**Content:**

1. **Java**
2. **Oops**
3. **Exception Handling**
4. **Collection**
5. **Java 8**
6. **Multithreading**
7. **Spring Boot :(Spring Boot, Spring MVC, Spring JDBC, Spring ORM, Spring Security, Hibernate, Unit Testing)**
8. **Micro Service:**
9. **SQL**
10. **Git/Github**
11. **Docker - Left**
12. **Kubernates - Left**
13. **CI/CD - Left**
14. **AWS - Left**
15. **Design Pattern ( Saga pattern , Solid Principal) – Notes notebook**

**JAVA :**

**JVM** executes Java bytecode and provides a runtime environment,

**JRE** includes JVM and libraries to run Java programs

**JDK** includes JRE plus tools to develop, compile, and run Java applications.

**DataType:**

**Two types of data types** —

**1**.**Primitive** (store actual values like int, float, boolean)

**2.Non-Primitive/Reference** (store object references like String, Array, Class objects).

**Operator:**

**Arithmetic: + - \* / %**

**Relational: Compare values, return true/f**

**Logical: && || ! to combine conditions**

**Bitwise: Operate on bits (& | ^ ~ << >>)**

**Assignment: =, +=, -=, ...**

**Unary: Single operand (++, --, -)**

**Ternary: condition ? trueVal : falseVal**

**instanceof: Checks object type**

**Control Flow Statements:**

**Conditional Statements:**

* if-else: Execute code if condition true, else another block
* if-else if-else: Multiple conditions
* switch: Test variable against values (break prevents fall-through)

**Loops:**

* for: Known number of iterations
* for-each: Iterate arrays/collections (read-only)
* while: Repeat while condition true (pre-check)

**Branching Statements:**

* break: Exit loop/switch
* continue: Skip current iteration
* return: Exit method (optionally with value)

**Strings:**

* **String**: Immutable, stored in **String Pool**, thread-safe, good for keys in HashMap. "hello" reuses pool; new String("hello") creates a new object.
* **StringBuilder**: Mutable, **not thread-safe**, efficient for many modifications (append, insert, delete, reverse).
* **StringBuffer**: Mutable, **thread-safe** (synchronized), slower than StringBuilder; use in multi-threaded scenarios.
* **Tip**: Single-thread → StringBuilder; Multi-thread → StringBuffer.

**Arrays:**

* Fixed-size container for same type values.
* **Declaration:** dataType[] arr;
* **Initialization:** int[] nums = new int[5]; or String[] names = {"A","B"};
* **Access:** arr[index], **length:** arr.length
* **Multidimensional:** int[][] matrix = new int[3][3];
* **Limit:** Size fixed, cannot grow/shrink

**Wrapper Classes:**

* Object representation of primitives (int → Integer, double → Double, etc.)
* Needed for collections, utility methods, null support
* **Autoboxing:** primitive → object (Integer i = 10;)
* **Unboxing:** object → primitive (int n = i;)

**OOPS: Object-Oriented Programming (OOPS)**

**Encapsulation:**

* **Definition:** Binding data (fields) and methods into a single unit (class).
* **Purpose:** Protect data and control access — core of **data hiding**.
* **How:**
  + Make variables **private**, Provide **public getters/setters** to access or modify them.
* **Benefits:**
  + Increases security, Improves maintainability and flexibility, Prevents direct modification of data.

**Example:**

**class Person {**

**private String name;**

**public String getName() { return name; }**

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

**Inheritance :**

* **Definition:** Mechanism where one class (**child/subclass**) inherits properties & methods of another (**parent/superclass**).
* **Keyword:** extends (for classes), implements (for interfaces).
* **Types:**
  + **Single** – one parent, one child
  + **Multilevel** – chain of inheritance
  + **Hierarchical** – one parent, multiple children  
    *(Java doesn’t support multiple inheritance with classes)*
* **Benefits:** Code reuse, easier maintenance, and polymorphism.

**Example:**

class Animal { void eat(){} }

class Dog extends Animal { void bark(){} };

**Polymorphism :**

* **Definition:** Ability of an object to take **many forms** (same method behaves differently).
* **Types:**
  + **Compile-time (Static):** Method overloading (same name, different params).
  + **Runtime (Dynamic):** Method overriding (subclass provides its own implementation).
* **Keyword:** @Override (for overriding).
* **Benefit:** Increases flexibility and reusability of code.

**Example:**

class Animal { void sound(){

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

} };

class Dog extends Animal { void sound(){

System.out.println("Bark");

} };

Here, the same *sound()* behaves differently at runtime.

**Abstraction :**

**Definition:** Hiding implementation details and showing only essential features.

* **Purpose:** Focus on what an object does, not how it does it.
* **How:**
  + Using **abstract classes** (abstract keyword).
  + Using **interfaces** (fully abstract by default).
* **Benefits:** Simplifies code, increases flexibility, and improves security.

