Q1. Can you explain how to implement data caching to minimize repeated calls to an external API?
For example, if we have an API that retrieves an access token, what strategies can we use to cache the access token effectively?
Additionally, what cache refresh policies would you recommend to ensure that the data remains current?
This version clearly outlines your query while maintaining a professional tone suitable for an interview.

API Token Caching:

Data caching is used to avoid repeated external API calls by storing frequently used data (like access tokens) temporarily and reusing it until it expires

🔹 How to Cache an API Access Token

1. Call the authentication API once to get the token.
2. Store the token in cache (Redis / in-memory) along with its expiry time (TTL).
3. Before every API call:
    - If token is valid → reuse it
    - If expired or near expiry → fetch a new token and update cache

---

🔹 Cache Refresh Strategies

- **TTL-based expiry**: Token auto-expires after its lifetime.
- **Proactive refresh**: Refresh token a little before expiry (buffer time).
- **Lazy refresh**: Refresh only when a token is requested and expired.
- **Background refresh**: Scheduled refresh for high-traffic systems.

🔹 Types of Cache

- **In-memory** (Caffeine, HashMap): Fast, single instance.
- **Distributed** (Redis): Scalable, multi-instance apps.
- **Persistent**: Disk-based, survives restarts.

.

Q2. Final keyword usage

```
String a = "Tata";
String b = a;
a = "Birla";
```

Memory Understanding

```
String Pool:
"Tata"  <---- a , b
```

After reassignment:

```
"Tata"  <---- b
"Birla" <---- a
```

✔ `b` still refers to `"Tata"`

Key Points

- **Strings are immutable**
- Assignment does **not modify object**, only reference
- `final String a = "Tata";` → you cannot reassign `a`

Q3. Difference between Comparable and Comparator along with compare and compareTo.

Comparable (Natural Order)

```
class Student implements Comparable<Student> {
    int age;

    public int compareTo(Student s) {
        return this.age - s.age;
    }
}

Collections.sort(studentList);
```

✔ Sorting logic is **inside class**

Comparator (Custom Order)

```
class NameComparator implements Comparator<Student> {
    public int compare(Student s1, Student s2) {
        return s1.name.compareTo(s2.name);
    }
}

Collections.sort(studentList, new NameComparator());
```

✔ Sorting logic is **external**

Interview Tip

Use **Comparable** when one natural order
Use **Comparator** when multiple sorting strategies

Q4. How will you test a method that will throw exception and how will you assert its msg using junit 5 (use: assertThrows() method)

In JUnit 5, I use `assertThrows()` to verify the exception type and then assert the exception message using `getMessage()`.

Method

```
void validate(int age) {
    if(age < 18)
        throw new IllegalArgumentException("Age must be >= 18");
}
```

Test Case

```
@Test
void testValidate() {
    IllegalArgumentException ex =
        assertThrows(IllegalArgumentException.class,
            () -> validate(15));

    assertEquals("Age must be >= 18", ex.getMessage());
}
```

✔ Exception type
✔ Message verification

Q5. Immutable class ?

An immutable class is a class whose object state cannot be changed after creation, achieved by making the class final, fields private final, no setters, and using defensive copies.

🔹 Rules to Create an Immutable Class

1. **Make the class `final`**
   → Prevents subclassing

2. **Make all fields `private final`**
   → State cannot change

3. **No setters**
   → No external modification

4. **Initialize fields via constructor only**

5. **Return defensive copies for mutable fields**

```
final class Employee {
    private final int id;

    public Employee(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }
}
```

Why Immutable?

- `final class` → no subclass modification
- `final fields` → no reassignment
- No setters

✔ Thread-safe
✔ Cache-friendly

Immutable Class with List (Very Important Interview Question)

A `List` is mutable, so just making it `final` is NOT enough.
You must use defensive copying + unmodifiable wrappers.

❌ WRONG (Looks Immutable but IS NOT)

```java
import java.util.List;

public final class Employee {

    private final List<String> skills;

    public Employee(List<String> skills) {
        this.skills = skills; // ❌ reference leak
    }

    public List<String> getSkills() {
        return skills; // ❌ external modification possible
    }
}
```

Problem:

```java
List<String> list = new ArrayList<>();
list.add("Java");

Employee e = new Employee(list);
```

```
e.getSkills().add("Spring"); // ❌ Object modified!
```

✅ CORRECT Immutable Class with List

```
import java.util.*;

public final class Employee {

    private final List<String> skills;

    public Employee(List<String> skills) {
        // Defensive copy + unmodifiable
        this.skills = Collections.unmodifiableList(new ArrayList<>(skills));
    }

    public List<String> getSkills() {
        return skills; // Safe to return
    }
}
```

🔍 Why This Works

1. **Defensive copy prevents constructor reference leak**
2. **Unmodifiable list prevents runtime modification**
3. **No setters**
4. **Class is `final`**

🔬 Proof (Test)

```
List<String> skills = new ArrayList<>();

skills.add("Java");


Employee emp = new Employee(skills);


// External change

skills.add("Spring");

System.out.println(emp.getSkills()); // [Java]


// Internal change attempt

   emp.getSkills().add("Docker"); // ❌ UnsupportedOperationException
```

Q6. What's concurrentModificationException? Many users are accessing the same list (adding, removing and deleting the records). Does java provide any out of the box solution for that?

`ConcurrentModificationException` is a **runtime exception** thrown when a **collection is structurally modified while it is being iterated without proper synchronization**.

➡️ Common with `ArrayList`, `HashMap`, `HashSet`

🔍 Simple Example

```
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
list.add(3);

for (Integer i : list) {
    if (i == 2) {
        list.remove(i); // ❌ CME
    }
}
```

❌ Why?

- Enhanced `for` loop uses an **Iterator**
- `ArrayList` iterators are **fail-fast**
- Structural modification detected → throws exception

🧠 Internals (Interview Gold ⭐)

Most collections maintain:

- `modCount` → number of structural modifications
- `expectedModCount` → iterator snapshot

If:

```
modCount != expectedModCount
```

➡️ `ConcurrentModificationException`

🧪 When does it happen?

- One thread iterating + another modifying
- Same thread modifying directly while iterating
- Multi-threaded access without synchronization

❓ Many Users Accessing Same List – What to Do?

Yes ✅ **Java provides out-of-the-box solutions**

✅ Solution 1: Use `Iterator.remove()` (Single Thread)

```
Iterator<Integer> it = list.iterator();
while (it.hasNext()) {
    if (it.next() == 2) {
        it.remove(); // ✅ Safe
    }
}
```

✔ Works only within same iterator

✅ Solution 2: `Collections.synchronizedList()` (Thread-safe but fail-fast)

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

⚠️ Still need **manual synchronization during iteration**

```
synchronized (list) {
    for (Integer i : list) {
        System.out.println(i);
    }
}
```

✔ Thread-safe
❌ Can still throw CME if not synchronized properly

✅ Solution 3: `CopyOnWriteArrayList` ⭐ (BEST for Many Readers)

```
import java.util.concurrent.CopyOnWriteArrayList;

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

🔹 How it works

- Creates a **new copy** on every write
- Iterators work on **snapshot**
- ❌ No `ConcurrentModificationException`

```
for (Integer i : list) {
    list.remove(i); // ✅ No CME
}
```

✔ Pros

- Thread-safe

- No locking during iteration
- No CME

❌ Cons

- Costly for frequent writes
- Higher memory usage

✅ Solution 4: Use Concurrent Collections

Use Case	Collection
Key-Value	`ConcurrentHashMap`
Queue	`ConcurrentLinkedQueue`
Set	`ConcurrentSkipListSet`

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

✔ No CME
✔ High performance

Q7. ways to create a string? What's the difference ?

String literals reuse objects from the String Constant Pool, while new String() always creates a new object in heap memory.

1️⃣ String Literal

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

- Stored in **String Constant Pool (SCP)**
- Reuses existing object if value already exists
- **Memory efficient**
- Recommended way

2️⃣ Using `new` Keyword

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

- Creates **new object in Heap**
- Does **not reuse** SCP object
- Less memory efficient

Key Difference

```
String s1 = "Java";

String s2 = "Java";

String s3 = new String("Java");


System.out.println(s1 == s2); // true (same SCP reference)

System.out.println(s1 == s3); // false (different objects)
```

```
System.out.println(s1.equals(s3)); // true (same content)
```

Q8. Overloading vs Overriding (Execution Flow)

Overloading

```
void add(int a, int b) {}

void add(double a, double b) {}
```

✔ Compile-time binding

Overriding

```
class A {

    void show() {}

}


class B extends A {

    void show() {}

}
```

✔ Runtime polymorphism

Q9. Java work as Pass by value or Pass by reference (It was tricky question)

Java is strictly pass-by-value; when objects are passed, a copy of the reference is passed, not the reference itself.

👉 **Java is ALWAYS *Pass by Value*** ✅
There is **NO pass by reference in Java**.

🔹 Why Is It Tricky?

When you pass an object, Java passes **a copy of the reference**, not the actual object reference.

🔹 Example 1: Primitive Type

