Java/Spring boot

Here is the **complete Java + Spring Boot interview question bank**, structured for **1–3 years experience** — covering fundamentals, OOPs, Collections, Exceptions, Spring Boot, REST APIs, JPA/Hibernate, Microservices, Security, and real-world scenarios.

🔥 JAVA + SPRING BOOT — FULL QUESTION BANK

📌 1. Java Core Basics

✅ 1) What is Java and why is it platform independent?

◼️ Explanation (simple, conceptual)

Java is an **object-oriented, high-level programming language** designed to be secure, portable, and robust.

It follows the **WORA principle — Write Once, Run Anywhere**.

Java achieves this using **bytecode** and **JVM**.

◼️ When/Why is it used

- Cross-platform development
- Enterprise applications
- Spring Boot backend applications
- Android apps
- Huge ecosystem and libraries

◼️ Why Java is platform independent? (MOST ASKED)

Because Java code is **not compiled into OS-specific machine code**.

Instead:

```
Java code → Compiler → Bytecode (.class) → JVM → Machine code
```

Each OS has its own JVM implementation.

So the same bytecode runs anywhere as long as a JVM exists.

■ Example

You compile `Hello.java` on Windows → run the same `.class` file on Linux → works.

■ Short summary line

Java is platform independent because JVM executes bytecode uniformly across all operating systems.

✅ 2) JDK vs JRE vs JVM

■ Explanation (simple, conceptual)

These 3 components form the Java runtime environment.

🔵 JDK (Java Development Kit)

- Contains tools to **develop** Java apps
- Includes: compiler (`javac`), debugger, JRE
- Required by developers

🔵 JRE (Java Runtime Environment)

- Used to **run** Java apps
- Contains JVM + libraries
- No compiler

🔵 JVM (Java Virtual Machine)

- Executes **bytecode**
- Converts bytecode → machine code
- Platform dependent implementation
- Manages memory (garbage collection)

■ **When/Why used**

- JDK → writing and compiling code
- JRE → only running code
- JVM → underlying engine executing bytecode

■ **Example**

To run a Spring Boot app: **requires JRE/JVM**

To write and compile code: **requires JDK**

■ **Short summary line**

JDK = develop; JRE = run; JVM = execute bytecode.

✅ 3) What is bytecode?

■ **Explanation (simple, conceptual)**

Bytecode is the **intermediate, platform-neutral code** generated by the Java compiler.

It is stored in `.class` files.

JVM interprets or JIT-compiles this bytecode into machine code.

■ **When/Why used**

- Allows Java to be platform independent
- Enables security (JVM sandbox)
- Allows optimization during runtime
- Helps portability across servers, OSes, and devices

■ **Example**

```
javac Hello.java  → generates Hello.class (bytecode)
java Hello        → JVM executes bytecode
```

Bytecode looks like:

```
0xCAFEBABE...
```

⬛ **Short summary line**

Bytecode is platform-independent code that JVM executes on any OS.

✅ 4) What are data types in Java?

⬛ **Explanation (simple, conceptual)**

Data types define **what kind of data** a variable can store.

Java has **two categories**:

1. **Primitive types**
2. **Reference types**

⬛ **When/Why used**

- To define variable memory size
- To ensure type-safety
- To avoid runtime type errors

🔵 Primitive Data Types (8 types)

Type	Size	Example
byte	1 byte	age = 25
short	2 bytes	count
int	4 bytes	id = 101
long	8 bytes	phone number
float	4 bytes	12.5f
double	8 bytes	12.556
char	2 bytes	'A'
boolean	1 bit	true/false

🟣 Reference Data Types

- Arrays

- Classes
- Objects
- Interfaces
- Strings

  Stored on heap, reference stored on stack.

■ Example

```
int x = 10;           // primitive
String name = "John";  // reference
```

■ Short summary line

Java has 8 primitive data types and reference types like objects, arrays, and interfaces.

✅ 5) What is a Class & Object?

■ Explanation (simple, conceptual)

Class

A blueprint/template that defines **properties (variables)** and **behaviors (methods)**.

Object

An instance of a class created in memory.

■ When/Why used

- To represent real-world entities (User, Product, Expense)
- To apply OOP principles like abstraction, encapsulation
- To enable reusable, modular code

■ Example

```
class Car {
    String color;
```

```
    void drive() {
        System.out.println("Car is driving");
    }
}

Car c = new Car();  // object
c.color = "Red";
c.drive();
```

■ Short summary line

Class is a blueprint; object is a real instance of that blueprint created at runtime.

✅ 6) What is static keyword used for?

■ Explanation (simple, conceptual)

`static` means the member belongs to the **class**, not the object.

- Loaded once in memory (Class Loader loads static area)
- Shared across all objects
- Can be accessed without creating object

It can be applied to:

- variables
- methods
- blocks
- nested classes

■ When/Why is it used?

- When you want a common value shared across all objects
- Utility methods (Math, Logger)
- To reduce memory usage
- For constants
- For helper classes

■ **Example**

```
class Counter {
    static int count = 0;

    Counter() { count++; }
}

System.out.println(Counter.count); // shared
```

■ **Short summary line**

static is used for members that should belong to the class itself and be shared across all objects.

✅ 7) What is this vs super keyword?

■ **Explanation (simple, conceptual)**

🔵 this keyword

Refers to **current class object**.

Used to access:

- current object variables
- current object methods
- constructors (this())

🟣 super keyword

Refers to **parent class object**.

Used to access:

- parent variables
- parent methods
- parent constructor (super())

■ **When/Why used?**

this

- To differentiate between class variable and method parameter
- To call another constructor in same class

super

- To reuse parent functionality
- To call parent constructor first

■ **Example**

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

class Child extends Parent {
    int x = 20;

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

■ **Short summary line**

this refers to current object; super refers to parent object.

✅ 8) What is final variable, method, class?

■ **Explanation (simple, conceptual)**

`final` means **cannot be changed**.

✔ final variable → constant (cannot be reassigned)

✔ final method → cannot be overridden

✔️ final class → cannot be inherited (no subclass)

◼️ **When/Why used**

final variable

- Used for constants
- Enforces immutability

final method

- To prevent method overriding
- For security (banking apps)

final class

- To prevent inheritance
- Used for utility classes (String class, Math class)

◼️ **Example**

```
final int MAX = 100;        // variable

final void display() { }     // method

final class Car { }          // class
```

◼️ **Short summary line**

final restricts modification—variable can't change, method can't override, class can't extend.

✅ 9) Why String is immutable?

◼️ **Explanation (simple, conceptual)**

String is immutable in Java because **its value cannot be changed once created**.

◼️ **When/Why is it immutable? (Major interview point)**

✔️ 1. Security

Credentials in URLs, class loading, configuration values must not change.

✔️ 2. String pooling

Immutability enables Java to store strings in **String constant pool**, improving memory.

✔️ 3. Performance & caching

Hashcode is cached → efficient for HashMap keys.

✔️ 4. Thread safety

Immutable objects are naturally thread-safe → no synchronization required.

■ Example

```
String s = "Hello";
s = s + "Java";
// creates new string "HelloJava"; old "Hello" stays in pool
```

■ Short summary line

String is immutable for security, string pool optimization, caching, and thread safety.

✅ 10) Difference between String, StringBuilder & StringBuffer

■ Explanation (simple, conceptual)

These classes handle text manipulation.

🔵 String

- Immutable
- Every modification creates new object
- Slow for heavy string operations

🟣 StringBuilder

- Mutable
- **Not thread-safe**
- Fastest for single-threaded operations

🔴 StringBuffer

- Mutable
- **Thread-safe (synchronized)**
- Slower than StringBuilder, faster than String

■ When/Why used

Type	When to use	Reason
String	Fixed or small immutable text	safe + pooled
StringBuilder	Fast string operations	no sync overhead
StringBuffer	Concurrent string modifications	thread-safe

■ Example

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

■ Short summary line

String = immutable, StringBuilder = fastest mutable, StringBuffer = thread-safe mutable.

📌 2. OOPs in Java (Top Asked)

✅ 1) What is OOP? Why do we need it?

■ Explanation (simple, conceptual)

OOP (Object-Oriented Programming) is a programming model that organizes software into **objects** that contain **data (variables)** and **behavior (methods)**.

Java is built on OOP principles to create modular, reusable systems.

■ **When/Why is it needed?**

✔ To manage complex applications easily

✔ To reuse code through inheritance

✔ To protect data using encapsulation

✔ To design flexible, maintainable systems

✔ To model real-world entities (User, Product, Order)

✔ To support polymorphism → one interface, many implementations

---

■ **Example (Simple Java OOP model)**

```
class Car {
    void start() { }
    void stop() { }
}
Car c = new Car();
```

■ **Short summary line**

OOP helps structure applications using objects, making systems reusable, scalable, and maintainable.

✅ 2) Encapsulation — Real Example

■ **Explanation (simple, conceptual)**

Encapsulation means **hiding internal data** and providing controlled access through getters/setters.

It protects the data and enforces validation.

■ **When/Why is it needed?**

✔ Prevents unauthorized access

✔ Ensures data validation

✔️ Achieves data hiding

✔️ Improves maintainability

This is widely used in **Java beans, DTOs, entities**.

■ Real Project Example (Java + Spring Boot)

User password should never be publicly accessible:

```
class User {
    private String password;   // hidden field

    public void setPassword(String password) {
        if(password.length() >= 8)
            this.password = password;
    }
}
```

Here:

- `password` is hidden from outside
- Only valid values are set → **data protection**

■ Short summary line

Encapsulation hides internal data and exposes controlled access using getters/setters.

✅ 3) Abstraction — How implemented in Java?

■ Explanation (simple, conceptual)

Abstraction hides complex internal implementation and exposes only the required features.

In Java, abstraction is achieved using:

✔️ **Abstract classes**

✔️ **Interfaces**

They show **what** a class can do, but hide **how** it is done.

■ **When/Why used?**

✔ To hide internal logic from caller

✔ To simplify design

✔ To enforce common structure across classes

✔ To separate interface from implementation

■ **Example (Java Abstraction with Interface)**

```
interface PaymentService {
    void pay(int amount);
}

class UpiPayment implements PaymentService {
    public void pay(int amount) {
        // UPI payment logic hidden
    }
}
```

Caller only knows:

```
paymentService.pay(1000);
```

Not **how** payment works internally.

■ **Short summary line**

Abstraction in Java is implemented using abstract classes and interfaces to hide internal implementation.

✅ 4) What is an Interface?

■ **Explanation (simple, conceptual)**

An interface is a **contract** that defines methods but does not provide implementation (until Java 8).

Classes that implement it must provide method bodies.

■ **When/Why used?**

✔️ To achieve loose coupling

✔️ For multiple implementations (Strategy pattern)

✔️ For abstraction

✔️ For dependency injection in Spring Boot

✔️ Supports multiple inheritance of type

■ **Example**

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

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

■ **Short summary line**

Interface defines a contract; implementing classes provide the actual behavior.