**Example:**

abstract class Shape {

abstract void draw();

}

class Circle extends Shape {

void draw() {

System.out.println("Drawing Circle"); }};

**Abstract Class vs Interface :**

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Purpose** | Shared base with partial implementation | Full abstraction and contract definition |
| **Methods** | Can have abstract **and** non-abstract methods | Only abstract methods (till Java 7); Java 8+ allows **default** & **static** methods |
| **Variables** | Can have instance & static variables | All variables are **public, static, final** |
| **Inheritance** | Supports **single inheritance** | Supports **multiple inheritance** |
| **Keyword** | abstract | interface, implemented using implements |
| **Constructor** | Can have constructors | Cannot have constructors |
| **Use Case** | When classes share common behavior or state | When defining a common API or capability |

**Equals() and HashCode()**

Used to compare objects in Java and maintain correct behavior in hash-based collections (e.g., HashMap, HashSet).

**Key Points:**

* equals() → Compares **content** of objects.
* hashCode() → Returns an integer hash (used in hashing algorithms).
* If you **override equals()**, you **must override hashCode()** to maintain the rule:
* If two objects are equal, they must have the same hash code.

### ****Composition vs Inheritance****

**Inheritance:**

* “**is-a**” relationship , reuse behavior from a parent class using extends ,
* Tight coupling — child depends on parent’s structure.

**Example:**

**class Vehicle { void start(){ System.out.println("Starts"); } }**

**class Car extends Vehicle { void drive(){ System.out.println("Drives"); } }**

Car **is a** Vehicle.

**Composition:**

* “**has-a**” relationship ,One class contains another as a field instead of extending it.
* Promotes loose coupling and flexibility.

**Example:**

class Engine { void run(){ System.out.println("Engine running"); } }

class Car {

private Engine engine = new Engine();

void start() { engine.run(); }

}

O/p: Car **has an** Engine.  
**NOTE:**

* Use **inheritance** for shared behavior.
* Use **composition** for flexibility and reusability (preferred in modern design).

**Exception Handling :**

* **Purpose:** Handle runtime errors gracefully, prevent program crash.
* **try:** Code that may throw exception
* **catch:** Handle specific exceptions (multiple allowed, most specific first)
* **finally:** Always executes, for cleanup (e.g., closing resources)
* **throw:** Explicitly throw an exception
* **throws:** Declare checked exceptions in method signature

**Types:**

* **Checked:** Must handle, compile-time (IOException, SQLException)
* **Unchecked (Runtime):** Optional to handle, runtime (NullPointerException, ArithmeticException)
* **Error:** Severe, cannot recover (OutOfMemoryError)
* **Custom Exceptions:** Extend Exception (checked) or RuntimeException(unchecked)

**Access Modifiers:**

* **public:** Anywhere
* **private:** Same class only
* **protected:** Same package + subclasses
* **default:** Same package only

**this & super:**

* **this:** Current object; differentiate variables, call another constructor, pass/return current object
* **super:** Parent object; call parent constructor/method, access hidden variables

**static:**

* Belongs to class, not instance
* **static variable:** Shared across all objects (one copy).

**static Keyword:**

* **Static variable:** Initialized once, shared, accessed via ClassName.var
* **Static method:** Class-level, called via ClassName.method(), can access only static members, cannot use this/super, cannot be overridden (method hiding)
* **Static block:** Runs once on class loading, used for static initialization, multiple blocks execute in order
* **Static nested class:** Can exist without outer instance, accesses only static members of outer class

**final Keyword:**

* **final variable:** Assigned once, constant (primitive value or reference cannot change)
* **final method:** Cannot be overridden by subclasses.
* **final class:** Cannot be subclassed; ensures immutability/security (e.g., String).

**Garbage Collection:**

* **Purpose:** Automatic memory management; frees memory of unreachable objects
* **JVM Component:** Daemon thread running in background
* **Advantages:** Prevents memory leaks, no manual memory management
* **Invocation:** System.gc() or Runtime.getRuntime().gc() (only a hint)
* **Memory Areas:**
  + **Heap:** Objects (Young/Old generations)
  + **Stack:** Method calls, local variables
  + **Method Area:** Class info, static variables
  + **PC Registers & Native Stacks:** Thread execution & native methods
* **Algorithms:** Mark & Sweep, Copying (simplified object cleanup).

**Serialization & Deserialization :**

* **Serialization:** Convert object → byte stream for storage/transfer (Serializable interface, ObjectOutputStream)
* **Deserialization:** Convert byte stream → object (ObjectInputStream).
* **transient:** Fields not serialized; reset to default on deserialization.

**Generics :**

* **Purpose:** Type safety, no casting, reduces runtime errors
* **Syntax:** List<String> list = new ArrayList<>();
* **Type Erasure:** Generic info removed at compile-time; ensures backward compatibility
  + Cannot create new T(), use instanceof T, or new T[size]
* **Wildcards:**
  + <?> → any type
  + <? extends T> → T or subclass (read-only, PECS: Producer extends)
  + <? super T> → T or superclass (write allowed, read as Object, PECS: Consumer super).