```
void change(int x) {
    x = 20;
}

int a = 10;
change(a);
System.out.println(a); // 10
```

✔ Value copy → no change

🔹 Example 2: Object (Reference Copy)

```
class Test {
    int x;
}

void modify(Test t) {
    t.x = 20;
}

Test obj = new Test();
obj.x = 10;
modify(obj);
System.out.println(obj.x); // 20
```

✔ Object state changes
❌ Reference not changed

🔹 Example 3: Reassigning Object (Important)

```
void change(Test t) {
    t = new Test();
    t.x = 50;
}

Test obj = new Test();
obj.x = 10;
change(obj);
System.out.println(obj.x); // 10
```

✔ Reassignment does NOT affect original object

🔑 Key Understanding

- Java passes **value**
- For objects → value = **copy of reference**
- You can modify object state
- You cannot change original reference

Q10. ExecutorService – Real Example

What is it?
`ExecutorService` is a **Java concurrency framework** used to **manage and execute threads** instead of creating threads manually.

Why use ExecutorService?

- Reuses threads (thread pool)
- Better performance
- Easy task submission
- Proper lifecycle management

Key Interfaces & Classes

- `Executor` → execute tasks
- `ExecutorService` → manage threads + shutdown
- `Executors` → factory methods

Common Thread Pools

```
ExecutorService es = Executors.newFixedThreadPool(5);
ExecutorService es = Executors.newCachedThreadPool();
ExecutorService es = Executors.newSingleThreadExecutor();
```

```
es.execute(() -> System.out.println("Runnable"));

Future<Integer> f = es.submit(() -> 10);

System.out.println(f.get()); // 10
```

```
es.shutdown();        // graceful

es.shutdownNow();     // force
```

🔹 Why Not Use `new Thread()`?

Problems with manual threads:

- **Thread creation is expensive**
- **No reuse of threads**
- **Difficult lifecycle management**
- **Risk of resource exhaustion**

✔ `ExecutorService` solves all of this using Thread Pools

🔹 Executor Framework Hierarchy

```
Executor

    ↓

ExecutorService

    ↓

ScheduledExecutorService
```

🔹 Core Components

1️⃣ Task

- `Runnable` **→ no return value**
- `Callable<V>` **→ returns result, can throw exception**

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

```java
Callable<Integer> c = () -> 10;
```

2️⃣ Thread Pool

A set of reusable worker threads.

Created using `Executors` factory class.

🔹 Types of ExecutorService (Very Important)

✅ Fixed Thread Pool

```java
ExecutorService es = Executors.newFixedThreadPool(3);
```

- **Fixed number of threads**
- **Excess tasks go to queue**
- **Best for stable workloads**

✅ Cached Thread Pool

```java
ExecutorService es = Executors.newCachedThreadPool();
```

- **Creates threads as needed**
- **Reuses idle threads**
- **Risky for heavy load**

✅ Single Thread Executor

```
ExecutorService es = Executors.newSingleThreadExecutor();
```

- One thread only
- Tasks executed sequentially
- Thread replacement on failure

✅ Scheduled Executor

```
ScheduledExecutorService ses =
        Executors.newScheduledThreadPool(2);
```

- Delay / periodic execution
- Replacement for `Timer`

🔹 execute() vs submit()

execute()	submit()
From Executor	From ExecutorService
No return value	Returns `Future`

Exceptions not-trackable	Exceptions captured

```
es.execute(() -> System.out.println("Run"));
```

```
Future<Integer> f = es.submit(() -> 100);
System.out.println(f.get());
```

🔹 Future (Result Handling)

```
Future<Integer> future = es.submit(() -> {
    Thread.sleep(1000);
    return 42;
});

Integer result = future.get(); // blocks
```

Future Methods

- `get()` → wait for result
- `isDone()`
- `cancel()`

🔹 Shutdown Lifecycle (VERY IMPORTANT)

Graceful Shutdown

```
es.shutdown();
```

- Stops accepting new tasks
- Completes existing tasks

Force Shutdown

```
es.shutdownNow();
```

- Attempts to stop running tasks
- Returns pending tasks

Best Practice

```
try {
    // submit tasks
} finally {
    es.shutdown();
}
```

🔹 ThreadPoolExecutor (Internal Engine)

`Executors` use `ThreadPoolExecutor` internally.

```
ThreadPoolExecutor executor =
    new ThreadPoolExecutor(
        2,          // core threads
        5,          // max threads
        60,         // idle timeout
        TimeUnit.SECONDS,
        new LinkedBlockingQueue<>(10)
    );
```

Key Parameters

- `corePoolSize`
- `maximumPoolSize`
- `workQueue`
- `RejectedExecutionHandler`

🔹 Real-Time Example (Production Style)

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

for (int i = 0; i < 10; i++) {
    int taskId = i;
```

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

es.shutdown();
```

🔹 Common Interview Questions

Q. Is ExecutorService thread-safe?
✔ Yes

Q. Can we reuse ExecutorService?
✔ Yes, until shutdown

Q. What happens if it is not shutdown?
❌ JVM may not exit (non-daemon threads)

Q11. CountDownLatch vs CyclicBarrier (Real Use Case)

Both are **thread synchronization utilities** from `java.util.concurrent`, used to make threads wait until a condition is met.

🔹 CountDownLatch

What it does:
Allows one or more threads to wait until a **count reaches zero**.

Key points

- One-time use ❌ (cannot be reset)
- Threads wait on `await()`
- Other threads reduce count using `countDown()`

Example

```
CountDownLatch latch = new CountDownLatch(3);


latch.await();        // waits


latch.countDown();   // called by worker threads
```

Use case

- Main thread waits for multiple services to start
- Wait for all tasks to complete before proceeding

🔹 CyclicBarrier

What it does:
Makes a **fixed number of threads wait for each other** at a common point.

Key points

- Reusable ✅ (cyclic)
- Threads wait using `await()`
- Optional barrier action when all arrive

Example

```
CyclicBarrier barrier = new CyclicBarrier(3);


barrier.await(); // all 3 threads must reach here
```

Use case

- Parallel processing phases
- Multi-threaded algorithms (map-reduce style)

🔑 Key Differences

Feature	CountDownLatch	CyclicBarrier
Reusable	❌ No	✅ Yes
Who waits	One or more threads	All threads

Reset	❌ Not possible	✅ Automatic
Barrier action	❌ No	✅ Yes
Common use	Waiting for tasks	Synchronizing phases

`CountDownLatch waits until a count reaches zero and is one-time, while CyclicBarrier makes a fixed number of threads wait for each other and is reusable.`

Q12. Thread Pools – When to Use

Java provides Fixed, Cached, Single, Scheduled, and Work-Stealing thread pools to handle different concurrency scenarios efficiently.

Pool	Use Case
Fixed	Limited resources
Cached	Short-lived tasks
Single	Sequential execution
Scheduled	Delayed tasks

Q13. Use of Default method in java8

Default methods allow interfaces to have method implementations, mainly to maintain backward compatibility and enable code reuse in Java 8.

Why Default Methods Were Introduced

1. **Backward compatibility**
   → Add new methods to interfaces **without breaking existing implementations**

2. **Code reuse**
   → Share common behavior across implementing classes

3. **Support functional-style APIs**
   → Used heavily in Java 8 APIs (`Collection`, `Stream`)

Example

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

class Car implements Vehicle {
    // no need to override
}
```

Multiple Inheritance Conflict (Important)

```
interface A {
    default void show() { }
}

interface B {
    default void show() { }
}
```

```java
class C implements A, B {
    public void show() {    // must override
        A.super.show();
    }
}
```

Q14. Abstract vs Interface (Design Level)

Abstract classes support inheritance with state and constructors, while interfaces support multiple inheritance and define behavior contracts.

Points	Abstract Class	Interface
Definition	Cannot be instantiated; contains both abstract (without implementation) and concrete methods (with implementation)	Specifies a set of methods a class must implement; methods are abstract by default.
Implementation Method	Can have both implemented and abstract methods.	Methods are abstract by default; Java 8, can have default and static methods.
Inheritance	class can inherit from only one abstract class.	A class can implement multiple interfaces.
Access Modifiers	Methods and properties can have any access modifier (public, protected, private).	Methods and properties are implicitly public.
Variables	Can have member variables (final, non-final, static, non-static).	Variables are implicitly public, static, and final (constants).

✔ Abstract → IS-A + behavior
✔ Interface → CAN-DO

Q15. Best Way to Create Thread

Java provides multiple ways to create threads. The main 4 are asked in interviews.

Threads can be created by extending Thread, implementing Runnable or Callable, or using ExecutorService, with ExecutorService being the recommended approach.