✅ 5) Abstract Class vs Interface

■ **Explanation (simple, conceptual)**

Both provide abstraction, but they differ in capabilities.

🔵 Abstract Class

- Can have **abstract + non-abstract methods**
- Can have **fields, constructors**
- Supports single inheritance
- Can have method body

🟣 Interface

- Only method signatures (before Java 8)
- From Java 8: can have **default & static methods**
- Cannot have constructors
- Supports **multiple inheritance**
- Variables are **public static final** by default

■ **When/Why used**

✔ Use Abstract Class when:

- You need shared code for all subclasses
- Common variables required
- Partial implementation needed

✔ Use Interface when:

- You need 100% abstraction
- You want multiple implementations
- You want to use dependency injection (Spring Boot)

■ **Example**

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

interface Flyable {
    void fly();
}
```

■ **Short summary line**

Abstract class provides partial abstraction; interface provides full abstraction and supports multiple inheritance.

✅ 6) Inheritance — single / multi-level / multiple?

■ Explanation (simple, conceptual)

Inheritance allows one class to acquire the **properties and behaviors** of another class.

It supports code reuse and better structure.

Types of inheritance in Java:

🔵 1. Single Inheritance

One parent → one child.

🔵 2. Multilevel Inheritance

Grandparent → Parent → Child (chain)

🔵 3. Hierarchical Inheritance

One parent → multiple children

🔴 4. Multiple Inheritance (NOT allowed with classes)

A class cannot extend two classes simultaneously.

■ When/Why used

- To reuse existing code
- To extend behavior
- To implement parent-child hierarchies
- To support polymorphism

■ Example

```
class A { }
class B extends A { }      // Single

class C extends B { }      // Multilevel
```

Interfaces allow multiple inheritance:

```
interface X {}
interface Y {}
class Test implements X, Y {}
```

■ **Short summary line**

Java supports single, multilevel, and hierarchical inheritance; multiple inheritance only via interfaces.

✅ 7) Why Java doesn't support multiple inheritance?

■ **Explanation (simple, conceptual)**

Java avoids multiple inheritance of classes to prevent **ambiguity and complexity**.

The main problem is the **Diamond Problem**, where the compiler cannot decide which parent version of a method to call.

■ **When/Why this matters**

- Maintains clarity
- Avoids method conflict
- Keeps inheritance tree simple
- Encourages use of interfaces instead

■ **Example – Diamond Problem**

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

If B & C both have method `show()`, and D extends both:

→ Java wouldn't know which `show()` to inherit.

Hence Java prohibits:

```
class D extends B, C  // ❌ not allowed
```

But using interfaces:

```
interface B { void show(); }
interface C { void show(); }

class D implements B, C {
   public void show() {}
}
```

■ **Short summary line**

Java avoids multiple class inheritance to prevent ambiguity (Diamond Problem); interfaces solve this safely.

✅ 8) Polymorphism — Method Overloading vs Overriding

■ **Explanation (simple, conceptual)**

Polymorphism = one name, multiple behaviors.

Two types:

🔵 Method Overloading (Compile-time Polymorphism)

Same method name, different parameter list.

✔ Happens inside **same class**

✔ Decided at **compile time**

✔ Used for flexibility

🟣 Method Overriding (Runtime Polymorphism)

Child class provides new implementation for parent's method.

✔ Happens across **parent-child classes**

✔ Decided at **runtime**

✔ Enables dynamic dispatch

■ **When/Why used**

Overloading:

- For readability
- For multiple ways to call same method

  (ex: print(int), print(String))

Overriding:

- For runtime behavior change
- For polymorphic calls
- Essential in frameworks (Spring Boot beans)

■ **Example**

```
// Overloading
void add(int a, int b) {}
void add(double a, double b) {}

// Overriding
class Parent { void show(){ } }
class Child extends Parent {
    void show(){ }   // overriding
}
```

■ **Short summary line**

Overloading = same method name, different parameters; overriding = child modifies parent method at runtime.

✅ 9) What is runtime polymorphism?

■ **Explanation (simple, conceptual)**

Runtime polymorphism means the **method to be executed is decided at runtime**, not compile-time.

Achieved using **method overriding** + **upcasting** (parent reference → child object).

⬛ When/Why used

- Late binding
- API design
- Flexible behavior
- Used heavily in frameworks like Spring (Bean injection)

⬛ Example

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

Animal a = new Dog();
a.sound();  // Bark → decided at runtime
```

⬛ Short summary line

Runtime polymorphism occurs when overriding lets the JVM decide method implementation at runtime.

✅ 10) Access Modifiers in Java

⬛ Explanation (simple, conceptual)

Access modifiers control **visibility** of classes, variables, and methods.

⬛ When/Why used

- To secure data
- To control encapsulation
- To restrict unwanted access
- To enforce proper API design

🔵 public

Accessible everywhere.

🔵 protected

Accessible within package + subclasses.

🔵 default *(no keyword)*

Accessible only within same package.

🔵 private

Accessible only within the class.

■ Example

```
public class A {
    private int x;       // only inside A
    protected int y;     // subclass + same package
    int z;               // package-private
    public int k;        // everywhere
}
```

■ Short summary line

Java has four access levels: private → default → protected → public (least to most visible).

📌 3. Memory, Garbage Collection

✅ 1) Stack Memory vs Heap Memory

■ Explanation (simple, conceptual)

Java divides memory into two main areas:

🔵 Stack Memory

- Stores **local variables**, method parameters, and function call frames
- Follows **LIFO** (Last In First Out)

- Very fast
- Thread-specific (each thread gets its own stack)

🟣 Heap Memory

- Stores **objects**, arrays, and reference types
- Shared by all threads
- Managed by **Garbage Collector**
- Slower compared to stack

⬛ When/Why used

Stack:

- For short-lived data
- Temporary variables
- Method calls

Heap:

- For long-lived data
- Objects created using `new`
- Shared resources

⬛ Example

```
void test() {
    int x = 10;       // stack
    Student s = new Student();  // object stored in heap, 's' reference in stac
k
}
```

⬛ Short summary line

Stack stores method-level data; heap stores objects and is managed by the Garbage Collector.

✅ 2) How Garbage Collection works?

⬛ Explanation (simple, conceptual)

Garbage Collection (GC) automatically finds and removes **unused objects** from heap memory to free space.

Java uses a **mark-and-sweep** algorithm:

✔️ 1. Mark Phase

GC identifies reachable (alive) objects via root references.

✔️ 2. Sweep Phase

Unreachable objects are deleted and memory is reclaimed.

⬛ When/Why used

- Prevent memory leaks
- Improve performance
- Automatically manage memory so developer does not manually free memory

---

⬛ How GC decides object is unused

If no reference points to an object → considered *garbage*.

⬛ Example (simple)

```
Student s1 = new Student();
s1 = null;    // eligible for garbage collection
```

⬛ Short summary line

Garbage Collector removes unreachable heap objects using mark-and-sweep to free memory automatically.

✅ 3) What is finalize()?

⬛ Explanation (simple, conceptual)

`finalize()` is a method in Object class that is called **before an object is garbage collected**.

But it is **deprecated in Java 9+** because it is unreliable.

■ **When/Why used** (OLD Approach)

- To release resources before GC
- To close files, streams, network connections

But now replaced by:

✔ try-with-resources

✔ AutoCloseable

✔ Cleaners

■ **Example**

```
@Override
protected void finalize() throws Throwable {
    System.out.println("Object is being garbage collected");
}
```

■ **Short summary line**

finalize() was called before GC for cleanup, but is deprecated because it is unpredictable.

✅ 4) WeakReference vs SoftReference

■ **Explanation (simple, conceptual)**

Java provides special references besides strong references to help GC manage memory efficiently.

🔵 Strong Reference

Normal reference → **not eligible for GC** until reference is removed.

```
Student s = new Student();  // strong reference
```

🟣 SoftReference

- GC clears it **only when memory is low**
- Used for **caching**
- Object survives longer

```
SoftReference<Student> ref = new SoftReference<>(new Student());
```

🔴 WeakReference

- GC clears it **as soon as no strong reference exists**
- Used for memory-sensitive applications (WeakHashMap)

```
WeakReference<Student> ref = new WeakReference<>(new Student());
```

■ When/Why used

Type	When we use	Why
SoftReference	Cache of images, heavy objects	Only cleared in low memory
WeakReference	WeakHashMap keys, listeners	Cleared quickly to avoid memory leaks

---

■ Example (WeakHashMap)

```
Map<Key, Value> map = new WeakHashMap<>();
// Keys can be GCed automatically
```

■ Short summary line

SoftReference lasts until memory is needed; WeakReference is cleared immediately when unreferenced—useful for caches and avoiding memory leaks.

📌 4. Collections Framework

✅ 1) What are Collections in Java?

⬛ Explanation (simple, conceptual)

Collections in Java provide **predefined data structures** for storing and manipulating groups of objects like lists, sets, and maps.

They are part of `java.util` package and include:

- Interfaces (List, Set, Map)
- Implementations (ArrayList, HashMap, HashSet)
- Utility classes (Collections, Arrays)

⬛ When/Why used

✔️ Replace array limitations (fixed size, no built-in methods)

✔️ Dynamic sizing

✔️ Searching, sorting, iteration

✔️ Supports generic types

✔️ Widely used in real applications

⬛ Example

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

⬛ Short summary line

Collections framework provides dynamic, efficient data structures like List, Set, and Map for storing and managing data.

✅ 2) List vs Set vs Map difference

⬛ Explanation (simple, conceptual)

Feature	List	Set	Map
Stores	Ordered elements	Unique elements	Key-value pairs
Allows duplicates	✔️ Yes	❌ No	Keys no, values yes

Feature	List	Set	Map
Index-based	✔️ Yes	❌ No	N/A
Examples	ArrayList, LinkedList	HashSet, TreeSet	HashMap, TreeMap

■ **When/Why used**

✔️ List

- Ordered data
- Duplicate allowed
- Indexed access
- Example: users list, product list

✔️ Set

- No duplicates required
- Fast lookup
- Example: storing unique emails, IDs

✔️ Map

- Key-value storage
- Fast retrieval by key
- Example: caching, configuration map

■ **Example**

```
List<String> l = new ArrayList<>();
Set<String> s = new HashSet<>();
Map<Integer, String> m = new HashMap<>();
```

■ **Short summary line**

List = ordered & duplicates, Set = unique elements, Map = key-value pairs.

✅ 3) ArrayList vs LinkedList

⬛ Explanation (simple, conceptual)

Both implement **List**, but their internal structures differ.