**Java Collection Framework :**

**Collection (interface)**

**│**

**├── List**

**│ ├── ArrayList**

**│ ├── LinkedList**

**│ └── Vector (Legacy, synchronized) ----> stack**

**│**

**├── Set**

**│ ├── HashSet**

**│ ├── LinkedHashSet**

**│ └── TreeSet (sorted)**

**│**

**└── Queue**

**├── PriorityQueue**

**└── ArrayDeque**

**Map is not part of Collection but part of the framework:**

**Map**

**├── HashMap**

**├── LinkedHashMap**

**└── TreeMap (sorted)**

**What is a Collection?**

* A **container** that holds a group of objects as a single unit.
* Common ops:
  1. Add
  2. Remove
  3. Retrieve
  4. Iterate

|  |  |
| --- | --- |
| **collection** | A group of stored objects (general term). |
| **Collection** | Root **interface** in java.util (extends Iterable). |
| **Collections** | **Utility class** with static methods (sort, reverse, etc). |

### ****Collection Interface****

* **Root of the collection hierarchy**, extends Iterable.
* Key methods:  
  add(), remove(), addAll(), removeAll(), retainAll(), contains(), isEmpty(), size(), iterator(), toArray().

### ****Sub-interfaces of Collection****

1. **List**
2. **Set**
3. **Queue**
4. **Deque**

### ****List Interface****

* **Ordered**, allows **duplicates**.
* Access via **index**.
* Implementations: ArrayList, LinkedList, Vector.
* **List-specific methods:**  
  add(int, E), remove(int), get(int), set(int, E), indexOf(Object), listIterator(), subList().

### ****ArrayList****

* **Resizable array**, allows null.
* Implements List, RandomAccess.
* ✅ Fast random access
* ❌ Slow insertion/deletion in middle

### ****LinkedList:****

* Stores elements in **nodes (doubly linked)**.
* Fast insertion/deletion in middle.
* Implements List, Deque.
* Key methods:  
  addFirst(), addLast(), removeFirst(), removeLast(), getFirst(), getLast().

### ****Vector****

* Legacy class (since JDK 1.0).
* **Synchronized**, dynamic array.
* Similar to ArrayList.
* Slower due to synchronization.

 **Iterator** – Used to traverse Collection (forward only)

 **ListIterator** – Used with List, supports forward and backward traversal.

### ****CopyOnWriteArrayList****

CopyOnWriteArrayList is a **thread-safe variant of ArrayList** in Java that allows **safe iteration** even when multiple threads modify the list concurrently.

On every write operation, it **creates a new copy** of the underlying array.

When **reads are frequent** and **writes are rare**

**In short:**  
Use **ArrayList** for speed in single-threaded apps. Use **CopyOnWriteArrayList** for safe iteration in multi-threaded read-heavy apps.

**Fail-Fast Iterators** operate on the **original collection** and throw ConcurrentModificationException if the collection is modified during iteration.

**Fail-Safe Iterators** operate on a **snapshot (copy)** of the collection, so modifications do not affect iteration, making them **thread-safe** but slightly **slower**.

**Set Interface:**

* **No duplicates**, **unordered**.
* Max one null.
* Implementations:  
  HashSet, LinkedHashSet, TreeSet.

### ****HashSet :****

 I**mplements:** Set interface ,Uses a **HashMap** internally (values stored as keys in HashMap)

 **Order:** **Unordered, Unsorted** (insertion order not maintained) , **Duplicates:** **Not allowed** (checks using hashCode() + equals())

 Allows **one null element**

 **Important:** For **custom objects**, must **override equals() and hashCode()**.

 **Use Case:** When you need **unique elements** and **don’t care about order**.

### ****LinkedHashSet:****

### Uses LinkedHashMap internally , Maintains insertion order , Not allowed

### Performance: Slightly slower than HashSet (due to linked list maintenance)

### Null: Allows one null element

### Use Case: When you need unique elements but want to preserve insertion order.

### ****TreeSet:****

**implements: NavigableSet, SortedSet , Uses TreeMap (Red-Black Tree) internally ,duplicates: Not allowed**

**Sorted order (ascending by default)**

**Performance: O(log n) for add, remove, contains**

**Nulls: Not allowed (throws NullPointerException)Sorting Basis:**

**Uses natural ordering (via Comparable) Or custom ordering (via Comparator)**

### ****Map Interface****

### Stores **key–value pairs**.

* No duplicate keys; values can repeat.

## ****HashMap – Key Points****

* Stores **key-value pairs**
* Allows **one null key** and **multiple null values**
* **Not thread-safe**
* Used most commonly due to **high performance**

**Interview Questions:**

* How does HashMap handle collisions?  
  → Using **LinkedList** or **Red-Black Tree** (from Java 8).
* What is load factor and capacity?  
  → Default capacity = 16, load factor = 0.75.

Is HashMap thread-safe?  
→ No, use **ConcurrentHashMap** for thread safety

## ****LinkedHashMap – Key Points****

* Extends **HashMap**
* Maintains **insertion order**
* Uses a **doubly linked list** to maintain order
* Slightly slower than HashMap

How LinkedHashMap maintains order?  
→ Internally maintains **doubly-linked list**.

Can we maintain access order?  
→ Yes, by using constructor:

## ****TreeMap – Key Points****

* Stores data in **sorted order of keys**
* Uses **Red-Black Tree** internally
* **Does not allow null key**, but allows **null values**
* Slower than HashMap (O(log n))

**Interview Questions:**

* How does TreeMap maintain order?  
  → Using **Red-Black Tree** (self-balancing binary search tree).
* Can we use custom sorting?  
  → Yes, using Comparator:

## ****Hashtable – Key Points****

* **Synchronized** version of HashMap (thread-safe)
* **Does not allow null key or value**
* **Legacy class** (from Java 1.0)
* Slower due to synchronization

Why Hashtable is legacy?  
→ Introduced before Collections Framework.