🔹 1. Extend `Thread` Class

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

MyThread t = new MyThread();
t.start();
```

❌ **Limitation: Cannot extend another class**

🔹 2. Implement `Runnable` Interface ⭐ (Preferred)

```
class MyTask implements Runnable {
    public void run() {
        System.out.println("Thread running");
    }
}

Thread t = new Thread(new MyTask());
t.start();
```

✅ **Advantage: Supports inheritance**

🔹 3. Implement `Callable` (Java 5+)

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

FutureTask<Integer> ft = new FutureTask<>(task);


Thread t = new Thread(ft);

t.start();


System.out.println(ft.get());
```

✅ **Can return result & throw exception**

🔹 4. Using ExecutorService ⭐⭐⭐ (Best Practice)

```
ExecutorService es = Executors.newFixedThreadPool(2);


es.submit(() -> System.out.println("Thread running"));

es.shutdown();
```

✅ **Thread pooling**
✅ **Production standard**

🔹 Java 8 Lambda Way

```
new Thread(() -> System.out.println("Lambda thread")).start();
```

🔑 Comparison Summary

Way	Return Value	Recommended
Thread	❌	❌
Runnable	❌	✅
Callable	✅	✅
ExecutorService	✅	⭐⭐⭐

Q16. Best Approach to Create Thread.

The best approach to create threads in Java is using ExecutorService because it provides thread pooling, better performance, and proper lifecycle management.

❌ Why NOT `new Thread()`?

- Creates a new thread every time (expensive)

- No reuse
- Hard to manage lifecycle
- Poor scalability

✅ Best Approach: `ExecutorService`

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


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


executor.shutdown();
```

🔹 Why ExecutorService is Best?

- ✅ Thread pooling (reuse threads)
- ✅ Better performance
- ✅ Handles multiple users safely
- ✅ Easy shutdown & lifecycle control
- ✅ Production standard

🔹 When to Use What? (Quick Tip)

Scenario	Best Choice
Small demo	Runnable
Need result	Callable
Production apps	ExecutorService

Q17. Map vs FlatMap (Visual)

`map()` transforms elements one-to-one, while `flatMap()` transforms and flattens nested structures into a single stream.

🔹 `map()`

What it does:
Transforms **each element into exactly one element**.

1 → 1 mapping

Example

```
List<String> names = List.of("java", "spring");

List<String> result =
    names.stream()
         .map(String::toUpperCase)
         .toList();
```

```
System.out.println(result); // [JAVA, SPRING]
```

🔹 `flatMap()`

What it does:
Transforms **each element into multiple elements** and **flattens** them into a single stream.

1 → many → flattened

Example

```
List<List<String>> list =
    List.of(List.of("A", "B"), List.of("C", "D"));

List<String> result =
    list.stream()
        .flatMap(List::stream)
        .toList();

System.out.println(result); // [A, B, C, D]
```

Q18. Parallel Stream – Internal

👉A **parallel stream** splits data into multiple parts and processes them **concurrently** using multiple threads.

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

🔹 Internal Working (Step by Step)

1️⃣Splitting the Data

- Uses `Spliterator`
- Breaks the source into smaller chunks

```
Spliterator spliterator = list.spliterator();
```

2️⃣ Thread Management

- Uses **ForkJoinPool.commonPool**
- Pool size ≈ **number of CPU cores**

```
ForkJoinPool.commonPool();
```

3️⃣ Fork–Join Model

- Tasks are **forked** into subtasks
- Idle threads **steal work** (work-stealing)

4️⃣ Processing in Parallel

- Each chunk processed independently
- Operations must be **stateless & non-blocking**

5️⃣ Merging the Result

- Partial results are **combined**
- Order may not be preserved

🔹 Simple Flow Diagram

```
Data Source
    ↓
Spliterator (split)
    ↓
ForkJoinPool (threads)
    ↓
Parallel Processing
    ↓
Combine Results
```

🔹 When to Use Parallel Streams?

✔ Large data
✔ CPU-intensive tasks
✔ Stateless operations

❌ Small collections
❌ I/O or synchronized logic

=> when we choose Stream over parallel Stream?

👉 We choose Stream over Parallel Stream when data is small, order is important, tasks are I/O-bound, or thread safety and predictability are required.

✅ Choose Stream when:

1️⃣ Data size is small

- Parallel overhead > benefit

2️⃣ Operations are I/O or blocking

- DB calls, REST calls, file I/O

3️⃣ Order matters

```
stream().forEach(...) // preserves order
```

4️⃣ **Operations are stateful or synchronized**

- Shared variables
- Locks

5️⃣ **Running in application servers**

- Parallel streams use **common ForkJoinPool**
- Can impact other tasks

6️⃣ **Predictable & debuggable behavior needed**

- Easier to debug
- Fewer concurrency issues

❌ Avoid Parallel Stream when:

- Using `synchronized`
- Modifying shared mutable data
- Calling external services
- Data is small

Q19. Explain completable future?

🔹 **What is `CompletableFuture`?**

CompletableFuture is an asynchronous, non-blocking API that allows chaining, combining, and handling results of async computations efficiently.

🔹 Why `CompletableFuture` over `Future`?

Future	CompletableFuture
Blocking (`get()`)	Non-blocking
No chaining	Supports chaining
No easy exception handling	Built-in exception handling
No combining tasks	Combine multiple async tasks

🔹 Basic Example

```
CompletableFuture<String> cf =
    CompletableFuture.supplyAsync(() -> "Hello");

System.out.println(cf.get()); // Hello
```

Runs in **ForkJoinPool.commonPool** by default.

🔹 Core Methods (Most Important)

1️⃣ Run Async (No Result)

```
CompletableFuture.runAsync(() ->
    System.out.println("Running async"));
```

2️⃣ Supply Async (With Result)

```java
CompletableFuture<Integer> cf =
    CompletableFuture.supplyAsync(() -> 10);
```

🔹 Chaining Operations

`thenApply()` → transform result

```java
CompletableFuture<Integer> cf =
    CompletableFuture.supplyAsync(() -> 10)
                     .thenApply(x -> x * 2);
```

`thenAccept()` → consume result

```java
cf.thenAccept(System.out::println);
```

`thenRun()` → no input, no output

```java
cf.thenRun(() -> System.out.println("Done"));
```

🔹 Combining Multiple Futures

`thenCombine()` (Both results needed)

```java
CompletableFuture<Integer> f1 =
    CompletableFuture.supplyAsync(() -> 10);

CompletableFuture<Integer> f2 =
    CompletableFuture.supplyAsync(() -> 20);

CompletableFuture<Integer> result =
    f1.thenCombine(f2, Integer::sum);
```

`allOf()` / `anyOf()`

```
CompletableFuture.allOf(f1, f2).join();
```

🔹 Exception Handling ⭐

`exceptionally()`

```
cf.exceptionally(ex -> 0);
```

`handle()` (success or failure)

```
cf.handle((res, ex) -> ex == null ? res : 0);
```

🔹 Custom Executor

```
ExecutorService executor = Executors.newFixedThreadPool(3);

CompletableFuture.supplyAsync(() -> "Task", executor);
```

🔹 CompletableFuture vs ExecutorService

ExecutorService	CompletableFuture
Task execution	Async workflow
Manual chaining	Built-in chaining
Blocking style	Non-blocking

👉 **Best used together**

🔹 Real-World Use Case

- Parallel service calls
- Microservices aggregation
- Async REST APIs
- Background processing

Q20. What is a Future Object?

Future represents the result of an asynchronous task and provides methods to check completion, retrieve results, or cancel execution.

🔹 How Future Works

1. Task is submitted to `ExecutorService`
2. Method returns a `Future`
3. Task executes in background
4. Result is retrieved later

🔹 Simple Example

```
ExecutorService executor = Executors.newSingleThreadExecutor();


Future<Integer> future =

    executor.submit(() -> {
```

```
        Thread.sleep(1000);
        return 42;
    });

System.out.println(future.get()); // blocks until result is ready
executor.shutdown();
```

🔹 Important Future Methods

Method	Purpose
`get()`	Get result (blocking)
`get(timeout)`	Timed wait
`isDone()`	Task completed?
`isCancelled()`	Task cancelled?

`cancel(true)`	Cancel task

🔹 Limitations of Future ❌

- Blocking (`get()`)
- No chaining of tasks
- Poor exception handling
- Cannot combine multiple futures

➡️ These are solved by **CompletableFuture**

Q21 . HashMap internal working?

`HashMap` stores key–value pairs and provides O(1) average time complexity for `put()` and `get()`.

HashMap stores data in buckets using hashCode; collisions are handled via linked lists or red-black trees, providing O(1) average lookup time.

🔹 Core Data Structure

- **Backed by an array of buckets**
- **Each bucket stores:**
  - **LinkedList (before Java 8)**
  - **Red-Black Tree (Java 8+, if collisions are high)**

```java
Node<K,V>[] table;
```

🔹 How `put()` Works (Step-by-Step)