🔵 ArrayList

- Backed by **dynamic array**
- Fast for **get()**, slow for insertion in middle
- Better for read-heavy operations

🟣 LinkedList

- Backed by **doubly linked list**
- Fast for insertion/deletion at head or middle
- Slower for random access (no index)

⬛ When/Why used

Use Case	Choose	Reason
Frequent reads	ArrayList	O(1) random access
Frequent insert/delete	LinkedList	O(1) add/remove
Memory efficient	ArrayList	Less overhead
Queue/Deque	LinkedList	Has addFirst, addLast

⬛ Example

```
List<String> arrayList = new ArrayList<>();
List<String> linkedList = new LinkedList<>();
```

⬛ Short summary line

ArrayList is fast for reads; LinkedList is fast for insertions/deletions.

✅ 4) HashMap vs TreeMap vs LinkedHashMap

⬛ Explanation (simple, conceptual)

🔵 HashMap

- Unordered
- Fastest lookup (O(1))
- Uses hashing

🟣 LinkedHashMap

- Maintains **insertion order**
- Slightly slower than HashMap
- Good for predictable iteration

🔴 TreeMap

- Sorted map (keys sorted)
- Uses Red-Black Tree
- O(log n) performance

■ When/Why used

Type	When to Use	Reason
HashMap	Best performance	Fast lookup
LinkedHashMap	Need insertion order	Maintains order
TreeMap	Need sorted keys	Automatically sorts

■ Example

```
Map<Integer, String> hash = new HashMap<>();
Map<Integer, String> linked = new LinkedHashMap<>();
Map<Integer, String> tree = new TreeMap<>();
```

■ Short summary line

HashMap = fastest, LinkedHashMap = ordered, TreeMap = sorted.

✅ 5) HashSet vs TreeSet

■ Explanation (simple, conceptual)

🔵 HashSet

- Stores **unique** elements
- **Unordered**
- Uses **HashMap internally**
- Operations: O(1) average time

🔴 TreeSet

- Stores **unique + sorted** elements
- Uses **Red-Black Tree**
- Operations: O(log n)

■ When/Why used

Use Case	Choose	Reason
Fast lookup	HashSet	O(1) speed
Need sorted set	TreeSet	Automatic sorting
Large data	HashSet	Better performance
Range queries (headset, tailset)	TreeSet	Tree-based structure

---

■ Example

```
Set<Integer> hs = new HashSet<>();
Set<Integer> ts = new TreeSet<>();
```

■ Short summary line

HashSet = unique + fast; TreeSet = unique + sorted.

✅ 6) How HashMap works internally?

■ Explanation (simple, conceptual)

HashMap stores key-value pairs using:

- **Array of buckets**
- **LinkedList / Balanced Tree** inside buckets

- Uses `hashCode()` + `equals()` to locate keys

■ When/Why used

- Fast search
- Caching
- Storing key-value data
- Widely used in Spring Boot and JPA internals

■ Internal Steps (Interview-Ready)

✓ 1. hashCode() is calculated

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

✓ 2. Index = hash % bucket_size

✓ 3. Bucket matched → LinkedList or TreeNode used

- If few collisions → LinkedList
- If many collisions → Tree (Red-Black Tree)

✓ 4. equals() is used

To confirm the correct key.

■ Example

```
Map<String, Integer> map = new HashMap<>();
map.put("John", 25);
```

Steps:

- hash("John") → bucket index
- Store Entry(key="John", value=25)
- Retrieval uses same hash → equals()

■ Short summary line

HashMap uses hashCode + equals to store entries in buckets with LinkedList/Tree for collision handling.

✅ 7) Fail-fast vs Fail-safe iterators

⬛ Explanation (simple, conceptual)

🔵 Fail-fast Iterator

- Throws **ConcurrentModificationException** if structure changes during iteration
- Works on original collection

Examples:

- ArrayList iterator
- HashMap iterator

🔴 Fail-safe Iterator

- Does **not** throw exception
- Works on a **copy** of the collection
- Modifications do NOT affect iteration

Examples:

- ConcurrentHashMap
- CopyOnWriteArrayList

⬛ When/Why used

Use Case	Iterator Type	Reason
Multi-threaded environment	Fail-safe	No exception
Performance-critical	Fail-fast	Faster, no copying
Safe concurrent updates	Fail-safe	Works on snapshot

⬛ Example

Fail-fast:

```
Iterator it = list.iterator();
list.add(10); // ConcurrentModificationException
```

Fail-safe:

```
ConcurrentHashMap<Integer, String> m = new ConcurrentHashMap<>();
for(var e : m.entrySet()) m.put(2, "x"); // no exception
```

■ Short summary line

Fail-fast throws error on modification; fail-safe works on a copy and avoids exceptions.

✅ 8) Comparable vs Comparator

■ Explanation (simple, conceptual)

🔵 Comparable

- Natural sorting
- Implemented inside the class
- Method: `compareTo()`
- Affects original class ordering

🔴 Comparator

- External sorting logic
- Define multiple sorting strategies
- Method: `compare()`

■ When/Why used

Use Case	Choose	Reason
Class has natural order	Comparable	One default sorting
Multiple sorting criteria	Comparator	Name, Age, Salary, etc.
Avoid modifying class	Comparator	External logic

⬛ Example

Comparable:

```
class Student implements Comparable<Student> {
    int age;
    public int compareTo(Student s) {
        return this.age - s.age;
    }
}
```

Comparator:

```
Comparator<Student> sortByName =
    (a, b) → a.name.compareTo(b.name);
```

⬛ Short summary line

Comparable gives natural order inside a class; Comparator gives custom orders externally.

✅ 9) Synchronized collection vs ConcurrentHashMap

⬛ Explanation (simple, conceptual)

🔵 Synchronized Collections

- Provided by `Collections.synchronizedMap()` , `Vector` , `Hashtable`
- Entire collection locked → **slower**
- One thread at a time

🔴 ConcurrentHashMap

- Modern concurrency feature
- Uses **segment locking / bucket-level locking**
- Multiple threads can read/write simultaneously
- No locking during read

- Very fast

⬛ When/Why used

Use Case	Choose	Reason
High concurrency	ConcurrentHashMap	Fine-grained locking
Legacy code	Synchronized Map	Simple but slow
Frequent reads	ConcurrentHashMap	Non-blocking reads

⬛ Example

Synchronized Map:

```
Map<Integer, String> syncMap = Collections.synchronizedMap(new HashMap<>());
```

ConcurrentHashMap:

```
ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();
```

⬛ Short summary line

Synchronized collections block entire map; ConcurrentHashMap uses fine-grained locking for high performance.

📌 5. Exception Handling

✅ 1) What is an exception?

⬛ Explanation (simple, conceptual)

An exception is an **unexpected event** that disrupts normal program execution during **runtime**.

Examples:

- Divide by zero
- Null pointer access

- File not found
- Database connection failure

Exceptions are objects of type `Throwable`.

⬛ When/Why is it used

✔️ To prevent application crashes

✔️ To handle errors gracefully

✔️ To show meaningful messages

✔️ To maintain normal flow

✔️ To separate error logic from regular code

⬛ Example

```
int x = 10 / 0;  // ArithmeticException
```

⬛ Short summary line

Exception is a runtime error event handled using try-catch to prevent application breakdown.

✅ 2) Checked vs Unchecked exceptions

⬛ Explanation (simple, conceptual)

🔵 Checked Exceptions

- Checked at **compile-time**
- Must be handled using **try-catch** or **throws**
- Come from `Exception` class (except RuntimeException)

Examples:

- IOException
- SQLException
- FileNotFoundException

🔴 Unchecked Exceptions

- Occur at **runtime**
- Not checked by compiler
- Come from `RuntimeException`

Examples:

- NullPointerException
- ArithmeticException
- IndexOutOfBoundsException

⬛ When/Why used

Type	When used	Why
Checked	Expected issues (files, DB)	Must handle
Unchecked	Programming bugs	Fix code

⬛ Example

```
// Checked
FileReader f = new FileReader("data.txt"); // must handle

// Unchecked
int x = 10 / 0; // ArithmeticException
```

⬛ Short summary line

Checked = compiler checks; Unchecked = runtime errors.

✅ 3) throw vs throws difference

⬛ Explanation (simple, conceptual)

🔵 throw

- Used to **manually throw** an exception
- Inside method

```
throw new IllegalArgumentException("Invalid input");
```

🔴 throws

- Used in method signature
- Says that method **may throw** an exception
- Delegates responsibility to caller

```
void read() throws IOException { }
```

◼️ When/Why used

Keyword	Why used
throw	To throw custom or specific exception
throws	To avoid handling inside method

◼️ Short example

```
void check(int age) {
    if(age < 18) throw new RuntimeException("Not allowed");
}

void readFile() throws IOException {
    FileReader fr = new FileReader("a.txt");
}
```

◼️ Short summary line

throw throws an exception; throws declares that method may throw exception.

✅ 4) finally block execution rules

◼️ Explanation (simple, conceptual)

`finally` block executes **ALWAYS**, regardless of exception—used for cleanup.

■ When/Why used

✔ To close files, DB connections, streams

✔ To release resources

✔ To ensure cleanup logic always runs

■ Execution Rules

finally executes even when:

✔ try has exception

✔ try has no exception

✔ catch executes

✔ return statement used

✔ break/continue used

✔ exception re-thrown

finally **does NOT** execute when:

❌ JVM shuts down

❌ System.exit() is called

❌ Power failure

■ Example

```
try {
    int x = 10 / 0;
} catch(Exception e) {
    System.out.println("Error");
} finally {
    System.out.println("Always executed");
}
```

■ Short summary line

finally block always runs for cleanup, except in severe JVM shutdown situations.

✅ 5) Custom exception class

⬛ Explanation (simple, conceptual)

When built-in exceptions are not meaningful, we create **custom exceptions** extending `Exception` or `RuntimeException`.

⬛ When/Why used

✔ Business-specific errors

✔ Clearer error messages

✔ Domain validation

✔ Cleaner API responses

Used in banking, payment, user validation scenarios.

⬛ Example

```
class InsufficientBalanceException extends Exception {
    public InsufficientBalanceException(String msg) {
        super(msg);
    }
}
```

Usage:

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

⬛ Short summary line

Custom exceptions provide domain-specific error handling with meaningful messages.

✅ 6) try-with-resources

⬛ Explanation (simple, conceptual)

try-with-resources automatically **closes resources** that implement `AutoCloseable`, such as:

- FileReader
- BufferedReader
- Connection
- PreparedStatement

It avoids memory leaks.

■ When/Why used

✔ To auto-close files and streams

✔ No need for finally block

✔ Reduces boilerplate

✔ Prevents resource leak exceptions

■ Example

```
try (BufferedReader br = new BufferedReader(new FileReader("a.txt"))) {
    System.out.println(br.readLine());
}
// br automatically closed here
```

■ Short summary line

try-with-resources automatically closes resources without needing a finally block.