Why Hashtable does not allow nulls?  
→ Null used to indicate absence of a key internally.

Should we use Hashtable now?  
→ No, use **ConcurrentHashMap** instead.

### ****ConcurrentHashMap****

* Thread-safe, high-performance **map for multithreading**.
* Uses **lock striping** (not full lock).
* No null keys/values.

Doesn’t throw ConcurrentModificationException

**Notes:**

|  |
| --- |
| **Comparable** : Natutral Ordering , Collections.sort(list)  Example : Sort by ID |
| **Comparator** :Custom Ordering , Collections.sort(list, comparator) |

|  |  |
| --- | --- |
| **Example** | Sort by Name or Salary |

# ****Quick Summary Table****

| **Topic** | **Key Difference Summary** | **Interview Use** |
| --- | --- | --- |
| **Array vs ArrayList** | Array = fixed size, ArrayList = dynamic | When data size changes, use ArrayList |
| **ArrayList vs LinkedList** | ArrayList = fast access, LinkedList = fast insert/delete | Use based on operation type |
| **HashMap vs Hashtable** | HashMap = faster, not thread-safe | Hashtable = synchronized, legacy |
| **Comparable vs Comparator** | Comparable = natural order, Comparator = custom order | Use Comparator for flexible sorting |

## Java Serialization:

### ****What is Serialization?****

* Converts object → **byte stream** for storage or transmission.
* Uses: Persistence, Communication, Object Cloning, Distribution.

### ****Serializable Interface****

* Marker interface (java.io.Serializable), no methods.

### ****Steps****

1. Implement Serializable.
2. Use ObjectOutputStream to write.
3. Use ObjectInputStream to read.

### ****SerialVersionUID****

* Unique 64-bit ID for version control.
* If mismatch → InvalidClassException.
* Best practice: define explicitly.

private static final long serialVersionUID = 1L;

### ****More Serialization Facts****

* ❌ Static fields not serialized.
* ✅ References are serialized **only if** they implement Serializable.
* Use transient to **skip serialization** of specific fields.

**JAVA 8:**

* **Lambda Expressions:** (params) -> expression for concise anonymous functions; used with functional interfaces.
* **Functional Interfaces:** One abstract method; e.g., Runnable, Comparator, Consumer.
* **Stream API:** Declarative operations on collections.
* Intermediate (transform) and Terminal (produce result).
* filter(Predicate) – Filters elements by condition.
* map(Function) – transform elements
* flatMap(Function) – flatten nested streams
* distinct() – remove duplicates
* sorted() / sorted(Comparator) – sort elements
* peek(Consumer) – perform action on each element without consuming

**Terminal Operations (produce result)**

* **Operation Description**
* **forEach(Consumer) -> Iterate and perform action.**
* **collect(Collector) -> Collect into list, set, map.**
* **reduce(BinaryOperator)->combine into single result.**
* **count() ->Count elements.**
* **min(Comparator), max(Comparator)->Find min or max.**
* **anyMatch, allMatch, noneMatch ->Match checks.**
* **findFirst(), findAny()->Get first or any element.**
* **Default Methods:** Methods in interfaces with default keyword; allow backward-compatible additions.
* **Optional:** A container object to avoid NullPointerException. Example : isPresent(), orElse(), ifPresent(), orElseThrow();
* **Date & Time API:** java.time package; immutable, thread-safe replacements for Date/Calendar.
  + Key classes: LocalDate, LocalTime, LocalDateTime, Instant, ZonedDateTime, Duration.

**MULTITHREADING:**

### ****What is a Process?****

* A process is a **program in execution**.
* Each process has its **own memory space** and **address space**.
* A process is **heavyweight** and consists of one or more threads.

### ****What is a Thread in Java?****

* A **thread** is a separate **path of execution** within a program.
* Threads are **lightweight**, share the **same memory**, and are **faster** to create.
* Every process has at least **one thread** (main thread).

**Multithreading:** Running multiple threads (lightweight units of execution) within a single process.

**Concurrency:** Multiple tasks in progress at the same time (can be parallel or interleaved).

### ****Ways to Create a Thread****

#### ****1. Extending**** Thread ****class****

class MyThread extends Thread {

public void run() {

System.out.println("Thread running: " + Thread.currentThread().getName());

}

}

public class Demo {

public static void main(String[] args) {

new MyThread().start(); // starts new thread

}

}

✅ Simple, but you **can’t extend another class**.

#### ****2. Implementing**** Runnable ****interface**** (Recommended)

class MyRunnable implements Runnable {

public void run() {

System.out.println("Runnable running: " + Thread.currentThread().getName());

}

}

public class Demo {

public static void main(String[] args) {

Thread t = new Thread(new MyRunnable());

t.start();

}

}

✅ Flexible (since it allows extending other classes).