```java
map.put(key, value);
```

1️⃣ Compute hash

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

2️⃣ Index calculation

```java
index = (n - 1) & hash;   // n = table size
```

3️⃣ Check bucket

- If empty → insert node
- If occupied:
    - Same key → replace value
    - Different key → collision

4️⃣ Collision handling

- Add to LinkedList
- If chain length > 8, convert to Red-Black Tree

5️⃣ Resize (Rehashing)

- Happens when size > `capacity × loadFactor`
- Default load factor = 0.75

🔹 How `get()` Works

```
map.get(key);
```

1️⃣ Compute hash
2️⃣ Find bucket index
3️⃣ Traverse list/tree
4️⃣ Match key using `equals()`

🔹 Collision Handling (Very Important)

Java Version	Technique
Java 7	LinkedList
Java 8+	LinkedList → Red-Black Tree

✔ Improves worst-case from O(n) → O(log n)

🔹 Important Defaults

Property	Value
Initial capacity	16
Load factor	0.75
Treeify threshold	8
Untreeify threshold	6

🔹 Why `hashCode()` & `equals()` Matter

- `hashCode()` → bucket selection
- `equals()` → key comparison inside bucket

❌ Wrong implementation → data loss / duplicates

🔹 Thread Safety

- `HashMap` is NOT thread-safe
- Use:

- `Collections.synchronizedMap()`
- `ConcurrentHashMap` **(preferred)**

=> Equal and hashcode method contract?

Java defines a strict contract between `equals()` and `hashCode()` in `Object` class.

If two objects are equal according to equals(), they must have the same hashCode; otherwise HashMap and HashSet will not work correctly.

🔹 equals() Contract

For any non-null references `x`, `y`, `z`:

1️⃣ Reflexive

`x.equals(x) == true`

2️⃣ Symmetric

`x.equals(y) == y.equals(x)`

3️⃣ Transitive

`x.equals(y) && y.equals(z) ⇒ x.equals(z)`

4️⃣ Consistent

- **Multiple calls return same result (if no change)**

5️⃣ Non-null

`x.equals(null) == false`

🔹 hashCode() Contract

1️⃣ If two objects are equal, they must have same hashCode

```
x.equals(y) == true ⇒ x.hashCode() == y.hashCode()
```

2️⃣ If objects are not equal, hashCodes may or may not be different

---

🔹 Combined Contract (Most Important)

Case	Allowed?
equals = true, hashCode same	✅ Mandatory
equals = false, hashCode same	✅ Allowed (collision)
equals = true, hashCode different	❌ NOT allowed

🔹 Why This Contract Matters?

- Used by HashMap, HashSet

- **Violating contract leads to:**
  - **Duplicate keys**
  - **Data not found**
  - **Performance issues**

🔹 Correct Implementation Example

```
class Employee {
    int id;
    String name;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Employee)) return false;
        Employee e = (Employee) o;
        return id == e.id && name.equals(e.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(id, name);
```

```
    }
}
```

=> If we override only equal method, will it work?

Overriding only equals() breaks the equals–hashCode contract and causes incorrect behavior in hash-based collections.

✅ Quick Rule to Remember

> **Override `equals()` ⇒ ALWAYS override `hashCode()`**

🔹 Why Not?

Java's contract says:

> **If two objects are equal (`equals()` returns true), they MUST have the same `hashCode()`**

If you override only `equals()`:

- Objects may be **equal**
- But they can end up in **different hash buckets**

➡️ Collections like `HashMap` and `HashSet` will **fail**

🔹 Example (Problem Case)

```
class Employee {
    int id;

    Employee(int id) {
        this.id = id;
```

```
    }

    @Override
    public boolean equals(Object o) {
        Employee e = (Employee) o;
        return this.id == e.id;
    }
}
```

Usage:

```
Employee e1 = new Employee(1);
Employee e2 = new Employee(1);

System.out.println(e1.equals(e2)); // true

Set<Employee> set = new HashSet<>();
set.add(e1);
set.add(e2);

System.out.println(set.size()); // 2 ❌ should be 1
```

🔹 Why This Happens?

- Default `hashCode()` (from `Object`) uses memory address
- Different objects → different hashCodes
- HashSet thinks they are **different keys**

🔹 Correct Way (Fix)

```
@Override

public int hashCode() {

    return Objects.hash(id);

}
```

Q22. What is a connection pool in the context of database management, and what are the advantages of using it?

What is a Connection Pool?

A connection pool is a cache of pre-created, reusable database connections that applications borrow and return instead of opening a new connection for every database request.

How It Works (Simple Flow)

1. Application requests a DB connection
2. Pool gives an available connection
3. Application uses it
4. Connection is returned to the pool (not closed)

Advantages of Using a Connection Pool

1️⃣ Better Performance

- Avoids costly creation/destruction of DB connections

2️⃣ Scalability

- Limits max connections → prevents DB overload

3️⃣ Resource Management

- Reuses connections efficiently

4️⃣ Faster Response Time

- Immediate availability of connections

5️⃣ Connection Validation & Recovery

- Pools detect and replace broken connections

6️⃣ Thread Safety

- Manages concurrent access safely

Real-World Example

In Java (Spring Boot):

- HikariCP (default)
- Apache DBCP

Q23. How does implement a connection pool class as a singleton, and why is it important to provide a get connection method?

A connection pool is implemented as a Singleton to ensure a single shared pool of database connections, and a getConnection() method is essential to safely manage, reuse, and control access to those connections.

🔹 Why `getConnection()` Method Is Important?

- Controls **how connections are borrowed**
- Ensures:
  - Thread safety
  - Max connection limit
  - Reuse of existing connections
- Prevents direct access to internal pool structure

👉 Without `getConnection()`, clients could misuse or leak connections.

🔹 Simple Connection Pool Singleton Implementation

1️⃣ ConnectionPool Class (Singleton)

```
import java.sql.Connection;

import java.sql.DriverManager;

import java.util.LinkedList;

import java.util.Queue;


public class ConnectionPool {
```

```java
}


private static ConnectionPool instance;


private static final int MAX_CONNECTIONS = 5;
private Queue<Connection> pool = new LinkedList<>();


private ConnectionPool() {
    for (int i = 0; i < MAX_CONNECTIONS; i++) {
        pool.add(createConnection());
    }
}


private Connection createConnection() {
    try {
        return DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/db",
            "user",
            "password"
        );
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

```java
// Singleton instance
public static synchronized ConnectionPool getInstance() {
    if (instance == null) {
        instance = new ConnectionPool();
    }
    return instance;
}
```

---

2 `getConnection()` Method

```java
public synchronized Connection getConnection() {
    if (pool.isEmpty()) {
        throw new RuntimeException("No connections available");
    }
    return pool.poll();
}
```

---

3 `releaseConnection()` Method

```java
public synchronized void releaseConnection(Connection connection) {
    pool.offer(connection);
}
```

```
}
```

🔹 How Client Uses It

```
ConnectionPool pool = ConnectionPool.getInstance();
Connection con = pool.getConnection();

// use connection

pool.releaseConnection(con);
```

🔹 Key Benefits of This Design

Feature	Benefit
Singleton	Single shared pool
getConnection()	Controlled access
Thread-safe	Safe for multiple users
Reuse	Better performance

Limit	Prevents DB overload

Q24. What is the purpose of the double-checked locking pattern in a multithreaded environment, and how does it prevent race conditions?

Double-checked locking safely creates a singleton in a multithreaded environment by minimizing synchronization and preventing race conditions using synchronized blocks and a volatile instance reference.

🔹 What is Double-Checked Locking?

Double-checked locking is a design pattern used to **safely initialize a singleton object lazily** in a **multithreaded environment**, while **minimizing synchronization overhead**.

🔹 The Problem It Solves

Without proper synchronization:

- Multiple threads can create **multiple instances** of a singleton
- Synchronizing every call is **slow**

DCL ensures:

- Thread safety ✅
- High performance ✅

🔹 How It Works (Step-by-Step)

1️⃣ First check (without lock)

- Avoids synchronization once the instance is created

2️⃣ **Lock only if needed**

- Synchronize only when instance is `null`

3️⃣ **Second check (with lock)**

- Prevents two threads from creating the instance at the same time

---

🔹 Correct Implementation (Java 5+)

```
public class Singleton {


    private static volatile Singleton instance;


    private Singleton() {}


    public static Singleton getInstance() {
        if (instance == null) {                // 1st check
            synchronized (Singleton.class) {
                if (instance == null) {        // 2nd check
                    instance = new Singleton();
                }
            }
        }
```

```
        return instance;
    }
}
```

🔹 Why `volatile` Is CRITICAL ⚠️

Without `volatile`, object creation can be **reordered**:

1. Memory allocation
2. Reference assignment
3. Constructor execution

Another thread may see a **partially constructed object**.