📌 6. Multithreading & Concurrency

✅ 1) Process vs Thread

■ Explanation (simple, conceptual)

🔵 Process

- Independent program in execution
- Has its **own memory space**
- Has code, data, heap, stack

- Heavyweight

🔴 Thread

- Lightweight unit of a process
- Shares memory of process (heap/code)
- Has its own **stack + program counter**
- Faster to create & switch

◼️ When/Why used

Concept	Why used
Process	Running independent apps (Chrome, VS Code)
Thread	Multitasking inside a process (multiple tabs in Chrome)

◼️ Example

```
// Process → Java application itself
// Thread → main thread + worker threads
```

◼️ Short summary line

Process = independent program; Thread = lightweight sub-task sharing memory inside process.

✅ 2) Runnable vs Thread class

◼️ Explanation (simple, conceptual)

🔵 Runnable (Interface)

- Defines `run()` method
- Used when you want to **share same object** among threads
- Better for OOP design
- Preferred in real projects

🔴 Thread (Class)

- Extends Thread class → cannot extend any other class
- Directly override `run()`
- Less flexible

■ When/Why used

Use Case	Choose	Reason
Need to extend another class	Runnable	Multiple-inheritance not supported
Want flexibility	Runnable	Better design
Simple quick thread	Thread	Less boilerplate

■ Example

Runnable:

```
class Task implements Runnable {
    public void run() {
        System.out.println("Running");
    }
}

new Thread(new Task()).start();
```

Thread:

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

new Task().start();
```

■ Short summary line

Runnable is preferred for reusability; Thread class is simple but less flexible.

✅ 3) Thread States

■ Explanation (simple, conceptual)

A thread goes through multiple lifecycle states:

🔵 1. NEW

Thread created but not started

→ `new Thread()`

🔵 2. RUNNABLE

Ready to run / running

→ After `start()`

🔵 3. BLOCKED

Waiting for a monitor lock

→ Enter synchronized block

🔵 4. WAITING

Waiting indefinitely for another thread

→ `wait()` , `join()` , `park()`

🔵 5. TIMED_WAITING

Waiting for a specified time

→ `sleep(2000)` , `wait(2000)`

🔵 6. TERMINATED

Thread completes execution

■ When/Why used

- Helps debug multithreading issues
- Used in monitoring, logs, profiling

■ Example

```
Thread.sleep(2000); // TIMED_WAITING
```

⬛ Short summary line

Thread lifecycle: NEW → RUNNABLE → BLOCKED/WAITING → TERMINATED.

✅ 4) synchronized keyword

⬛ Explanation (simple, conceptual)

`synchronized` ensures that **only one thread** executes a block/method at a time.

It provides **mutual exclusion** and **thread safety**.

⬛ When/Why used

✔ Prevent race conditions

✔ Protect shared resources

✔ Ensure consistent data updates

✔ Critical in banking, transactions, counters

⬛ Example

```
synchronized void increment() {
    count++;
}
```

Or block:

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

When a thread enters a synchronized block, **other threads are blocked** until it exits.

⬛ Short summary line

synchronized provides thread safety by allowing only one thread to access critical code at a time.

✅ 5) volatile keyword

⬛ Explanation (simple, conceptual)

`volatile` ensures a variable's value is **read directly from main memory**, not thread-local cache.

It guarantees:

✔ Visibility

✔ Prevents caching

✔ Prevents instruction reordering

But **does NOT provide atomicity**.

⬛ When/Why used

Use volatile when:

- Multiple threads read/write a shared variable
- Value changes frequently
- No complex operations needed

Typical use:

✔ flags

✔ status variables

✔ stop signals

⬛ Example

```
volatile boolean running = true;

void stop() { running = false; }
```

All threads will see updated value immediately.

⬛ Short summary line

volatile ensures visibility (reads from main memory), but does NOT make operations atomic.

✅ 6) ExecutorService — Use Case

⬛ Explanation (simple, conceptual)

`ExecutorService` is a framework that manages and executes threads efficiently using a **thread pool**, instead of manually creating threads.

It abstracts:

- thread creation
- thread scheduling
- task execution

⬛ When/Why is it used?

✔ To run tasks in parallel

✔ To reuse threads → reduce overhead

✔ To avoid creating too many threads

✔ To manage asynchronous tasks

✔ Widely used in:

- API calls
- background jobs
- batch processing
- microservices

⬛ Example

```
ExecutorService service = Executors.newFixedThreadPool(5);

service.submit(() → {
   System.out.println("Task executed by: " + Thread.currentThread().getNa
me());
});

service.shutdown();
```

⬛ Short summary line

ExecutorService manages thread creation using thread pools for efficient and scalable parallel execution.

✅ 7) Deadlock — How to prevent?

■ Explanation (simple, conceptual)

A deadlock occurs when **two or more threads wait forever** for each other's locks, and none can proceed.

This happens when thread A waits for thread B, and thread B waits for thread A.

■ When/Why it occurs

- Multiple locks acquired in different order
- Poor synchronization design
- Nested synchronized blocks

■ How to prevent deadlock (Interview-Ready)

✔ 1. **Lock ordering**

Always acquire locks in the **same sequence**.

✔ 2. **Use tryLock() instead of synchronized**

Timeout-based locking avoids infinite waiting.

✔ 3. **Avoid nested synchronized blocks**

Keep critical section small.

✔ 4. **Use higher-level concurrency tools**

- `ConcurrentHashMap`
- `Semaphore`
- `ReentrantLock`

✔ 5. **Avoid unnecessary locks**

■ Example (Deadlock scenario)

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

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

■ **Short summary line**

Deadlock occurs when threads wait forever for each other's locks; prevent it via lock ordering, timeouts, or avoiding nested locks.

✅ 8) Future & Callable

■ **Explanation (simple, conceptual)**

🔵 Callable

- Like Runnable but returns a **value**
- Can throw **checked exceptions**
- Method: `call()`

🔵 Future

- Represents the **result of an asynchronous computation**
- Provides methods:
    - `get()` (wait and return result)
    - `isDone()`
    - `cancel()`

■ **When/Why used**

✔ When you need a return value from a thread

✔ When tasks are long-running

✔ When you want to check completion status

✔ Used heavily in async programming & microservices

■ **Example**

```
Callable<Integer> task = () -> 10 + 20;

ExecutorService executor = Executors.newFixedThreadPool(2);

Future<Integer> result = executor.submit(task);

System.out.println(result.get());  // prints 30

executor.shutdown();
```

■ **Short summary line**

Callable returns a value; Future retrieves it asynchronously.

✅ 9) Thread Pool Benefits

■ **Explanation (simple, conceptual)**

Thread pool is a group of pre-created threads managed by the JVM.

ExecutorService internally uses thread pools.

■ **Benefits (Interview-Ready)**

✔ 1. **Improved performance**

Reuses threads → avoids cost of creating/destroying threads repeatedly.

✔ 2. **Avoids system overload**

Predefined number of threads → prevents "too many threads" problem.

✔ 3. **Better resource management**

JVM controls:

- thread lifecycle
- scheduling
- queuing

✔ 4. **Supports async execution**

Long-running tasks executed without blocking the main thread.

✔️ 5. **Scalable & efficient**

Ideal for production systems with high concurrency.

■ Example

```
ExecutorService pool = Executors.newFixedThreadPool(10);
pool.submit(new Task());
```

■ Short summary line

Thread pools boost performance by reusing threads, preventing overload, and enabling scalable concurrent execution.

📌 7. Java 8 Features (Very Important)

✅ 1) What is Stream API?

■ Explanation (simple, conceptual)

Stream API is a Java 8 feature that allows **functional-style operations on collections** (map, filter, reduce).

It processes data in a **pipeline** without modifying the original collection.

✔️ Works on data **streams**

✔️ Supports **lazy evaluation**

✔️ Supports **parallel processing**

■ When/Why used

- For clean, readable code
- To avoid loops and boilerplate
- To process large datasets efficiently
- For filtering, mapping, grouping, sorting operations

■ Example

```
List<Integer> list = Arrays.asList(1,2,3,4);

list.stream()
    .filter(x → x % 2 == 0)
    .forEach(System.out::println);  // prints 2, 4
```

■ **Short summary line**

Stream API enables functional-style, pipeline-based processing of collections.

✅ 2) map(), filter(), sorted(), collect()

■ **Explanation (simple, conceptual)**

🔵 filter()

Selects elements that match a condition.

🔵 map()

Transforms each element into another type/value.

🔵 sorted()

Sorts the stream data.

🔵 collect()

Converts stream back to List, Set, Map, etc.

■ **When/Why used**

✔ To transform lists

✔ To filter unwanted items

✔ To sort data

✔ To convert processed stream to a collection

■ **Example**

```
List<String> names = Arrays.asList("John","Steve","Adam");

List<String> result = names.stream()
    .filter(n -> n.startsWith("A"))
    .map(String::toUpperCase)
    .sorted()
    .collect(Collectors.toList());
```

■ Short summary line

filter = selection, map = transformation, sorted = ordering, collect = convert back to collection.

✅ 3) Lambda Expressions

■ Explanation (simple, conceptual)

Lambda is a **shorter way to write anonymous methods**.

It enables functional programming in Java 8.

Syntax:

```
(parameters) -> expression/body
```

■ When/Why used

✔ To simplify code

✔ To remove boilerplate anonymous classes

✔ Used with Stream API

✔ Used in functional interfaces

■ Example

Without Lambda:

```
Runnable r = new Runnable() {
    public void run() { System.out.println("Hello"); }
};
```

With Lambda:

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

⬛ **Short summary line**

Lambda expressions provide a concise way to implement functional interfaces.

✅ 4) Functional Interfaces

⬛ **Explanation (simple, conceptual)**

A functional interface contains **exactly one abstract method**.

Used with lambda expressions.

Examples:

- Runnable
- Callable
- Comparator
- Function
- Predicate
- Supplier
- Consumer

⬛ **When/Why used**

✔ Enables lambda expressions

✔ Supports functional programming

✔ Simplifies callback logic

⬛ **Example**

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

```
Calculator c = (a,b) → a + b;
```

⬛ **Short summary line**

Functional interfaces have one abstract method and are the base for lambda expressions.

✅ 5) Optional class

⬛ **Explanation (simple, conceptual)**

`Optional` is a container that may or may not hold a value.

Helps avoid **NullPointerException**.

⬛ **When/Why used**

✔ Avoid null checks everywhere

✔ Improve readability

✔ Safe return type from methods

✔ Better error handling

⬛ **Example**

```
Optional<String> name = Optional.ofNullable(getName());

name.ifPresent(System.out::println);

String defaultName = name.orElse("Unknown");
```

⬛ **Short summary line**

Optional avoids NullPointerException by handling missing values safely.

✅ 6) Default & Static methods in interface

⬛ **Explanation (simple, conceptual)**

Java 8 allows interfaces to have:

🔵 default methods

- Provide method implementation
- Can be overridden
- Used to add new features without breaking old code

🔵 static methods

- Belong to the interface
- Cannot be overridden
- Used for utility methods

■ When/Why used

✔ To evolve interfaces without breaking implementations

✔ To define helper methods

■ Example