**Thread LifeCycle:**

NEW → RUNNABLE → RUNNING → (WAITING/BLOCKED/TIMED\_WAITING) → RUNNABLE → TERMINATED

| **State** | **Description** | **Example / Transition** |
| --- | --- | --- |
| **1. NEW** | Thread is created but not yet started. | Thread t = new Thread(); |
| **2. RUNNABLE** | Thread is ready to run and waiting for CPU time. | After calling t.start() |
| **3. RUNNING** | Thread is actively executing code in run() method. | CPU schedules the thread |
| **4. BLOCKED** | Waiting to acquire a lock (synchronized resource). | Another thread holds the lock |
| **5. WAITING / TIMED\_WAITING** | Waiting indefinitely or for a specified time for another thread’s action. | wait(), join(), sleep(ms) |
| **6. TERMINATED (DEAD)** | Thread has completed execution or stopped due to error. | run() method ends |
|  |  |  |

## ****Synchronization & Thread Safety (Java):****

### ****1. Synchronization****

* Controls access to shared resources to **prevent race conditions**.
* Achieved using **synchronized keyword**, **volatile**, or **Atomic classes**.

### ****2. synchronized****

* **Synchronized Method:**
  + Locks **object** for instance methods (this)
  + Locks **class** for static methods (ClassName.class)
  + Only **one thread** can execute at a time per lock.
* **Synchronized Block:**
  + Locks a **specific object** for finer control.
  + Syntax:
  + synchronized(lock) {
  + // critical section
  + }
* **Re-entrant:** Thread can re-enter locks it already holds.
* **Intrinsic Lock:** Every object has a monitor lock.

### ****3. volatile****

* Ensures **visibility** of variable changes across threads.
* **Does NOT guarantee atomicity** (i++ is not atomic).
* Use when **one thread writes**, others only read.

private volatile boolean running = true;

### ****4. wait(), notify(), notifyAll()****

* From Object class, **must be in synchronized block/method**.
* **wait():** Thread releases lock and waits.
* **notify():** Wakes one waiting thread (non-deterministic).
* **notifyAll():** Wakes all waiting threads.
* **Use Case:** Producer-Consumer problem.

### ****5. Race Condition****

* Occurs when **multiple threads modify shared data** simultaneously, leading to unpredictable results.

**NOTE:**

* Use **synchronized** for critical sections.
* Use **volatile** for shared flags.
* Use **wait/notify** for inter-thread communication.

## ****Thread Pools (ExecutorService)****

* **Problem:** Creating threads per task is costly (memory & CPU).
* **Solution:** Thread pools reuse threads; tasks are submitted to a pool.
* **ExecutorService:** Interface for async task execution.
* **Executors Factory Methods:**
  + newFixedThreadPool(n) → fixed number of threads
  + newCachedThreadPool() → dynamic, reuses idle threads
  + newSingleThreadExecutor() → single-threaded sequential execution
  + newScheduledThreadPool(n) → delayed/periodic tasks
* **Shutdown:** shutdown() (graceful), shutdownNow() (forceful)

**Example:**

ExecutorService pool = Executors.newFixedThreadPool(3);

pool.submit(() -> System.out.println("Task executed"));

pool.shutdown();

**2. Concurrency Utilities (**java.util.concurrent**).**

* **Lock / ReentrantLock:** Flexible locking alternative to synchronized.
* lock(), unlock(), tryLock(), lockInterruptibly().
* **Condition:** Replaces wait/notify with multiple wait-sets.
* **ReentrantReadWriteLock:** Multiple readers / single writer (read-heavy optimization).
* **CountDownLatch:** Waits for a set of threads to complete.
* **CyclicBarrier:** Threads wait for each other at a barrier; reusable.
* **Semaphore:** Limits access to resources (permits).
* **BlockingQueue:** Thread-safe producer-consumer queues (put(), take()).
* **Atomic Variables:** (AtomicInteger, AtomicBoolean) → atomic updates without locks.

## ****3. Deadlock****

* **Definition:** Two or more threads block indefinitely, each waiting for resources held by others.
* **Coffman Conditions (all 4 must exist):**
  1. Mutual Exclusion
  2. Hold & Wait
  3. No Preemption
  4. Circular Wait
* **Prevention Strategies:**
  1. Break circular wait (consistent lock order)
  2. Avoid hold & wait (acquire all resources at once)
  3. Allow preemption (release if unavailable)
  4. Avoid mutual exclusion using concurrent data structures
* **Detection:** Thread dumps (jstack) or monitoring tools
* **Resolution:** Fix code to break at least one Coffman condition

✅ **Tip:**

* Use **ExecutorService + concurrent utilities** for robust, scalable multithreaded applications.
* Prefer **Atomic/BlockingQueue** for simple shared data instead of manual synchronization.
* Always **prevent or detect deadlocks** in production-grade systems

**Q. How can we make multiple threads (t1, t2, t3) run one by one in Java instead of concurrently?**

We can control the execution order of threads in Java using the **join() method** or **ExecutorService**.