👉 `volatile`:

- Prevents instruction reordering
- Ensures visibility across threads

🔹 How It Prevents Race Conditions

Mechanism	Role
First null check	Avoids unnecessary locking

`synchronized` block	Ensures only one thread creates instance
Second null check	Prevents duplicate creation
`volatile`	Guarantees safe publication

🔹 When to Use DCL?

- Lazy initialization required
- High-frequency access to instance
- Multithreaded environment

🔹 Alternatives (Often Better)

- **Eager initialization**
- **Static inner helper class**
- **Enum Singleton** ⭐ (Best)

=> Why is it important to perform a null check before acquiring a synchronized block in a double-checked locking pattern?

The first null check avoids unnecessary synchronization once the instance is created, improving performance, while the second null check inside the synchronized block ensures thread safety.

🔹 The Problem Without the First Null Check

```java
public static Singleton getInstance() {
    synchronized (Singleton.class) {
        if (instance == null) {
            instance = new Singleton();
        }
    }
    return instance;
}
```

- Every call acquires the lock ❌
- Even after the instance is already created
- Causes **performance bottleneck** in multi-threaded apps

🔹 Purpose of the First Null Check (Outside `synchronized`)

```java
if (instance == null) {   // First check (no lock)
    synchronized (Singleton.class) {
        if (instance == null) {  // Second check (with lock)
            instance = new Singleton();
        }
    }
}
```

✔ Why this matters:

1️⃣ Avoids locking after initialization

- Most calls just return the instance
- No synchronization cost

2️⃣ Improves performance under high concurrency

- Lock is acquired only once (during creation)

3️⃣ Reduces contention between threads

🔹 How Race Conditions Are Still Prevented

- Multiple threads may pass the **first null check**
- Only **one thread** enters the synchronized block
- **Second null check** ensures only one instance is created

🔹 Role of `volatile` (Important)

- Ensures visibility across threads
- Prevents partially constructed object from being seen

🔑 Key Takeaway

First check = performance optimization
Second check = thread safety

Q25. Deep Cloning?

Deep cloning creates a completely independent copy of an object along with all its nested objects, preventing shared references and side effects.

🔹 What is Deep Cloning?

Deep cloning means creating a **new object and new copies of all nested (mutable) objects** inside it.

➡️ Changes in the cloned object **do NOT affect** the original object.

🔹 Deep Cloning vs Shallow Cloning

Feature	Shallow Clone	Deep Clone
Object copy	New object	New object
Nested objects	Shared references	New copies
Side effects	Yes	No
Safety	❌ Risky	✅ Safe

🔹 Problem Example (Why Shallow Clone Fails)

```
class Address {

    String city;
```

```
}

class Person implements Cloneable {
    String name;
    Address address;

    protected Object clone() throws CloneNotSupportedException {
        return super.clone(); // ❌ shallow clone
    }
}

Person p1 = new Person();
p1.address = new Address();

Person p2 = (Person) p1.clone();
p2.address.city = "Delhi";

System.out.println(p1.address.city); // Delhi ❌
```

🔹 Deep Cloning Using `clone()` (Manual)

```
class Address {
```

```
    String city;

    Address(String city) {
        this.city = city;
    }
}

class Person implements Cloneable {
    String name;
    Address address;

    @Override
    protected Object clone() throws CloneNotSupportedException {
        Person cloned = (Person) super.clone();
        cloned.address = new Address(this.address.city); // deep copy
        return cloned;
    }
}
```

✔ Nested object copied separately

🔹 Deep Cloning Using Copy Constructor ⭐ (Preferred)

```java
class Address {
    String city;

    Address(Address a) {
        this.city = a.city;
    }
}

class Person {
    String name;
    Address address;

    Person(Person p) {
        this.name = p.name;
        this.address = new Address(p.address);
    }
}
```

✔ Clean
✔ No `Cloneable` issues
✔ Interview-friendly

🔹 Deep Cloning Using Serialization

```
// serialize → deserialize
```

✔ Easy
❌ Slow
❌ All classes must be Serializable

🔹 Best Practice (Interview Tip)

❌ Avoid `Cloneable`
✅ Use **copy constructors** or **factory methods**

Q26. Shallow Cloning?

Shallow cloning creates a new object but shares references to nested objects, so changes in one affect the other.

🔹 What is Shallow Cloning?

Shallow cloning creates a **new object**, but **copies references** of nested (mutable) objects instead of creating new ones.

➡️ Both original and clone **share the same internal objects**.

🔹 Key Characteristics

- New top-level object
- Nested objects are **not duplicated**
- Changes in nested objects affect both copies

🔹 Example of Shallow Cloning

```java
class Address {
    String city;
}

class Person implements Cloneable {
    String name;
    Address address;

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone(); // shallow clone
    }
}
```

🔬 Behavior

```java
Person p1 = new Person();
p1.address = new Address();
p1.address.city = "Delhi";

Person p2 = (Person) p1.clone();
p2.address.city = "Mumbai";
```

```
System.out.println(p1.address.city); // Mumbai ❌
```

🔹 Why This Happens?

- `super.clone()` copies:
    - Primitive values ✔
    - Object references ❌ (same memory)

🔹 When is Shallow Cloning Acceptable?

- All fields are **immutable** (`String`, `Integer`)
- Object is **effectively immutable**
- Performance is critical and sharing is safe

🔹 Shallow Clone vs Assignment

```
Person p2 = p1;  // same reference ❌
```

Assignment	Shallow Clone
Same object	New object

One reference	Two references

🔹 Shallow vs Deep Cloning (Quick)

Aspect	Shallow	Deep
Nested objects	Shared	Copied
Side effects	Yes	No
Safety	❌	✅

Q27. What is the use of try with resources?

`try-with-resources` (Java 7+) is used to automatically close resources such as files, database connections, and streams after use, even if an exception occurs.

🔹 Why It Is Needed

Without it, resources must be closed manually in `finally`, which is:

- **Error-prone**
- **Verbose**
- **Can cause resource leaks**

🔹 Basic Example

```
try (FileInputStream fis = new FileInputStream("data.txt")) {
    // use resource
}
```

➡️ `fis.close()` is called automatically

🔹 Conditions

- Resource must implement `AutoCloseable`
- `close()` is called implicitly

🔹 Multiple Resources

```
try (
    FileInputStream fis = new FileInputStream("a.txt");
    BufferedReader br = new BufferedReader(new
InputStreamReader(fis))
) {
    System.out.println(br.readLine());
}
```

🔹 Key Advantages

1️⃣ **Prevents resource leaks**
2️⃣ **Cleaner & shorter code**
3️⃣ **Exception-safe**
4️⃣ **Automatic closing in reverse order**

Q28. Use of finally?

The `finally` **block** is used to execute **important cleanup code** that must run **whether an exception occurs or not**.

🔹 Why `finally` Is Needed

- Ensures **resource cleanup**
- Guarantees execution after `try` / `catch`
- Used for closing files, DB connections, releasing locks

🔹 Basic Example

```
try {

    int a = 10 / 0;

} catch (ArithmeticException e) {

    System.out.println("Exception caught");

} finally {
```

```java
    System.out.println("Always executed");
}
```

✔ `finally` runs even after exception

🔹 Common Use Cases

- Closing resources
- Releasing locks
- Cleanup operations
- Logging

🔹 Does `finally` Always Execute?

❌ **No**, it will NOT execute if:

- `System.exit()` is called
- JVM crashes
- Power failure

🔹 `finally` with Return Statement

```java
try {
    return 10;
```

```
} finally {
    System.out.println("Executed");
}
```

✔ `finally` executes **before return**

⚠️ Avoid `return` inside `finally` (overrides return)

🔹 `finally` vs `try-with-resources`

finally	try-with-resources
Manual cleanup	Automatic cleanup
More code	Cleaner
Can be skipped by System.exit	Safer

Q29. LinkedHashMap - use case where we can use it. Example - LRU Cache

LinkedHashMap internally maintains a doubly linked list, which enables it to implement an efficient LRU cache by reordering entries on access and evicting the least recently used entry in O(1) time.

1️⃣ Why LinkedHashMap Exists

Problem with HashMap

`HashMap`:

- Stores data in **buckets**
- Gives **O(1)** average performance
- ❌ **Does NOT maintain order**

Example:

```
Map<Integer, String> map = new HashMap<>();
map.put(3, "C");
map.put(1, "A");
map.put(2, "B");

System.out.println(map);
// Output order is unpredictable
```

But many real-world systems need:

- Predictable order
- Cache eviction (LRU / FIFO)
- Recently used tracking

👉 **LinkedHashMap solves this problem**

2️⃣ What is LinkedHashMap?

`LinkedHashMap` is:

> A **HashMap + Doubly Linked List**

Internally:

- Uses **HashMap** for fast lookup
- Maintains a **doubly linked list** of entries

Each entry has:

```
before <---> current <---> after
```

3️⃣ Ordering Modes in LinkedHashMap

🔹 1. Insertion Order (Default)

Entries remain in the order they were inserted.