```
interface Vehicle {
    default void start() {
        System.out.println("Vehicle started");
    }

    static void service() {
        System.out.println("Vehicle servicing");
    }
}
```

■ Short summary line

Default methods add behavior to interfaces; static methods provide shared utilities.

✅ 7) Method Reference

■ Explanation (simple, conceptual)

A method reference is a **shorter form of a lambda** when a method already exists.

Types:

- Static method reference → `Class::method`
- Instance method reference → `obj::method`
- Constructor reference → `Class::new`

⬛ **When/Why used**

✔️ To simplify lambda expressions

✔️ To reuse existing methods

✔️ To improve readability

⬛ **Example**

Lambda:

```
list.forEach(x → System.out.println(x));
```

Method reference:

```
list.forEach(System.out::println);
```

⬛ **Short summary line**

Method reference is a shortcut to use existing methods instead of writing lambdas.

🚀 SPRING + SPRING BOOT

📌 8. Spring Boot Basics

✅ 1) What is Spring Boot?

⬛ **Explanation (simple, conceptual)**

Spring Boot is a framework built on top of Spring that makes it easy to create **production-ready applications** with **minimal configuration**.

It provides:

✔ Auto-configuration

✔ Embedded servers (Tomcat/Jetty)

✔ Opinionated defaults

✔ Ready-to-use starters

■ **When/Why used**

- To develop REST APIs quickly
- To avoid manual Spring configuration
- Auto-wiring and auto-setup
- Easy deployment with embedded server
- Faster development for microservices

■ **Example**

Create a REST API with just:

```
@RestController
public class HelloController {
   @GetMapping("/hello")
   public String hello() { return "Hello"; }
}
```

■ **Short summary line**

Spring Boot simplifies Spring development using auto-config, starters, and embedded servers.

✅ 2) Difference between Spring & Spring Boot

■ **Explanation (simple, conceptual)**

Feature	Spring Framework	Spring Boot
Setup	Requires manual configuration	Auto-configured
Server	Need to deploy WAR	Comes with embedded Tomcat
Dependencies	Must add individually	Comes with starters
XML	XML/Java config	Mostly annotations
Speed	Slower to set up	Fast development

■ When/Why used

- Spring Boot is preferred for **REST APIs, microservices, cloud apps**
- Spring is used for older, enterprise, legacy systems

■ Example

Spring → Need to configure DataSource manually

Spring Boot → Auto-configures DataSource using properties

■ Short summary line

Spring Boot is Spring + auto-config + starters + embedded server for faster development.

✅ 3) What are starters?

■ Explanation (simple, conceptual)

Starters are **predefined dependency bundles** provided by Spring Boot.

They group common dependencies into a single one.

Examples:

- `spring-boot-starter-web`
- `spring-boot-starter-data-jpa`
- `spring-boot-starter-security`

■ When/Why used

✔ Reduce dependency confusion

✔ Ensure compatible versions

✔ One dependency = many libraries included

■ **Example**

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</artifact>
```

This includes:

Tomcat + Jackson + Spring MVC

■ **Short summary line**

Starters are ready-made dependency bundles for faster Spring Boot setup.

✅ 4) Auto-Configuration in Spring Boot

■ **Explanation (simple, conceptual)**

Auto-configuration automatically configures Spring beans **based on classpath and properties**.

Triggered by:

```
@EnableAutoConfiguration
```

Spring Boot scans:

- Dependencies
- application.properties
- Bean definitions

...and configures required beans automatically.

■ **When/Why used**

✔ Reduce boilerplate

✔ No need of XML or manual config

✔ Automatically configures DataSource, JPA, MVC, Security etc.

■ **Example**

Add MySQL driver → Spring Boot auto-configures:

- DataSource
- Connection pool
- Transaction manager

⬛ **Short summary line**

Auto-configuration sets up Spring components automatically based on classpath & configs.

✅ 5) application.properties vs application.yml

⬛ **Explanation (simple, conceptual)**

🔵 application.properties

- Key=value format
- Simple, flat configuration

🟣 application.yml

- YAML format (hierarchical)
- More readable
- Better for nested structures

⬛ **When/Why used**

Use Case	Choose	Reason
Simple values	properties	Easy typing
Complex configs (DB, security)	yml	Cleaner hierarchy
Microservices	yml	Preferred by Spring Cloud

⬛ **Example**

properties

```
server.port=8080
spring.datasource.url=jdbc:mysql://localhost:3306/test
```

yml

```
server:
  port: 8080
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/test
```

⬛ Short summary line

properties = key-value; yml = hierarchical and more readable.

✅ 6) What is Spring Boot Actuator?

⬛ Explanation (simple, conceptual)

Actuator provides **production monitoring endpoints** to check application health and metrics.

Endpoints like:

- `/actuator/health`
- `/actuator/info`
- `/actuator/metrics`
- `/actuator/loggers`

⬛ When/Why used

✔️ Monitor microservices

✔️ Health checks for Kubernetes / load balancers

✔️ Performance metrics

✔️ Debugging issues in production

⬛ Example (Dependency)

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

■ **Short summary line**

Actuator provides monitoring, metrics, and health endpoints for production-ready apps.

✅ 7) What is @SpringBootApplication?

■ **Explanation (simple, conceptual)**

`@SpringBootApplication` is a meta-annotation that includes:

✔ `@Configuration` – defines configuration class

✔ `@EnableAutoConfiguration` – enables auto-config

✔ `@ComponentScan` – scans components/beans

■ **When/Why used**

✔ Main entry point of Spring Boot

✔ Replaces multiple annotations

✔ Automatically scans packages

■ **Example**

```java
@SpringBootApplication
public class MyApp {
    public static void main(String[] args) {
        SpringApplication.run(MyApp.class, args);
    }
}
```

■ **Short summary line**

@SpringBootApplication = @Configuration + @EnableAutoConfiguration + @ComponentScan.

📌 9. Spring Core + DI

✅ 1) What is IOC (Inversion of Control)?

◼️ Explanation (simple, conceptual)

IOC is a design principle where **control of object creation and dependency management is given to the framework**, not the developer.

In Spring:

- The **container** creates objects (beans)
- The **container** injects dependencies
- The **container** manages the lifecycle

◼️ When/Why used

✔️ Avoids manual object creation (`new`)

✔️ Reduces tight coupling

✔️ Centralizes object configuration

✔️ Makes applications easier to test (mock dependencies)

◼️ Example

Without IoC:

```
Service s = new Service();   // developer creates object
```

With IoC:

```
@Service
class Service { }
```

Spring creates and manages the object.

◼️ Short summary line

IOC means Spring, not the developer, controls object creation and dependency management.

✅ 2) What is Dependency Injection?

◼️ Explanation (simple, conceptual)

Dependency Injection (DI) is a pattern where **dependencies are provided from outside**, rather than the class creating them.

Spring performs DI automatically.

◼️ When/Why used

✔️ Avoids tightly coupled code

✔️ Improves testability (use mocks)

✔️ Makes components reusable

✔️ Simplifies configuration

◼️ Example

```
@Service
class OrderService {
    private final PaymentService paymentService;

    @Autowired
    public OrderService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }
}
```

◼️ Short summary line

DI injects required objects into a class instead of creating them manually.

✅ 3) Types of DI: Constructor vs Setter

◼️ Explanation (simple, conceptual)

🔵 Constructor Injection

- Dependencies provided via constructor
- Best practice & recommended

- Makes object immutable
- Helps testing (mandatory dependencies)

🔵 Setter Injection

- Injected through setter methods
- For optional dependencies
- Less preferred

■ When/Why used

Type	Use Case	Reason
Constructor DI	Mandatory dependencies	Safe & test-friendly
Setter DI	Optional dependencies	Flexible but less safe

■ Example

Constructor:

```
@Autowired
public UserService(Repo repo) { this.repo = repo; }
```

Setter:

```
@Autowired
public void setRepo(Repo repo) { this.repo = repo; }
```

■ Short summary line

Constructor DI = recommended; Setter DI = optional dependencies.

✅ 4) @Component vs @Service vs @Repository

■ Explanation (simple, conceptual)

All three mark classes as **Spring-managed beans**, but differ by purpose.

🔵 @Component

Generic stereotype for any Spring bean.

🔵 @Service

Used for **business logic layer** classes.

Provides better readability & semantic meaning.

🔵 @Repository

Used for **DAO / database layer**.

Also provides automatic exception translation.

■ When/Why used

✔ To organize application layers

✔ To help Spring handle them differently

✔ For cleaner architecture

■ Example

```
@Component
class Util { }

@Service
class UserService { }

@Repository
class UserRepository { }
```

■ Short summary line

@Component = generic bean; @Service = business logic; @Repository = data access with exception translation.

✅ 5) @Autowired — Field vs Constructor Injection

■ Explanation (simple, conceptual)

@Autowired tells Spring to **inject the required dependency**.

🔴 Field Injection (Not recommended)

```
@Autowired
private UserService service;
```

Problems: not testable, not immutable.

🔵 Constructor Injection (Recommended)

```
private final UserService service;

@Autowired
public Controller(UserService service) {
    this.service = service;
}
```

✔️ Best practice

✔️ Supports unit testing

✔️ Supports immutability

✔️ Prevents null dependencies

■ When/Why used

- Constructor DI is recommended for production code
- Field DI sometimes used in quick demos or legacy code

■ Short summary line

Constructor injection is preferred; field injection is discouraged for testability and immutability.

✅ 6) What is Bean Life Cycle?

■ Explanation (simple, conceptual)

Spring manages the entire lifecycle of a bean from creation to destruction.

■ Lifecycle Steps (Interview-Ready)

✔ 1. Bean Instantiation

Object creation.

✔ 2. Dependency Injection

Wiring dependencies.

✔ 3. Bean Post Processors

`postProcessBeforeInitialization()`

✔ 4. Initialization

`@PostConstruct` or `InitializingBean.afterPropertiesSet()`

✔ 5. Ready to Use

✔ 6. Destruction

`@PreDestroy` or `DisposableBean.destroy()`

■ Example

```
@Component
class TestBean {

    @PostConstruct
    public void init() {
        System.out.println("Bean initialized");
    }

    @PreDestroy
    public void destroy() {
        System.out.println("Bean destroyed");
    }
}
```

■ Short summary line

Bean lifecycle: create → inject → initialize → use → destroy.

✅ 7) Prototype vs Singleton Bean Scope

■ **Explanation (simple, conceptual)**

🔵 Singleton (Default)

- Only **one instance** per Spring container
- Shared globally

🔴 Prototype

- New bean instance **every time requested**
- Not managed fully by Spring (no destroy method)

■ **When/Why used**

Scope	When used	Why
Singleton	Services, repositories	Thread-safe, shared instance
Prototype	Stateful objects	New object per request

■ **Example**

```
@Scope("singleton")
class A {}

@Scope("prototype")
class B {}
```

■ **Short summary line**

Singleton = one shared instance; Prototype = new instance on each request.