**SPRINGBOOT:**

## ****1. Spring Core Concepts****

### ****IoC & Dependency Injection (DI)****

* **IoC:** Spring container manages object creation & lifecycle.
* **DI:** Container injects dependencies instead of objects creating them.  
  **Types:**

1. **Constructor Injection** (Recommended, immutable)
2. **Setter Injection** (Flexible, optional deps)
3. **Field Injection** (Concise, less testable)

@Autowired

public UserService(UserRepository repo){

this.repo = repo;

};

### ****Bean Stereotypes & Lifecycle****

| **Annotation** | **Layer / Use** |
| --- | --- |
| @Component | Generic Spring bean |
| @Service | Business logic |
| @Repository | Data access, exception translation |
| @Controller | Web layer (MVC) |
| @RestController | REST API, returns JSON/XML |
| @Bean | Method-level bean |
| @Configuration | Defines @Bean methods |
| @Autowired / @Qualifier | Dependency injection |
| @Value("${prop}") | Inject config values |

### ****Spring Boot Web Annotations****

* @PathVariable → URI variables (/users/{id})
* @RequestParam → Query params (?name=Alice)
* @RequestBody → Map request body to object
* @ResponseBody → Return object as response
* ResponseEntity<T> → Full control of HTTP response
* **Exception Handling:** @ResponseStatus, @ExceptionHandler, @ControllerAdvice/@RestControllerAdvice

### ****Application Configuration****

* application.properties / application.yml → External config
* Profiles: application-dev.yml, application-prod.yml

## ****2. Spring Data JPA / Hibernate****

* **ORM:** Map Java objects → DB tables
* **JPA Annotations:** Spring Data JPA simplifies data access layer development by providing abstractions over JPA (Java Persistence API) and Hibernate (a popular JPA implementation).
  + @Entity, @Table, @Id, @GeneratedValue, @Column
  + Relationships: @OneToMany, @ManyToOne, @OneToOne, @ManyToMany
* **Repositories:**
  + CrudRepository → Basic CRUD
  + PagingAndSortingRepository → CRUD + paging & sorting
  + JpaRepository → JPA-specific + batch ops
* **Derived Queries:** Auto-generate from method names:

findByFirstName(String name)

findByAgeGreaterThan(int age)

**Lazy vs Eager Loading:**

* + Lazy: Load data on access (default @OneToMany, @ManyToMany)
  + Eager: Load immediately (default @ManyToOne, @OneToOne)
  + Recommendation: Prefer lazy + explicit fetch to avoid N+1 problem.

**Microservices :**

### ****Definition****

* An architecture where an application is built as a suite of **small, independent, deployable services**.
* Each service:
  + Runs in its own process
  + Communicates via REST / Message Queues
  + Focused on a **single business capability**
  + Independently deployable & scalable
  + Often has its own database (decentralized data)

### ****Advantages****

* **Scalability:** Scale services independently.
* **Resilience:** Failure in one service doesn’t break the app.
* **Tech Flexibility:** Different languages/databases per service.
* **Faster Development:** Smaller teams work independently
* **Easier Maintenance:** Smaller codebases
* **Deployment Flexibility:** Independent, faster releases

### ****Disadvantages****

* **Complexity:** Distributed systems, network latency, debugging
* **Operational Overhead:** More services = more monitoring/infrastructure
* **Data Consistency:** Hard to maintain across services
* **Inter-service Communication:** Needs robust mechanisms

### ****Common Patterns****

| **Pattern** | **Purpose / Example** |
| --- | --- |
| **API Gateway** | Single entry point; routing, auth, load balancing. Example: Spring Cloud Gateway, Zuul |
| **Service Discovery** | Locate services dynamically. Example: Eureka, Consul |
| **Circuit Breaker** | Prevent cascading failures. Example: Resilience4j |
| **Centralized Config** | Externalize configs. Example: Spring Cloud Config Server |
| **Load Balancing** | Distribute traffic. Client-side: Ribbon/Spring Cloud LoadBalancer; Server-side: Nginx/ELB |

### ****Communication Between Services****

1. **Synchronous (REST/HTTP)**
   * Pros: Simple, easy to debug
   * Cons: Blocking, tight coupling, cascading failures
2. **Asynchronous (Messaging Queues)**
   * Pros: Loose coupling, resilient, scalable, pub/sub supported
   * Cons: Complex, message ordering issues, needs broker (Kafka/RabbitMQ)

**DATABASE:**

## ****RDBMS Concepts & SQL****

**What is a transaction?**

**A:** A **transaction** is a unit of work that performs one or more operations on a database in a way that they **all succeed or all fail together**.

A **transaction** ensures data **consistency** and **integrity**,  
following **ACID** principles, managed in Spring using **@Transactional**.

* **RDBMS:** Stores structured data in tables (rows & columns) with relationships via PK & FK.
* **Examples:** MySQL, PostgreSQL, Oracle, SQL Server, H2.
* **ACID Properties:** Ensures reliable transactions
  + **Atomicity:** All or nothing
  + **Consistency:** Database remains valid after transactions
  + **Isolation:** Concurrent transactions don’t interfere
    - **Levels:**
      * Read Uncommitted → Dirty Read
      * Read Committed → Only committed data
      * Repeatable Read → No dirty/non-repeatable read, phantom possible
      * Serializable → Full isolation, slowest
  + **Durability:** Changes persist after commit
* **SQL Categories:**
  + **DML:** SELECT, INSERT, UPDATE, DELETE
  + **DDL:** CREATE, ALTER, DROP, TRUNCATE
  + **DCL:** GRANT, REVOKE
  + **TCL:** COMMIT, ROLLBACK, SAVEPOINT
* **Common Constructs:** JOIN, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT/OFFSET, Subqueries, UNION
* **Indexes:**
  + Speeds up SELECT, JOIN, ORDER BY
  + **Clustered:** Determines physical order (1 per table)
  + **Non-clustered:** Logical ordering, multiple per table
  + **Tradeoff:** Faster read, slower write, more storage