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

Example:

```
put(1) → put(2) → put(3)
Iteration: 1, 2, 3
```

🔹 2. Access Order (Important for LRU)

Entries are reordered **when accessed**

```
LinkedHashMap<Integer, String> map =
        new LinkedHashMap<>(16, 0.75f, true);
```

✔ Every `get()` or `put()` moves the entry to the **end**

4️⃣ How Access Order Works (Internally)

Initial state:

```
1 → 2 → 3
```

Access `get(1)`:

```
2 → 3 → 1
```

Add `put(4)`:

`2 → 3 → 1 → 4`

Now:

- **2** is least recently used
- **4** is most recently used

5️⃣ removeEldestEntry() – The Game Changer

This method decides **when to remove old entries**.

```
protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
```

- Called **after each put()**
- If returns `true`, eldest entry is removed

6️⃣ LRU Cache – What Is It?

LRU (Least Recently Used) Cache

- Removes the entry that was **least recently accessed**
- Used in:
    - Database connection pools
    - Browser cache
    - API token cache

- Operating systems

7️⃣ Implementing LRU Cache Using LinkedHashMap

Step-by-Step Design

Step 1: Extend LinkedHashMap

```
class LRUCache<K, V> extends LinkedHashMap<K, V> {
```

Step 2: Enable Access Order

```
super(capacity, 0.75f, true);
```

- `true` → access order
- This automatically maintains usage order

Step 3: Evict Old Entry

```
@Override
protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
    return size() > capacity;
}
```

✅ Final Implementation

```
import java.util.LinkedHashMap;
import java.util.Map;

class LRUCache<K, V> extends LinkedHashMap<K, V> {

    private final int capacity;
```

```
    public LRUCache(int capacity) {
        super(capacity, 0.75f, true);
        this.capacity = capacity;
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        return size() > capacity;
    }
}
```

8️⃣ Dry Run (VERY IMPORTANT)

Capacity = 3

```
put(1, A) → [1]
put(2, B) → [1, 2]
put(3, C) → [1, 2, 3]
```

Access:

```
get(1)
```

List becomes:

```
[2, 3, 1]
```

Add new element:

```
put(4, D)
```

- Size = 4 → exceeds capacity
- Eldest = **2** → removed

Final Cache:

```
[3, 1, 4]
```

9️⃣ Why LinkedHashMap is Better than Manual LRU

Manual LRU (Without LinkedHashMap)

Needs:

- HashMap
- Doubly Linked List
- Complex pointer management

LinkedHashMap LRU

- Built-in DLL
- Cleaner code
- Fewer bugs
- O(1) operations

🔟 Time & Space Complexity

Operation	Complexity
get()	O(1)
put()	O(1)
remove()	O(1)

Space:

- Slightly more than HashMap (due to DLL pointers)

1️⃣1️⃣ Thread Safety Consideration

`LinkedHashMap` is **NOT thread-safe**

Make it thread-safe:

```
Map<K, V> cache =
    Collections.synchronizedMap(new LRUCache<>(3));
```

Or use:

- `ConcurrentHashMap` + custom eviction logic (more complex)

1️⃣2️⃣ Real-World Use Cases

System	Usage
Web servers	Session cache
Microservices	Access token cache
DB layer	Query result cache
OS	Page replacement

1️⃣3️⃣ Interview Questions & Answers

Q: Why not HashMap for LRU?

❌ No order tracking

Q: Why accessOrder = true?

✔ Moves recently accessed elements to the end

Q: When is removeEldestEntry called?

✔ After `put()` operation

Q30. ConcurrentHashMap - where would you use it ? Able to explain internal implementation?

ConcurrentHashMap achieves thread safety and high concurrency by using lock-free reads, CAS operations, and fine-grained bucket-level locking instead of a global lock.

1️⃣ Why ConcurrentHashMap Exists

Problem with HashMap

- **Not thread-safe**
- Concurrent updates can cause:
  - Data corruption
  - Infinite loops during resize (pre-Java 8)
  - Lost updates

Problem with Collections.synchronizedMap()

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

- Uses **single global lock**
- Only **one thread** can read or write at a time
- Poor performance under high concurrency

2️⃣ What is ConcurrentHashMap?

`ConcurrentHashMap` is a **thread-safe, high-performance Map** designed for **concurrent access without global locking**.

Key Characteristics

✔ Thread-safe
✔ High concurrency
✔ No `ConcurrentModificationException`
✔ Locking only on **required portions**

3️⃣ Where Would You Use ConcurrentHashMap? (Use Cases)

🔹 1. Shared Cache in Multi-Threaded Applications ⭐⭐⭐

- Application-level caches
- Token / session cache
- Feature flags

```
ConcurrentHashMap<String, String> tokenCache;
```

🔹 2. High-Read / High-Write Systems

- Microservices
- Event processing systems
- Messaging systems

🔹 3. Maintaining Global Counters / Metrics

```
ConcurrentHashMap<String, AtomicInteger> metrics;
```

🔹 4. Avoiding Blocking Reads

- Reads do **NOT** block writes
- Writes do **NOT** block reads (mostly)

🔹 5. When Iteration Must Not Fail

- Iterators are **weakly consistent**
- No `ConcurrentModificationException`

4️⃣ Key Differences (Quick Interview Table)

Feature	HashMap	SynchronizedMap	ConcurrentHashMap
Thread-safe	❌	✔	✔
Global lock	❌	✔	❌
Read concurrency	❌	❌	✔
Performance	Fast	Slow	Fast
Null keys	✔	✔	❌

5️⃣ Internal Implementation (Very Important)

⚠️ Depends on Java Version

- **Java 7** → Segment-based locking
- **Java 8+** → Node + CAS + synchronized blocks

🔷 Java 7 Internal Implementation

Structure

```
ConcurrentHashMap
 └── Segment[] segments
      └── HashEntry[] table
```

🔹 Segment

- A **mini HashMap**
- Each segment has its own **lock**
- Default: **16 segments**

🔹 How Locking Worked

- Each segment handled a subset of keys
- Only one thread can modify a segment
- Other segments remain available

Example:

```
Thread 1 → Segment 3
Thread 2 → Segment 7
(No blocking)
```

✔ Better than global lock
❌ Still limited concurrency

🔷 Java 8+ Internal Implementation (Most Important)

🚀 Major Redesign (High Performance)

Data Structure

```
Node<K,V>[] table
```

Each bucket can be:

- Linked list
- Red-Black Tree (if many collisions)

6️⃣ Key Techniques Used in Java 8+

🔹 1. CAS (Compare-And-Swap)

- Lock-free atomic operations
- Used during:
  - Insertion
  - Size updates
  - Initialization

```
CAS(tab[i], null, new Node())
```

✔ No locking
✔ Very fast

🔹 2. Fine-Grained Locking

- Lock only the **bucket**
- Uses `synchronized` on **first node of bucket**

```
synchronized (f) {
   // modify bucket
}
```

✔ No global lock
✔ High concurrency

🔹 3. Reads Are Lock-Free

```
map.get(key);
```

- Uses volatile reads
- No locks
- Extremely fast

7️⃣ How put() Works (Java 8+)

Step-by-Step Flow

1️⃣ Compute hash
2️⃣ Locate bucket index

3️⃣ If bucket is empty → **CAS insert**
4️⃣ If bucket not empty → lock bucket head
5️⃣ Insert or update node
6️⃣ Convert list to tree if size > 8

Diagram

```
Bucket index

    ↓

[ Node ] → Node → Node
```

8️⃣ How get() Works

```
get(key)
```

- Reads bucket
- Traverses nodes
- No locking
- Uses volatile variables for visibility

9️⃣ Resize Operation (Important)

- Multiple threads help in resizing
- Uses **ForwardingNode**
- Old table → New table
- Threads cooperate

✔ No full blocking
✔ Safe migration

🔟 Why Null Is NOT Allowed

```
map.put(null, "A"); // ❌
```

Reason:

- Null return from `get()` would be ambiguous:
    - Key absent?
    - Key mapped to null?

1️⃣1️⃣ Iteration Behavior

- Iterators are **weakly consistent**
- Reflect some updates
- Never throw `ConcurrentModificationException`

1️⃣2️⃣ Performance Summary

Operation	Locking
get()	No lock
put()	Bucket-level lock
remove()	Bucket-level lock
resize()	Cooperative

1️⃣3️⃣ Real-World Example

Thread-Safe Cache

```
ConcurrentHashMap<String, String> cache = new ConcurrentHashMap<>();

cache.put("token", "abc123");
cache.get("token");
```

Q31. Synchronized HashMap vs Concurrent HashMap ?
Synchronized HashMap uses a single global lock, causing poor concurrency, while ConcurrentHashMap uses fine-grained locking and lock-free reads, providing much better performance in multi-threaded environments.