📌 10. REST API with Spring Boot

✅ 1) @RestController vs @Controller

■ **Explanation (simple, conceptual)**

🔵 @Controller

- Used for **web MVC applications**
- Returns **View (HTML/JSP/Thymeleaf)**
- Usually paired with `@ResponseBody` for JSON

🔴 @RestController

- Used for **REST APIs**
- Combines `@Controller + @ResponseBody`
- Returns **JSON/XML directly** instead of view

⬛ When/Why used

Use Case	Annotation	Reason
Return HTML	@Controller	Renders UI pages
Return JSON for APIs	@RestController	Auto-converts objects to JSON

⬛ Example

```
@RestController
class UserApi {
    @GetMapping("/user")
    public User getUser() { return new User("John"); }
}
```

```
@Controller
class HomeController {
    @GetMapping("/home")
    public String home() { return "home.html"; }
}
```

⬛ Short summary line

@Controller returns views; @RestController returns JSON for REST APIs.

✅ 2) @GetMapping, @PostMapping, @PutMapping, @DeleteMapping

■ **Explanation (simple, conceptual)**

These annotations handle HTTP methods in REST APIs.

- **@GetMapping** → Fetch data
- **@PostMapping** → Create new data
- **@PutMapping** → Update existing data
- **@DeleteMapping** → Remove data

■ **When/Why used**

✔ Follow RESTful design

✔ Clean, readable API code

✔ Automatically map to correct HTTP method

■ **Example**

```
@GetMapping("/users")
public List<User> getAll() { ... }

@PostMapping("/users")
public User create(@RequestBody User user) { ... }

@PutMapping("/users/{id}")
public User update(@PathVariable int id, @RequestBody User user) { ... }

@DeleteMapping("/users/{id}")
public void delete(@PathVariable int id) { ... }
```

■ **Short summary line**

Mapping annotations define CRUD operations using RESTful HTTP methods.

✅ 3) RequestParam vs PathVariable

■ **Explanation (conceptual)**

🔵 @RequestParam

- Used for **query parameters**
- Optional or fixed key-value pairs
- Example: `/search?name=John`

🔴 @PathVariable

- Used for **dynamic URL values**
- Example: `/users/10`

⬛ When/Why used

Use Case	Choose	Example
Filters/search	RequestParam	`/users?role=admin`
Identify resource	PathVariable	`/users/5`

⬛ Example

```
@GetMapping("/search")
public String search(@RequestParam String name) { ... }

@GetMapping("/user/{id}")
public String getUser(@PathVariable int id) { ... }
```

⬛ Short summary line

RequestParam = query parameter; PathVariable = URL path parameter.

✅ 4) RequestBody vs ResponseBody

⬛ Explanation (simple, conceptual)

🔵 @RequestBody

- Converts **JSON → Java object**
- Used for POST/PUT requests

🔴 @ResponseBody

- Converts **Java object → JSON**

- Automatically included in @RestController

■ **When/Why used**

✔ For receiving JSON input

✔ For returning JSON output

✔ For REST APIs

■ **Example**

```
@PostMapping("/add")
public User save(@RequestBody User user) {
    return user;
}
```

■ **Short summary line**

RequestBody reads JSON input; ResponseBody sends JSON output.

✅ 5) What is ResponseEntity?

■ **Explanation (simple, conceptual)**

`ResponseEntity` represents the **entire HTTP response**, including:

✔ Body

✔ Status code

✔ Headers

It gives full control over output.

■ **When/Why used**

Use when you need:

- Custom HTTP status
- Custom headers
- Error handling
- Standard API responses

■ **Example**

```
@GetMapping("/user")
public ResponseEntity<User> getUser() {
    return ResponseEntity
        .status(HttpStatus.OK)
        .body(new User("John"));
}
```

Error example:

```
return ResponseEntity.status(HttpStatus.NOT_FOUND).body("User not foun
d");
```

■ **Short summary line**

ResponseEntity gives full control over response body, status code, and headers.

✅ 6) Status Codes in REST API

■ **Explanation (simple, conceptual)**

HTTP Status Codes indicate the **result** of a client's API request.

Most commonly used categories:

🔵 2xx — Success

- **200 OK** → Successful GET
- **201 Created** → Successful POST
- **204 No Content** → Successful DELETE/PUT without response body

🔵 4xx — Client Errors

- **400 Bad Request** → Invalid input/validation error
- **401 Unauthorized** → Missing/invalid authentication
- **403 Forbidden** → Authentication OK but access denied
- **404 Not Found** → Resource not found
- **409 Conflict** → Duplicate data

🔵 5xx — Server Errors

- **500 Internal Server Error**
- **503 Service Unavailable**

⬛ When/Why used

✔ Clear communication between client & server

✔ Standardized responses

✔ Helps debugging & monitoring

✔ API consumers depend on correct status codes

⬛ Example

```
return ResponseEntity.status(HttpStatus.CREATED).body(user);
```

⬛ Short summary line

Status codes show whether API request succeeded, failed, or caused server/client errors.

✅ 7) DTO — Why use DTO instead of entity?

⬛ Explanation (simple, conceptual)

DTO (**Data Transfer Object**) is used to transfer data between client and server.

It is **not** tied to the database table.

Entities = Database representation

DTO = API representation

⬛ When/Why used

✔ **Security** — hide sensitive fields (password, createdDate, role)

✔ **Avoid exposing DB structure**

✔ **Validation** — better request validation

✔ **Custom response shaping**

✔️ **Decoupling —** changes in DB should not break API

⬛ **Example**

❌ Bad API (exposes entity)

```
@Entity
class UserEntity {
  private int id;
  private String password; // exposed!
}
```

✔️ Good API (uses DTO)

```
class UserDTO {
  private int id;
  private String name;
}
```

⬛ **Short summary line**

DTO protects entity structure, improves security, and gives clean API models.

✅ 8) How to validate API request?

⬛ **Explanation (simple, conceptual)**

Spring Boot uses **Bean Validation** (Hibernate Validator) with annotations like:

- `@NotNull`
- `@Size`
- `@Email`
- `@Min`
- `@Pattern`

Apply on DTO fields + use `@Valid` in controller.

⬛ **When/Why used**

✔️ To validate request data

✔️ To avoid manual validation in controller

✔️ To prevent invalid data entering DB

✔️ To return meaningful validation errors

■ **Example**

DTO

```
public class UserRequest {

    @NotBlank(message = "Name is required")
    private String name;

    @Email(message = "Invalid email")
    private String email;

    @Min(18)
    private int age;
}
```

Controller

```
@PostMapping("/users")
public ResponseEntity<?> createUser(@Valid @RequestBody UserRequest
req) {
    return ResponseEntity.ok("Created");
}
```

Spring automatically returns:

```
400 Bad Request
{ "email": "Invalid email" }
```

■ **Short summary line**

Validation is done using Bean Validation annotations + @Valid in controller.

✅ 9) Exception Handling using @ControllerAdvice

⬛ Explanation (simple, conceptual)

`@ControllerAdvice` is a **global exception handler** in Spring.

It centralizes all error handling in a single class instead of writing try-catch in controllers.

⬛ When/Why used

✔️ Consistent error responses

✔️ No repeated try-catch blocks

✔️ Cleaner controller code

✔️ Common format for all exceptions

✔️ Better logging and debugging

⬛ Example

Global Exception Handler

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<?> handleNotFound(ResourceNotFoundException
ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
                             .body(ex.getMessage());
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<?> handleGeneric(Exception ex) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                             .body("Something went wrong");
    }
}
```

Used with custom exception:

```
if(user == null)
    throw new ResourceNotFoundException("User not found");
```

⬛ **Short summary line**

@ControllerAdvice provides centralized, consistent exception handling for the entire application.

📌 11. Spring Boot + JPA / Hibernate

✅ 1) What is JPA? Why ORM?

⬛ **Explanation (simple, conceptual)**

JPA (**Java Persistence API**) is a specification that defines how Java objects interact with relational databases.

It is not an implementation — Hibernate is the most common implementation.

ORM (**Object Relational Mapping**) maps **Java objects ↔ database tables** automatically.

⬛ **When/Why used**

✔ Avoid writing boilerplate JDBC code

✔ Avoid manual SQL for CRUD

✔ Convert Java objects to DB rows easily

✔ Cleaner, maintainable code

✔ Standard way to map entities

⬛ **Example**

```
@Entity
class User {
    @Id
    private int id;
```

```
    private String name;
}
```

JPA maps class → table, fields → columns.

⬛ Short summary line

JPA is a standard for ORM; ORM avoids manual SQL by mapping Java objects to DB tables.

✅ 2) Entity & @Table annotations

⬛ Explanation (simple, conceptual)

`@Entity` marks a class as a JPA-managed persistent object.

`@Table` specifies the table name in the database.

⬛ When/Why used

✔ To map Java class to DB table

✔ To specify custom table name

✔ Required for all JPA persistence operations

⬛ Example

```
@Entity
@Table(name = "users")  // optional
public class User {
    @Id
    private int id;

    private String name;
}
```

⬛ Short summary line

@Entity marks class for ORM; @Table customizes the DB table mapping.

✅ 3) @OneToOne, @OneToMany, @ManyToMany relations

■ **Explanation (simple, conceptual)**

🔵 @OneToOne

One entity relates to exactly one other entity.

Example: User ↔ Profile

🔵 @OneToMany

One entity relates to many others.

Example:

User → List

🔵 @ManyToMany

Many entities relate to many others (via join table).

Example:

Students ↔ Courses

■ **When/Why used**

✔ To define relationships between tables

✔ Automatically manage joining, cascading

✔ Cleaner object structure

■ **Examples**

One-to-One

```
@OneToOne
@JoinColumn(name = "profile_id")
private Profile profile;
```

One-to-Many

```
@OneToMany(mappedBy = "user")
private List<Order> orders;
```

Many-to-Many

```
@ManyToMany
@JoinTable(
    name = "student_course",
    joinColumns = @JoinColumn(name="student_id"),
    inverseJoinColumns = @JoinColumn(name="course_id")
)
private List<Course> courses;
```

■ Short summary line

JPA relations model real-world table relationships like one-to-one, one-to-many, and many-to-many.

✅ 4) Lazy vs Eager Fetching

■ Explanation (simple, conceptual)

🔵 Lazy (default for collections)

Data is loaded **only when accessed**.

Better performance.

🔴 Eager (default for @ManyToOne, @OneToOne)

Loads related data **immediately**, even if not needed.

■ When/Why used

Fetch Type	Use Case	Reason
Lazy	Large collections	Performance friendly
Eager	Small relationships	Avoid extra queries

■ Example