## ****2. JDBC (Java Database Connectivity)****

* **Purpose:** Connect Java apps to RDBMS.
* **Steps:** Load driver → Connection → Statement/PreparedStatement → Execute → Process ResultSet → Close resources
* **PreparedStatement:** Prevents SQL injection, better for repeated queries
* **Drawback:** Verbose, boilerplate-heavy → ORM preferred

## ****3. JPA (Java Persistence API)****

* **Purpose:** ORM specification for mapping Java objects to DB tables
* **Features:**
  + **EntityManager:** CRUD, queries, persistence context
  + **JPQL:** Object-oriented query language
  + Simplifies JDBC, focuses on business logic

## ****4. Hibernate****

* **Purpose:** Popular ORM framework, JPA implementation
* **Querying:** Native SQL, HQL, Criteria API
* **Caching:**
  + **First-Level (Session Cache):** Per EntityManager, default
  + **Second-Level:** Shared across sessions, optional (Ehcache, Redis)
* **Fetching Strategies:**
  + **Lazy (default @OneToMany, @ManyToMany):** Load when accessed → Avoid unnecessary data
  + **Eager (default @ManyToOne, @OneToOne):** Load immediately
  + **Optimizations:**
    - JOIN FETCH, @EntityGraph → Fetch related entities in one query
    - Batch Fetching → Reduce multiple queries
    - DTOs → Fetch only required fields

## ****5. Database Normalization & Denormalization****

* **Normalization:** Reduce redundancy, improve integrity
  + **1NF:** Atomic columns, no repeating groups
  + **2NF:** Full dependency on PK, remove partial dependencies
  + **3NF:** No transitive dependencies
  + **BCNF:** Stronger 3NF, fixes specific anomalies
  + **Pros:** Less redundancy, better integrity
  + **Cons:** More JOINs, possible slower reads
* **Denormalization:** Introduce redundancy to improve read performance
  + **Pros:** Faster queries, simpler reads
  + **Cons:** More storage, update anomalies
  + **Use Case:** Reporting, data warehousing, read-heavy apps

## ****GIT/GITHUB:****

### ****Why Version Control?****

* **Track Changes:** Who changed what, when, why.
* **Collaboration:** Multiple developers on same code without conflicts.
* **Reversion:** Revert to previous code versions.
* **Branching & Merging:** Parallel development of features/bug fixes.
* **Backup & Audit:** Full history distributed locally for redundancy.

### ****Git Basics****

* **DVCS (Distributed VCS):** Every developer has full repo locally.  
  **Benefits:** Offline work, faster operations, resilient, flexible branches.
* **Core Concepts:**
  + **Repo:** .git folder with history & metadata
  + **Commit:** Snapshot of files with SHA-1 hash, author, message
  + **Branch:** Pointer to a commit, used for isolated feature/dev work
  + **Head:** Points to tip of current branch
  + **Index/Staging Area:** Prepares files for next commit
  + **Working Directory:** Actual files being worked on

### ****Common Commands****

* **Setup & Info:**
  + git init → Initialize repo
  + git clone <url> → Copy remote repo locally
  + git status → Check file status
* **Staging & Commit:**
  + git add <file> / git add . → Stage changes
  + git commit -m "message" → Commit staged changes
* **History & Diff:**
  + git log / git log --oneline → Show history
  + git diff → Changes not staged
  + git diff --staged → Changes staged but not committed
* **Remote Operations:**
  + git push → Upload local commits
  + git pull → Fetch + merge remote changes
  + git fetch → Download changes without merge

### ****Branching & Merging****

* **Branch Management:**
  + git branch → List branches
  + git branch <name> → Create branch
  + git checkout <name> → Switch branch
  + git checkout -b <name> → Create & switch branch
  + git branch -d/-D <name> → Delete branch
* **Merging:**
  + git merge <branch> → Combine changes
    - **Fast-Forward:** Pointer moves if no new commits
    - **Three-Way Merge:** Creates merge commit if branches diverged
* **Rebase:**
  + git rebase <base> → Reapply commits on top of base branch
  + **Use:** Clean linear history
  + **Caution:** Don’t rebase public/shared branches

### ****Common Branching Strategies****

* **Git Flow:** Master + develop + feature/release/hotfix branches (large projects)
* **GitHub Flow / GitLab Flow:** Main branch with short-lived feature branches (simpler, modern workflow)

### ****Quick Tips****

* Always write **descriptive commit messages**
* Use **branches for features/bugs**, merge to main/develop
* Rebase only **private feature branches**
* Regularly **pull/fetch** from remote to avoid conflicts.