1️⃣ What Is a Synchronized HashMap?

You usually get it like this:

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

How it works

- Wraps a `HashMap`
- **Every method is synchronized**
- Uses **one global lock** (object-level lock)

Effect

- Only **one thread** can access the map at a time
- Read and write both block each other

2️⃣ What Is ConcurrentHashMap?

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

How it works (Java 8+)

- No global lock
- Uses:
    - **Lock-free reads**
    - **CAS (Compare-And-Swap)**
    - **Bucket-level locking** for writes

3️⃣ Core Difference (High Level)

Aspect	**Synchronized HashMap**	**ConcurrentHashMap**
Thread safety	✔	✔
Locking	Single global lock	Fine-grained locks
Read concurrency	❌	✔
Write concurrency	❌	✔
Performance	Poor under load	Excellent
Null key/value	✔	❌

Fail-fast iterator	✔	❌ (weakly consistent)

4️⃣ Locking Mechanism (Most Important Interview Topic)

🔒 Synchronized HashMap Locking

```
synchronized (map) {
    map.put(k, v);
}
```

- One lock for **all operations**
- Even `get()` is synchronized

Result

```
Thread-1 (get) → blocks Thread-2 (put)
```

⚙️ ConcurrentHashMap Locking

Reads (`get`)

- **No lock**
- Uses volatile variables

Writes (`put`)

- Locks **only the bucket**
- Other buckets remain accessible

```
Thread-1 → Bucket 3
Thread-2 → Bucket 7
(No blocking)
```

5️⃣ Internal Implementation Comparison

Synchronized HashMap

```
HashMap
   ↓
Synchronized Wrapper
   ↓
Single Lock
```

ConcurrentHashMap (Java 8+)

```
Node[] table
   ↓
Bucket-level synchronized blocks
   ↓
CAS operations
```

6️⃣ Iteration Behavior (Very Common Interview Trap)

Synchronized HashMap

```
Iterator<Entry<K,V>> it = map.entrySet().iterator();
```

- **Fail-fast**
- Throws `ConcurrentModificationException`
- Must manually synchronize iteration

```
synchronized(map) {
    for (...) { }
}
```

ConcurrentHashMap

- **Weakly consistent iterator**
- Does NOT throw exception
- Reflects some changes made during iteration

7️⃣ Performance Comparison (Real Life)

Threads	SynchronizedMap	ConcurrentHashMap
Low	OK	OK
Medium	Slow	Fast
High	Very Slow	Scales well

8️⃣ Null Handling (Interview Favorite)

Map Type	Null Key	Null Value
HashMap	✔	✔
Synchronized HashMap	✔	✔
ConcurrentHashMap	❌	❌

Why ConcurrentHashMap Disallows Null?

- Avoid ambiguity:

```
map.get(key) == null
```

→ key absent OR value is null?

9️⃣ When to Use Which?

✅ Use Synchronized HashMap When

- Very **low concurrency**
- Legacy code
- Simple use case
- Ordering is needed via `LinkedHashMap`

✅ Use ConcurrentHashMap When

- High-concurrency systems
- Multi-threaded caching
- Microservices
- Performance-critical applications

🔟 Code Example Comparison

Synchronized HashMap

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

map.put("A", "1");
```

ConcurrentHashMap

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

map.put("A", "1");
```

1️⃣1️⃣ Atomic Operations (Big Advantage)

ConcurrentHashMap

```
map.computeIfAbsent(key, k -> loadValue());
```

✔ Atomic
✔ Thread-safe

Synchronized HashMap

❌ Requires manual synchronization

Q32. What happens if we add duplicate key, value records into concurrent HashMap ?

The new value replaces the old value for the same key — atomically and thread-safely.

1️⃣ Fundamental Rule of Any Map (Including ConcurrentHashMap)

- **Keys are unique**
- **Values can be duplicated**
- Adding an entry with an **existing key** → **overwrites the old value**

This rule applies to:

- `HashMap`
- `SynchronizedMap`
- `ConcurrentHashMap`

2️⃣ Simple Example

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

map.put(1, "A");
map.put(1, "B");  // duplicate key
```

Result

```
Key: 1 → Value: "B"
```

✔ Old value `"A"` is replaced
✔ No exception
✔ Thread-safe

3️⃣ What Does put() Return?

```
String oldValue = map.put(1, "C");
```

- Returns **previous value**
- Returns `null` if key was not present

Example

```
map.put(1, "A");      // returns null
map.put(1, "B");      // returns "A"
```

4️⃣ Duplicate Value Scenario

```
map.put(1, "A");
map.put(2, "A"); // duplicate value, different key
```

✔ Allowed
✔ No issue

Maps enforce uniqueness on **keys**, not values.

5️⃣ What Happens Internally (Java 8+)

Case: Duplicate Key Insert

1️⃣ Hash is calculated
2️⃣ Bucket index is found
3️⃣ Bucket is locked (only that bucket)
4️⃣ Existing node with same key is found
5️⃣ **Value is replaced atomically**

No structural modification → no resizing.

6️⃣ Multi-Threaded Scenario (Very Important)

```
Thread-1: map.put(1, "A");
Thread-2: map.put(1, "B");
```

What happens?

- Only **one thread wins last**
- Final value = `"A"` or `"B"` (non-deterministic)
- **Map remains consistent**
- No corruption

✔ Thread-safe
✔ Atomic replacement

7️⃣ Does It Create Duplicate Entries Internally?

❌ **No**

- Only **one node per key**
- Value reference is replaced
- No duplicate nodes created

8️⃣ Special Case: putIfAbsent()

```
map.putIfAbsent(1, "A");
map.putIfAbsent(1, "B");
```

Result

```
Key: 1 → Value: "A"
```

✔ First insert succeeds
✔ Second insert ignored
✔ Atomic check + put

9️⃣ compute / merge Behavior (Advanced)

compute()

```
map.compute(1, (k, v) -> v + "X");
```

✔ Atomically updates value

---

merge()

```
map.merge(1, "B", (oldV, newV) -> oldV + newV);
```

✔ Useful for counters, accumulators

🔟 Interview Traps & Clarifications

Q: Will it throw an exception?

❌ No

Q: Will map size increase?

❌ No (size remains same)

Q: Is replacement thread-safe?

✔ Yes (bucket-level locking + CAS)

1️⃣1️⃣ Comparison with Synchronized HashMap

Behavior	SynchronizedMap	ConcurrentHashMap
Duplicate key	Replace value	Replace value
Thread safety	✔	✔
Performance	Poor	Excellent

Q33. Can we use the Employee object as Key of Hashmap. If so, how?

Yes, an Employee object can be used as a HashMap key if `equals()` and `hashCode()` are correctly overridden and the key fields are immutable.

Why HashMap Needs `equals()` and `hashCode()`

When you do:

```
map.put(employeeKey, value);
```

HashMap works in **two steps**:

1. **`hashCode()`** → decides **which bucket** the key goes into
2. **`equals()`** → checks **key equality** inside that bucket

If these are not overridden, HashMap uses `Object`'s implementation, which compares **memory addresses**, not logical equality.

❌ Wrong Way (Default Behavior)

```
class Employee {
    int id;
    String name;
}

Employee e1 = new Employee(1, "A");
Employee e2 = new Employee(1, "A");

map.put(e1, "Developer");
map.get(e2);   // ❌ returns null
```

Because:

- `e1.equals(e2)` → `false`

- Different hash codes → different buckets

✅ Correct Way (Interview-Expected)

Step 1: Make Employee Immutable (Best Practice)

```
final class Employee {
    private final int id;
    private final String name;

    public Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }
```

Step 2: Override `equals()`

```
   @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return
false;
        Employee emp = (Employee) o;
        return id == emp.id &&
               Objects.equals(name, emp.name);
    }
```

Step 3: Override `hashCode()`

```java
    @Override
    public int hashCode() {
        return Objects.hash(id, name);
    }
}
```

✅ Now HashMap Works Correctly

```java
Map<Employee, String> map = new HashMap<>();

Employee e1 = new Employee(101, "Bhanu");
Employee e2 = new Employee(101, "Bhanu");

map.put(e1, "Java Developer");

System.out.println(map.get(e2)); // ✔ Java Developer
```

🔑 Important Rules (Must Know)

✔ Equal objects → Same hashCode

```
e1.equals(e2) == true
⇒ e1.hashCode() == e2.hashCode()
```

✔ Key fields should be immutable

Changing key fields after insertion breaks the map.

```java
// Dangerous
emp.setId(200);
```

```java
map.get(emp); // ❌ may fail
```

Best Practice Summary

Rule	Why
Override `equals()`	Logical equality
Override `hashCode()`	Correct bucket
Use immutable fields	Prevent corruption
Use unique identifier	Better performance

Q34. print a hashMap in sorted order of keys?

To print a HashMap in sorted order of keys, convert it to a TreeMap or sort the key set and iterate over it.

✅ Approach 1: Using `TreeMap` (Most Common & Interview-Preferred)

`TreeMap` automatically keeps keys **sorted**.

