?**  
In HttpSecurity, use authorizeHttpRequests() with antMatchers("/admin/\*\*").hasRole("ADMIN") or method-level security with @PreAuthorize("hasRole('ADMIN')").

**What is the difference between hasRole and hasAuthority?**  
hasRole("ADMIN") expects authority ROLE\_ADMIN (prefix added). hasAuthority("ADMIN\_PRIV") checks for exact granted authority.

**How do you secure methods using @PreAuthorize and @PostAuthorize?**  
Enable method security (@EnableMethodSecurity) and annotate methods with SpEL expressions like @PreAuthorize("hasRole('ADMIN')"). @PostAuthorize runs after method execution and can inspect return objects.

**What is the difference between URL-based and method-level security?**  
URL-based is enforced at the web layer (filters) controlling access to endpoints. Method-level security secures service layer methods even when invoked internally, guarding business operations.

**How do you implement role hierarchy?**  
Define RoleHierarchy bean (e.g., ROLE\_ADMIN > ROLE\_USER) and configure it into the ExpressionHandler used by security expressions.

**How do you prevent unauthorized access to REST endpoints?**  
Configure HttpSecurity to require authentication for endpoints and use @PreAuthorize for finer rules. Ensure tokens or sessions are validated in filters.

**How do you use @Secured annotation?**  
Annotate methods with @Secured("ROLE\_ADMIN") to restrict access. Enable with @EnableMethodSecurity(securedEnabled = true).

**How do you dynamically assign roles to users at runtime?**  
Load dynamic roles from DB and populate GrantedAuthority in UserDetails implementation at login or refresh token issuance.

**How do you restrict access to certain HTTP methods (GET, POST, DELETE)?**  
Match both URL and method in HttpSecurity config:

requestMatchers(HttpMethod.DELETE, "/users/\*\*").hasRole("ADMIN");

**How do you implement fine-grained access control in REST APIs?**  
Use method-level security with domain-specific checks (e.g., @PreAuthorize("@permissionEvaluator.canEdit(principal, #id)"))

or custom PermissionEvaluator.

**How do you create a REST API in Spring Boot?**

**To create a REST API in Spring Boot, I follow these steps:**

1. **Create a Spring Boot project** using Spring Initializr with the *Spring Web* dependency.
2. **Define a Model/Entity class** that represents the data structure, for example Student.
3. **Create a Repository interface** by extending JpaRepository or CrudRepository. Spring Data JPA automatically provides CRUD methods.
4. **Write a Service layer** (optional but recommended) that contains business logic and interacts with the repository.
5. **Create a REST Controller** using @RestController and map URLs using @RequestMapping, @GetMapping, @PostMapping, @PutMapping, and @DeleteMapping.
6. **Use @RequestBody, @PathVariable, @RequestParam** to receive input from the client.
7. **Run the Spring Boot application**, and the APIs will be available on localhost:8080

**What is @Autowired and how does it work?**

@Autowired is used in Spring to perform **automatic dependency injection**.  
When we annotate a field, constructor, or setter with @Autowired, Spring looks for a matching bean in the **ApplicationContext** and injects it automatically.

**What are the different types of Dependency Injection?**

Dependency Injection (DI) is a design pattern where an object’s dependencies are provided externally by the framework (Spring), instead of the object creating them itself.

In Spring, there are **3 main types of Dependency Injection**:

**⭐ 1. Constructor Injection (Recommended)**

Dependencies are provided through the class constructor.

@Service

public class StudentService {

private final StudentRepository repo;

public StudentService(StudentRepository repo) {

this.repo = repo;

}

}

**✔ Advantages:**

* Makes dependencies **immutable**
* Best for **mandatory** dependencies
* Supports **unit testing** easily
* Prevents **circular dependencies**

**✔ Most recommended by Spring team**

**⭐ 2. Setter Injection**

Dependencies are injected using setter methods.

@Service

public class StudentService {

private StudentRepository repo;

@Autowired

public void setRepo(StudentRepository repo) {

this.repo = repo;

}

}

**✔ When to use:**

* Dependency is **optional**
* You want to allow changing dependency at runtime

**⭐ 3. Field Injection**

Dependencies are injected directly into fields.