```
@OneToMany(fetch = FetchType.LAZY)
private List<Order> orders;
```

```
@OneToOne(fetch = FetchType.EAGER)
private Profile profile;
```

■ **Performance Tip (Interview Gold)**

✔ Use **Lazy** by default

✔ Avoid Eager unless absolutely required

✔ Eager can cause *N+1 query problem*

■ **Short summary line**

Lazy loads data when needed; Eager loads immediately — Lazy is preferred for performance.

✅ 5) PagingAndSortingRepository vs JpaRepository

■ **Explanation (simple, conceptual)**

🔵 PagingAndSortingRepository

- Adds **pagination** and **sorting**
- Methods: `findAll(Pageable p)`, `findAll(Sort s)`

🔵 JpaRepository

- Extends PagingAndSortingRepository
- Adds full CRUD + batch operations
- Best for most projects
- Most commonly used

■ **When/Why used**

Feature	PagingAndSortingRepository	JpaRepository
Pagination	✔ Yes	✔ Yes
Sorting	✔ Yes	✔ Yes
CRUD	Basic	Full
Batch operations	No	Yes
Most used	❌	✔✔✔

■ Example

Paging:

```
Page<User> users = repo.findAll(PageRequest.of(0, 10));
```

JpaRepository:

```
public interface UserRepo extends JpaRepository<User, Integer> { }
```

■ Short summary line

JpaRepository is a superset providing full CRUD + pagination + batch operations; PagingAndSortingRepo provides only paging/sorting.

✅ 6) findBy methods in Spring JPA

■ Explanation (simple, conceptual)

Spring Data JPA allows creating query methods **just by naming convention**.

Spring interprets the method name and creates the SQL automatically.

Examples:

- `findByName(String name)`
- `findByEmailAndStatus(String email, String status)`
- `findByAgeGreaterThan(int age)`

■ When/Why used

✔ Reduce boilerplate SQL

✔ No need to write JPQL manually

✔️ Highly readable method names

✔️ Faster development

⬛ **Example**

```
public interface UserRepo extends JpaRepository<User, Integer> {
    List<User> findByName(String name);
    User findByEmail(String email);
    List<User> findByAgeGreaterThan(int age);
}
```

Spring converts these to queries automatically.

⬛ **Short summary line**

Spring JPA generates SQL automatically based on method naming conventions like findBy, readBy, getBy.

✅ 7) Transaction Management — @Transactional

⬛ **Explanation (simple, conceptual)**

`@Transactional` ensures that a block of code executes in a **single database transaction**.

Features:

- Commit on success
- Rollback on exception
- Ensures data consistency

⬛ **When/Why used**

✔️ Multi-step DB operations

✔️ Save + update + delete in same method

✔️ Avoid partial data updates

✔️ Prevent data corruption

⬛ **Example**

```
@Service
public class UserService {

    @Transactional
    public void updateUser(User user) {
        repo.save(user);
        logRepo.save(new Log("User updated")); // both succeed or roll back
    }
}
```

If any line fails, entire transaction rolls back.

■ **Short summary line**

@Transactional ensures all DB operations run atomically — success commits, failure rolls back.

✅ 8) Change Tracking vs Dirty Checking

■ **Explanation (simple, conceptual)**

🔵 Change Tracking

Hibernate tracks all loaded entity objects.

It knows original values + current values.

🔵 Dirty Checking

Before committing, Hibernate checks if any tracked entity has **changed**.

If yes → it automatically generates **UPDATE** statements.

■ **When/Why used**

✔ Automatically update only changed fields

✔ No need to call `repo.save()` repeatedly

✔ Reduces boilerplate

■ **Example**

```
User u = repo.findById(1).get();
u.setName("John Updated");
// No save required manually
```

On transaction commit, Hibernate executes:

```
UPDATE user SET name='John Updated' WHERE id=1;
```

■ **Short summary line**

Change tracking monitors loaded entities; Dirty checking automatically updates modified fields on commit.

✅ 9) HQL vs JPQL difference

■ **Explanation (simple, conceptual)**

🔵 HQL (Hibernate Query Language)

- Proprietary to Hibernate
- Works with Hibernate-specific features
- Entity-oriented

🔵 JPQL (Java Persistence Query Language)

- Standard JPA query language
- Works with any JPA provider (Hibernate, EclipseLink)
- Also entity-oriented

■ **When/Why used**

✔ Use JPQL in JPA-based projects

✔ Use HQL only when needing Hibernate-specific behavior

■ **Example**

JPQL:

```
@Query("SELECT u FROM User u WHERE u.name = :name")
List<User> findByName(@Param("name") String name);
```

HQL:

```
Query q = session.createQuery("FROM User WHERE name = :name");
```

■ Key Differences

Feature	HQL	JPQL
Standard	❌ Hibernate-only	✔ JPA standard
Portability	Low	High
Usage	Hibernate Session	JPA EntityManager

■ Short summary line

JPQL is standard JPA query language; HQL is Hibernate's proprietary version with extra features.

📌 12. Spring Security

✅ 1) What is Spring Security?

■ Explanation (simple, conceptual)

Spring Security is a powerful framework used to secure Spring Boot applications. It provides:

✔ Authentication (who are you?)

✔ Authorization (what can you access?)

✔ Password encryption

✔ Filters for request security

✔ Protection against attacks (CSRF, XSS, Session Fixation)

It integrates seamlessly with Spring Boot using auto-configuration.

■ **When/Why used**

- To secure REST APIs
- To restrict access based on user roles
- To authenticate users with DB/JWT/OAuth
- To protect endpoints with security filters
- To handle login/logout securely

■ **Example**

```
http
  .authorizeHttpRequests()
  .requestMatchers("/admin").hasRole("ADMIN")
  .anyRequest().authenticated()
  .and()
  .httpBasic();
```

■ **Short summary line**

Spring Security handles authentication, authorization, and application-level security in Spring Boot.

✅ 2) Basic Auth vs Token Auth

■ **Explanation (simple, conceptual)**

🔵 Basic Authentication

- Credentials (username & password) sent with **every request**
- Encoded in Base64 (not encrypted)
- Stateless
- Simple but less secure

🔴 Token Authentication (e.g., JWT)

- Client receives a token after login
- Sends token on each request

- No need to send credentials
- More secure, scalable, modern

■ When/Why used

Type	When Used	Why
Basic Auth	Internal apps, testing	Simple, no setup
Token Auth	REST APIs, mobile apps, microservices	Scalable, secure, stateless

---

■ Short summary line

Basic Auth sends credentials each request; Token Auth sends a secure token instead.

✅ 3) JWT Authentication Flow

■ Explanation (simple, conceptual)

JWT (**JSON Web Token**) is a stateless token used for authentication in APIs.

■ Flow (Interview-ready)

✔ 1. Client sends login request

`POST /login { username, password }`

✔ 2. Server verifies credentials

✔ 3. Server generates JWT

Token contains:

- userId
- roles
- expiry

✔ 4. Client stores token (LocalStorage/SessionStorage)

✔ 5. For each request, client sends token

`Authorization: Bearer <token>`

✔ 6. Server validates token on every request

No DB call needed.

⬛ Example JWT Structure

```
header.payload.signature
```

⬛ Short summary line

JWT is stateless authentication where client sends a signed token with each request.

✅ 4) Filters in Spring Security

⬛ Explanation (simple, conceptual)

Spring Security uses a **chain of filters** before requests reach controllers.

Important filters:

- `UsernamePasswordAuthenticationFilter`
- `BasicAuthenticationFilter`
- `JwtAuthenticationFilter` (custom)
- `SecurityContextPersistenceFilter`
- `ExceptionTranslationFilter`

⬛ When/Why used

✔ Validate authentication before hitting controller

✔ Add custom token/jwt validation

✔ Log requests

✔ Reject unauthorized requests early

⬛ Example (Custom JWT Filter)

```
public class JwtFilter extends OncePerRequestFilter {
    protected void doFilterInternal(HttpServletRequest req, ... ) {
        // read token, validate, set authentication
```

```
    }
}
```

Registered in SecurityConfig:

```
http.addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.clas
s);
```

⬛ **Short summary line**

Filters intercept requests, validate authentication, and enforce security rules before controllers.

✅ 5) CSRF Token Usage

⬛ **Explanation (simple, conceptual)**

CSRF (Cross-Site Request Forgery) token protects against unauthorized form submissions.

- Server generates hidden CSRF token
- Client must send it back with state-changing requests
- Prevents attackers from forging requests on user's behalf

⬛ **When/Why used**

✔ Enabled for web apps using stateful sessions

✔ Not recommended for stateless REST APIs (usually disabled)

⬛ **Example**

```
http.csrf().disable();  // for REST APIs
```

Or enable:

```
http.csrf().csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyF
alse());
```

⬛ **Short summary line**

CSRF token protects web forms; usually disabled for stateless REST APIs.

✅ 6) Role-based Access Control (RBAC)

◼️ **Explanation (simple, conceptual)**

RBAC restricts API access based on user roles (ADMIN, USER, MANAGER).

◼️ **When/Why used**

✔️ To control access to sensitive APIs

✔️ To restrict functionality

✔️ To enforce security policies

◼️ **Example**

In SecurityConfig:

```
http.authorizeHttpRequests()
    .requestMatchers("/admin/**").hasRole("ADMIN")
    .requestMatchers("/user/**").hasAnyRole("USER", "ADMIN")
    .anyRequest().authenticated();
```

In JWT token:

```
roles: ["ADMIN", "USER"]
```

◼️ **Short summary line**

RBAC protects APIs by granting access only to users with specific roles.

📌 13. Spring Boot Microservices

✅ 1) What is Microservice?

◼️ **Explanation (simple, conceptual)**

A microservice is a small, independent, deployable service that owns a **single business capability**.

Characteristics:

- Independently deployable
- Own database per service
- Lightweight communication (REST/Message queue)
- Autonomously scalable

■ **When/Why used**

✔ Large applications broken into manageable services

✔ Independent deployment without affecting other services

✔ Better scalability — scale only required services

✔ Technology freedom (Polyglot)

■ **Example**

Banking System:

- Account Service
- Payment Service
- Fraud Service
- Notification Service

Each runs independently.

■ **Short summary line**

Microservices are small, independent services designed around business capabilities.

✅ 2) Monolithic vs Microservices

■ **Explanation (simple, conceptual)**

🔵 Monolithic Architecture

- Entire application = one whole project

- Single codebase, single deployment
- One database for everything

🔴 Microservices Architecture

- Application split into independent services
- Each service has own deployment
- Each service may have its own database

■ When/Why used

Feature	Monolithic	Microservices
Deployment	One unit	Independent
Scalability	Entire app	Per service
Failure	Impacts whole app	Isolated
Complexity	Simple initially	High operational complexity
Best for	Small projects	Large enterprise applications

■ Short summary line

Monolithic = single unit; Microservices = many independent services.

✅ 3) Service Registry — Eureka

■ Explanation (simple, conceptual)

Eureka Server is a **Service Registry** where microservices **register themselves** and **discover other services**.

Two main components:

- **Eureka Server** → registry
- **Eureka Client** → microservice that registers itself

■ When/Why used

✔ Dynamic service discovery

✔ Avoids hardcoding URLs

✔ Load balancing (Ribbon + Eureka)

✔ Supports auto-scaling containers

⬛ **Example**

Eureka Server