**JAVA 8 Question for interview**

Java Stream API Interview Questions (Employee / Number Based)

Q1. How to print all even numbers from a list?

numbers.stream().filter(n -> n % 2 == 0).forEach(System.out::println);

Q2. How to print all odd numbers from a list?

numbers.stream().filter(n -> n % 2 != 0) .forEach(System.out::println);

Q3. How to find the sum of all numbers?

int sum = numbers.stream().mapToInt(Integer::intValue).sum();

Q4. How to find the maximum and minimum number?

int max = numbers.stream().max(Integer::compare).get();

int min = numbers.stream().min(Integer::compare).get();

Q5. How to count even numbers?

long count = numbers.stream().filter(n -> n % 2 == 0).count();

Q6. How to sort a list in ascending and descending order?

numbers.stream().sorted().forEach(System.out::println); // ascending

numbers.stream().sorted(Comparator.reverseOrder()).forEach(System.out::println); // descending

Intermediate Stream API Questions (Employee List)

Q7. How to print all employee names?

employeeList.stream().map(Employee::getEmpName).forEach(System.out::println);

Q8. How to filter employees older than 30?

employeeList.stream().filter(e -> e.getAge() > 30).forEach(System.out::println);

Q9. How to get a list of employees in the "HR" department?

List<Employee> hrEmployees = employeeList.stream()

.filter(e -> "HR".equals(e.getDepartment())).collect(Collectors.toList());

Q10. How to sort employees by empName?

List<Employee> sortedByName = employeeList.stream().sorted(Comparator.comparing(Employee::getEmpName))

.collect(Collectors.toList());

Q11. How to sort employees by salary in descending order?

List<Employee> sortedBySalary = employeeList.stream().sorted(Comparator.comparing(Employee::getSalary).reversed()).collect(Collectors.toList());

Q12. How to find the employee with the highest salary?

Optional<Employee> highestSalary = employeeList.stream().max(Comparator.comparing(Employee::getSalary));

Q13. How to find the total salary of all employees?

double totalSalary = employeeList.stream().mapToDouble(Employee::getSalary).sum();

Q14. How to get the average salary of employees?

double averageSalary = employeeList.stream().mapToDouble(Employee::getSalary).average().orElse(0.0);

Q15. How to group employees by department?

Map<String, List<Employee>> employeesByDept = employeeList.stream().collect(Collectors.groupingBy(Employee::getDepartment));

Q16. How to count employees in each department?

Map<String, Long> countByDept = employeeList.stream().collect(Collectors.groupingBy(Employee::getDepartment, Collectors.counting()));

Q17. How to check if any employee has salary > 50,000?

boolean anyHighSalary = employeeList.stream().anyMatch(e -> e.getSalary() > 50000);

Q18. How to check if all employees are active (if active field exists)?

boolean allActive = employeeList.stream().allMatch(Employee::isActive);

Q19. How to get a comma-separated string of employee names?

String names =employeeList.stream().map(Employee::getEmpName).collect(Collectors.joining(", "));

Q20. How to get the youngest employee?

Optional<Employee> youngest = employeeList.stream().min(Comparator.comparing(Employee::getAge));

Q21. How to get employees who joined after 2015?

List<Employee> recentJoinees = employeeList.stream().filter(e -> e.getYearOfJoining() > 2015)

.collect(Collectors.toList());

Q22. How to get employee names in uppercase?

employeeList.stream().map(e -> e.getEmpName().toUpperCase()).forEach(System.out::println);

Q23. How to get the second highest salary?

Optional<Employee> secondHighest = employeeList.stream().sorted(Comparator.comparing(Employee::getSalary)

.reversed()).skip(1)

.findFirst();

Q24. How to partition employees by gender?

Map<Boolean, List<Employee>> partitionByGender = employeeList.stream()

.collect(Collectors.partitioningBy(e -> "Male".equals(e.getGender())));

Q25. How to find employees with salary greater than the average salary?

double avgSalary = employeeList.stream().mapToDouble(Employee::getSalary).average().orElse(0.0);

List<Employee> aboveAverage = employeeList.stream().filter(e -> e.getSalary() > avgSalary)

.collect(Collectors.toList());

Q26. How to find the first employee who joined in 2011?

Optional<Employee> first2011 = employeeList.stream().filter(e -> e.getYearOfJoining() == 2011)

.findFirst();

Q27. How to find the highest paid 3 employees?

employeeList.stream().sorted(Comparator.comparing(Employee::getSalary).reversed()).limit(3)

.forEach(System.out::println);

Q28. How to group employees by gender and department?

Map<String, Map<String, List<Employee>>> grouped = employeeList.stream().collect(Collectors.groupingBy(Employee::getGender,Collectors.groupingBy(Employee::getDepartment)));

Q29. How to list employees by department sorted by name?

Map<String, List<Employee>> sortedDept = employeeList.stream()

.collect(Collectors.groupingBy(Employee::getDepartment,

Collectors.collectingAndThen(Collectors.toList(),list -> list.stream()

.sorted(Comparator.comparing(Employee::getEmpName)).collect(Collectors.toLis)));

Q30. How to get salary statistics (min, max, average, sum)?

DoubleSummaryStatistics stats = employeeList.stream().collect(Collectors.summarizingDouble(Employee::getSalary));

**Singleton Design Pattern:**

public class Singleton {

private static Singleton instance;

private Singleton() {}

public static synchronized Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}