@Autowired

private StudentRepository repo;

**✔ Advantage:**

* Less code, easy to write

**❌ Disadvantages:**

* Not recommended for production
* Hard to test
* Makes class dependent on Spring container
* Does not support immutability

| **Annotation** | **Layer** | **Purpose** | **Special Feature** |
| --- | --- | --- | --- |
| **@Component** | Generic | Marks any class as a Spring bean | None |
| **@Service** | Service Layer | Business logic | Additional processing like AOP |
| **@Repository** | DAO Layer | Database operations | Exception translation |

**What is @Component in Spring?**

@Component is a Spring annotation used to mark a class as a **Spring-managed bean**.  
When a class is annotated with @Component, Spring automatically:

1. Detects it during **component scanning**
2. Creates an object (bean) of that class
3. Stores it in the **ApplicationContext (IoC container)**
4. Makes it available for **dependency injection.**

**MICROSERVICES**

**🌱 Beginner-Level**

**1. What is a microservice?**

Explain the concept, autonomy, decentralized data, independent deployment, etc.

**2. What are the advantages of microservices?**

Scalability, fault isolation, polyglot architecture, faster deployment.

**3. What are the challenges of microservices?**

Distributed complexity, network latency, data consistency, monitoring difficulty.

**4. How is a microservice different from a monolith?**

**5. What is REST, and why is it commonly used in microservices?**

**6. What tools/frameworks are used to build microservices in Java?**

Spring Boot, Spring Cloud, Micronaut, Quarkus, Helidon.

**7. What is the role of Spring Boot in microservices?**

**⚙️ Intermediate-Level Questions**

**🛠 Architecture & Design**

**8. What is service discovery?**

Discuss Netflix Eureka, Consul, Kubernetes DNS.

**9. What is an API gateway and why do we need one?**

Rate limiting, authentication, routing—examples: Zuul, Spring Cloud Gateway, Kong.

**10. What is circuit breaking?**

Explain resilience patterns using **Resilience4j**, Hystrix.

**11. Explain synchronous vs asynchronous communication in microservices.**

**12. How do you implement inter-service communication?**

REST, gRPC, messaging (Kafka, RabbitMQ).

**13. What is the Saga pattern?**

Used for distributed transactions—choreography vs orchestration.

**14. What is eventual consistency?**

How microservices achieve data consistency without 2PC.

**🔐 Security**

**15. How do you secure microservices?**

JWT, OAuth2, Keycloak, API Gateway auth.

**16. What is OAuth2? What is OpenID Connect?**

**17. How does token propagation work in distributed systems?**

**🗄 Data & Storage**

**18. Why does each microservice need its own database?**

**19. How do you handle transactions across multiple microservices?**

Saga pattern, event-driven systems.

**20. What is CQRS and when do you use it?**

**📊 Monitoring & Deployment**

**21. How do you monitor microservices?**

Prometheus, Grafana, Zipkin, Jaeger, Splunk.

**22. What is distributed tracing?**

Tracing a request across services—TraceId, SpanId.

**23. What is containerization? Why use Docker?**

**24. How does Kubernetes help microservices?**

Scaling, service discovery, orchestration.

**🔥 Advanced-Level Questions**

**🧠 Deep Architecture & Patterns**

**25. What is Domain-Driven Design and how does it relate to microservices?**

**26. How do you decompose a monolith into microservices?**

**27. How do you design idempotent APIs?**

**28. How do you avoid the distributed monolith anti-pattern?**

**29. How do you handle API versioning in microservices?**

**🌀 Resilience & Performance**

**30. What is backpressure and how do you handle it?**

**31. How do you design services for high availability?**

**32. What is bulkheading in microservices?**

**33. What are sidecar patterns?**

(Service mesh—Istio, Linkerd.)

**🗳 Messaging & Events**

**34. Why choose Kafka over REST for communication?**

**35. What is event sourcing, and when is it helpful?**

**36. How do you guarantee message delivery?**

At-least-once, at-most-once, exactly-once semantics.

**☄️ Hands-On / Coding Questions**

**37. How do you implement circuit breakers using Resilience4j in Spring Boot?**

**38. How do you write a REST controller in Spring Boot?**

**39. Show a simple example of service discovery with Eureka.**

**40. How do you configure centralized logging across microservices?**