```
@EnableEurekaServer
@SpringBootApplication
public class DiscoveryServer { }
```

Eureka Client

```
@EnableEurekaClient
@SpringBootApplication
public class PaymentService { }
```

⬛ **Short summary line**

Eureka provides service discovery so microservices find each other dynamically.

✅ 4) API Gateway — Why use it?

⬛ **Explanation (simple, conceptual)**

API Gateway is a **single entry point** for all microservices.

It handles:

- Routing
- Authentication
- Rate limiting
- Logging
- Caching
- Load balancing

Examples:

- Spring Cloud Gateway

- Netflix Zuul

■ **When/Why used**

✓ To avoid calling microservices directly

✓ To hide internal microservice URLs

✓ To apply common security

✓ To aggregate multiple microservice responses

■ **Example**

```
spring:
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: lb://USER-SERVICE
          predicates:
            - Path=/users/**
```

■ **Short summary line**

API Gateway routes, secures, and manages all microservice traffic through a single entry point.

✅ 5) Circuit Breaker (Resilience4j / Hystrix)

■ **Explanation (simple, conceptual)**

Circuit Breaker prevents application failure by **stopping calls** to a failing service.

States:

- **Closed** → working normally
- **Open** → stops requests to failing service
- **Half-open** → tests if service is back

■ **When/Why used**

✔ Avoid cascading failures

✔ Protect microservices from slow/unavailable services

✔ Improve resilience

✔ Provide fallback logic

■ **Example (Resilience4j)**

```
@CircuitBreaker(name = "paymentCB", fallbackMethod = "fallbackPaymen
t")
public String callPayment() {
    return restTemplate.getForObject("payment/api", String.class);
}

public String fallbackPayment(Exception ex) {
    return "Payment service is down";
}
```

■ **Short summary line**

Circuit Breaker protects microservices by stopping calls to failing services and providing fallback responses.

✅ 6) Feign Client usage

■ **Explanation (simple, conceptual)**

Feign Client is a **declarative REST client**.

It allows calling other microservices **just like calling a normal Java interface**.

■ **When/Why used**

✔ Simplifies HTTP calls

✔ No need for RestTemplate

✔ Integrates well with Eureka for service discovery

✔ Clean code — minimal boilerplate

■ **Example**

```
@FeignClient(name = "PAYMENT-SERVICE")
public interface PaymentClient {
    @GetMapping("/payment/status")
    String getPaymentStatus();
}
```

Usage:

```
paymentClient.getPaymentStatus();
```

■ Short summary line

Feign Client provides simple, declarative REST calls between microservices.

✅ 7) Inter-service communication

■ Explanation (simple, conceptual)

Microservices talk to each other using:

🔵 Synchronous

- REST API (RestTemplate, WebClient, Feign Client)

🔴 Asynchronous

- Message Queues (Kafka, RabbitMQ, ActiveMQ)
- Event-driven communication

■ When/Why used

Type	When used	Reason
Synchronous	Real-time data	immediate response
Asynchronous	Background jobs, notification, heavy workloads	decoupling, reliability

---

■ Examples

REST call

```
String response = paymentClient.getPaymentStatus();
```

Kafka message

```
kafkaTemplate.send("orders", order);
```

⬛ **Short summary line**

Microservices communicate synchronously using REST or asynchronously using message queues like Kafka.

📌 14. Real-Time Scenario Questions

✅ 1) Explain your project architecture end-to-end

⬛ **Explanation (simple, conceptual)**

A typical **Angular + Spring Boot + Microservices + SQL** architecture has these layers:

🔵 Frontend Layer (Angular)

- UI components, forms, dashboards
- Sends HTTP calls using HttpClient
- Handles routing, validation, JWT storage

🔵 API Gateway

- Single entry point
- Routes requests to microservices
- Security, rate-limiting, logging

🔵 Microservices Layer (Spring Boot)

- Independent services like User, Payment, Fraud, Notification

- Each has its own:
  ✔️ Controller
  ✔️ Service
  ✔️ Repository
  ✔️ DTO
  ✔️ Entity

🔵 Database Layer

- SQL Server (or MySQL/PostgreSQL)
- Tables mapped using JPA Entities
- Microservices may have separate DBs

🔵 Support Systems

- Eureka (Service Discovery)
- Kafka/RabbitMQ (Async events)
- Redis (Caching)
- ELK/Splunk (Logging)

■ When/Why used

✔️ Scalability
✔️ Independent deployments
✔️ Security via Gateway
✔️ Clean separation of frontend and backend

■ Short summary line

Architecture: Angular → Gateway → Microservices → Database → Monitoring layers.

✅ 2) How does Angular call Spring Boot API?

■ Explanation (simple, conceptual)

Angular uses **HttpClient** to make REST API calls to Spring Boot.

Flow:

1. Angular component → service
2. Service sends HTTP request
3. Backend returns JSON
4. Angular displays data

■ **When/Why used**

✔ To fetch data from backend

✔ To send form data (POST)

✔ To upload/download files

✔ To secure API with JWT

■ **Example (Angular → Spring Boot)**

Angular Service

```
getUsers() {
  return this.http.get<User[]>('http://localhost:8080/api/users');
}
```

Spring Boot Controller

```
@GetMapping("/users")
public List<User> getUsers() {
    return userService.getAll();
}
```

■ **Short summary line**

Angular sends HTTP requests using HttpClient; Spring Boot returns JSON responses.

✅ 3) Handling concurrency update conflicts

■ **Explanation (simple, conceptual)**

Concurrency conflict occurs when **two users update the same record at the same time**.

Example:

Two admins updating same user profile.

■ **When/Why needed**

✔ Prevent data overwrites

✔ Ensure data consistency

✔ Handle multi-user environments

■ **Common Solutions**

✔ 1. **Optimistic Locking (Most common)**

Use `@Version` field in Entity.

```
@Version
private int version;
```

If version mismatch → throws `OptimisticLockException`.

✔ 2. **Pessimistic Locking**

Database-level lock using:

```
@Lock(LockModeType.PESSIMISTIC_WRITE)
```

✔ 3. **Last-write-wins**

Application allows latest update only (not recommended).

■ **Short summary line**

Concurrency handled using optimistic/pessimistic locking to prevent lost updates.

✅ 4) Cache implementation in API

■ Explanation (simple, conceptual)

Caching stores frequently accessed results to avoid hitting the database repeatedly.

Spring Boot supports:

- In-memory cache
- Redis cache
- EhCache

■ When/Why used

✔ Improve performance

✔ Reduce DB load

✔ Speed up frequently accessed APIs (products, config, user details)

■ Example (Spring Cache + Redis)

Enable caching

```
@EnableCaching
@SpringBootApplication
public class App {}
```

Cache result

```
@Cacheable("users")
public User getUser(int id) {
    return repo.findById(id).get();
}
```

Evict cache

```
@CacheEvict(value = "users", key = "#id")
```

```
public void deleteUser(int id) { ... }
```

■ **Short summary line**

Caching improves performance by avoiding repeated DB calls using @Cacheable & Redis/EhCache.

✅ 5) How do you handle large result sets?

■ **Explanation (simple, conceptual)**

Large datasets can slow down API responses.

We use **Pagination and Streaming** to optimize.

■ **When/Why used**

✔ Large tables (orders, transactions, logs)

✔ Prevent huge payloads

✔ Reduce DB load

■ **Solutions**

✔ 1. Pagination (Most common)

```
Page<User> page = repo.findAll(PageRequest.of(0, 20));
```

✔ 2. Streaming (for very large data)

```
@Query("select u from User u")
Stream<User> streamAll();
```

✔ 3. Server-side filtering & sorting

✔ 4. Limit selected fields (DTO projection)

```
@Query("select new UserDTO(u.id, u.name) from User u")
```

■ **Short summary line**

Use pagination, streaming, and projections to efficiently handle large datasets.

✅ 6) Logging in microservices

■ **Explanation (simple, conceptual)**

Logging is critical in distributed systems for monitoring, debugging, and tracing.

Common tools:

- Logback / SLF4J (application-level logging)
- ELK Stack (Elasticsearch, Logstash, Kibana)
- Splunk / Grafana
- Zipkin / Sleuth for distributed tracing

■ **When/Why used**

✔ Trace requests across services

✔ Identify failures

✔ Performance monitoring

✔ Production debugging

■ **Example**

Application Logging

```
private static final Logger log = LoggerFactory.getLogger(UserService.class);

log.info("User created successfully");
log.error("User not found: {}", id);
```

Spring Cloud Sleuth adds:

```
traceId, spanId
```

■ **Short summary line**

Microservices logging uses SLF4J + centralized tools (ELK/Splunk) with trace IDs for distributed tracing.

✅ 7) How to optimize slow DB/REST API?

■ **Explanation (simple, conceptual)**

API slowness may come from:

- Slow queries
- Unoptimized DB schema
- Heavy JSON processing
- Network delays
- Large payloads

■ **Optimization Techniques**

✔ DB Level

- Add Indexes
- Optimize JOINs
- Use projection (fetch only required columns)
- Avoid N+1 problem with FetchType.LAZY
- Use Pageable

✔ API Level

- Caching using Redis
- Reduce response size (DTO)
- Asynchronous calls (`@Async`)
- Connection pooling (HikariCP)
- Use batching for DB writes

■ **Example**

Using projection:

```
@Query("select new UserDTO(u.name, u.email) from User u")
```

Using pagination:

```
repo.findAll(PageRequest.of(0, 10));
```

■ **Short summary line**

Optimize DB with indexes/queries and API with caching, pagination, and DTO projections.

✅ 8) Deployment steps for Spring Boot application

■ **Explanation (simple, conceptual)**

Spring Boot apps can be deployed on:

- AWS EC2
- Azure App Service
- Docker containers
- Kubernetes cluster
- On-prem servers

■ **General Deployment Steps**

✔ 1. Build JAR

```
mvn clean install
```

✔ 2. Configure environment variables

- DB URL
- Username

- Password
- JWT secret

✔️ 3. Package JAR

```
target/app.jar
```

✔️ 4. Run on server

```
java -jar app.jar
```

OR Docker:

✔️ 5. Docker Deployment

Dockerfile:

```
FROM openjdk:17
COPY app.jar app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

✔️ 6. Cloud Deployment (example Azure)

- Push code to repo
- Run CI pipeline
- Deploy artifact to Azure App Service
- Configure environment settings
- Test endpoints

■ **Short summary line**

Deployment: build JAR → configure env → run JAR or deploy via Docker/Kubernetes/cloud service.

This covers **100% Java + Spring Boot interview scope**.