```java
Map<Integer, String> map = new HashMap<>();
map.put(3, "C");
map.put(1, "A");
map.put(2, "B");

Map<Integer, String> sortedMap = new TreeMap<>(map);
```

```
sortedMap.forEach((k, v) ->
    System.out.println(k + " = " + v)
);
```

Output

```
1 = A
2 = B
3 = C
```

✔ Clean
✔ Efficient
✔ Interview-friendly

✅ Approach 2: Sort Keys Manually (More Control)

Useful when:

- You don't want another map
- Custom logic needed

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

List<Integer> keys = new ArrayList<>(map.keySet());
Collections.sort(keys);

for (Integer key : keys) {
    System.out.println(key + " = " + map.get(key));
}
```

✔ Flexible
❌ Slightly more code

✅ Approach 3: Java 8 Streams (Modern Style)

```
map.entrySet()
   .stream()
   .sorted(Map.Entry.comparingByKey())
   .forEach(e ->
       System.out.println(e.getKey() + " = " + e.getValue())
   );
```

✔ Concise
✔ Very readable
❌ Not ideal for huge maps (stream overhead)

🔑 If Keys Are Custom Objects (e.g., Employee)

Option A: Implement `Comparable`

```
class Employee implements Comparable<Employee> {
    int id;

    @Override
    public int compareTo(Employee e) {
        return Integer.compare(this.id, e.id);
    }
}

Map<Employee, String> sorted = new TreeMap<>(map);
```

Option B: Use Comparator

```
Map<Employee, String> sorted =
    new TreeMap<>(Comparator.comparing(Employee::getId));

sorted.putAll(map);
```

⏱️ Time Complexity Comparison

Method	Complexity
TreeMap	O(n log n)
Sorting keys	O(n log n)
Streams	O(n log n)

Q35. What are virtual threads(Java 21 / Project Loom)?
Virtual threads are lightweight, JVM-managed threads that allow millions of concurrent tasks using a simple blocking style, without the cost of platform threads.

Why Virtual Threads Were Introduced

Problems with Platform (OS) Threads

- **Heavyweight** (1–2 MB stack per thread)
- **Limited scalability** (thousands max)
- Blocking I/O **wastes OS threads**
- Complex async code (callbacks, futures, reactive)

What Virtual Threads Solve

- **Cheap to create** (KBs, not MBs)
- **Millions of threads possible**
- Blocking calls don't block OS threads
- Write **simple, synchronous code** that scales

Platform Thread vs Virtual Thread

Feature	Platform Thread	Virtual Thread
Managed by	OS	JVM
Stack size	Large (MBs)	Small (grows dynamically)
Count	Thousands	Millions
Blocking I/O	Blocks OS thread	Unmounts from OS thread
Creation cost	High	Very low

How Virtual Threads Work (Internals – Simple)

- Virtual threads are scheduled by the **JVM**, not the OS

- They run on a small pool of **carrier (platform) threads**
- On blocking I/O:
    - Virtual thread **parks**
    - Carrier thread is **freed**
    - Another virtual thread runs

```
Virtual Threads (many)
        ↓
Carrier Threads (few)
        ↓
Operating System
```

Creating Virtual Threads

1️⃣ Single Virtual Thread

```
Thread.startVirtualThread(() -> {
    System.out.println("Hello from virtual thread");
});
```

2️⃣ Executor with Virtual Threads (Recommended)

```
ExecutorService executor =
        Executors.newVirtualThreadPerTaskExecutor();

executor.submit(() -> {
    // blocking I/O is OK
});
```

3️⃣ Using Thread.Builder

```
Thread vt = Thread.ofVirtual().start(() -> {
    System.out.println("Running");
});
```

Blocking Is OK with Virtual Threads 😎

```
Thread.sleep(1000);      // fine
socket.read();           // fine
dbQuery.execute();       // fine
```

👉 Blocking does **NOT** block OS threads.

Real-World Use Cases ⭐⭐⭐

✔ Web servers (Tomcat, Jetty, Netty)

- One virtual thread per request
- Handle **millions of connections**

✔ Microservices

- REST calls
- DB access
- External APIs

✔ **Batch & background jobs**

- Massive parallelism
- Simple code

What Virtual Threads Are NOT Good For ❌

Scenario	**Why**
CPU-bound tasks	Too many threads cause context switching
Tight loops	No I/O to park
Long synchronized blocks	Causes pinning

Important Limitation: Thread Pinning 🚨

Problem

If a virtual thread:

- Enters a `synchronized` block
- Performs blocking I/O inside it

➡️ The **carrier thread gets pinned**

Solution

- Prefer `ReentrantLock`

- Avoid blocking inside `synchronized`

Virtual Threads vs CompletableFuture vs Reactive

Feature	Virtual Threads	CompletableFuture	Reactive
Code style	Simple	Async	Async
Learning curve	Low	Medium	High
Scalability	High	High	Very High
Debugging	Easy	Hard	Hard

Are Virtual Threads Thread-Safe?

- Virtual threads behave like normal threads
- **All synchronization rules still apply**
- `synchronized`, locks, ThreadLocal work

Performance Summary

Metric	Result
Thread creation	Extremely fast

Memory usage	Very low
Blocking I/O	Scales
Throughput	Excellent

Q36. Java 8 program to find sum of even numbers from a list of integers.

✔️ **Solution 1: Use Identity Value (Best & Safest)**

```
int result = list.stream()
                 .filter(e -> e % 2 == 0)
                 .reduce(0, (a, b) -> a + b);
```

✔️ **Solution 2: Best Modern Approach (Recommended)**

```
int result = list.stream()
                 .filter(e -> e % 2 == 0)
                 .mapToInt(Integer::intValue)
                 .sum();
```

Q37. Asked to write service layer code using completable future.

CompletableFuture enables asynchronous, non-blocking execution of service calls, allowing multiple independent operations to run in parallel and be combined efficiently.

1️⃣ DTOs (Simple Models)

```
class User {
    int id;
    String name;
}

class Order {
    int id;
}
```

```java
class Payment {
    int id;
}

class UserResponse {
    User user;
    List<Order> orders;
    List<Payment> payments;
}
```

2️⃣ Repository / Client Layer (Blocking Calls)

```java
class UserRepository {
    User getUser(int userId) {
        sleep(1000);
        return new User();
    }
}

class OrderRepository {
    List<Order> getOrders(int userId) {
        sleep(1000);
        return List.of(new Order());
    }
}

class PaymentRepository {
    List<Payment> getPayments(int userId) {
        sleep(1000);
```

```
        return List.of(new Payment());
    }

    private void sleep(long ms) {
        try { Thread.sleep(ms); } catch (InterruptedException e)
{}
    }
}
```

3️⃣ Service Layer Using CompletableFuture ⭐⭐⭐

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

class UserService {

    private final Executor executor =
Executors.newFixedThreadPool(3);

    private final UserRepository userRepo = new
UserRepository();
    private final OrderRepository orderRepo = new
OrderRepository();
    private final PaymentRepository paymentRepo = new
PaymentRepository();

    public UserResponse getUserDetails(int userId) {

        CompletableFuture<User> userFuture =
                CompletableFuture.supplyAsync(() ->
userRepo.getUser(userId), executor);
```

```
        CompletableFuture<List<Order>> orderFuture =
                CompletableFuture.supplyAsync(() ->
orderRepo.getOrders(userId), executor);

        CompletableFuture<List<Payment>> paymentFuture =
                CompletableFuture.supplyAsync(() ->
paymentRepo.getPayments(userId), executor);

        CompletableFuture<UserResponse> responseFuture =
                CompletableFuture.allOf(userFuture, orderFuture,
paymentFuture)
                        .thenApply(v -> {
                            UserResponse response = new
UserResponse();
                            response.user = userFuture.join();
                            response.orders =
orderFuture.join();
                            response.payments =
paymentFuture.join();
                            return response;
                        });

        return responseFuture.join(); // block only at boundary
    }
}
```

4️⃣ Why `allOf()` Is Used

- Runs all tasks **in parallel**
- Waits for **all futures to complete**
- Combines results

5️⃣ Exception Handling (Important for Interview)

```
CompletableFuture<User> userFuture =
    CompletableFuture.supplyAsync(() ->
userRepo.getUser(userId))
        .exceptionally(ex -> {
            log.error("User service failed", ex);
            return null;
        });
```

Or globally:

```
responseFuture.exceptionally(ex -> {
    throw new RuntimeException("Failed to fetch data", ex);
});
```

6️⃣ Using `thenCompose()` (Dependent Calls)

```
CompletableFuture<UserProfile> profileFuture =
    CompletableFuture.supplyAsync(() -> getUser(userId))
        .thenCompose(user ->
            CompletableFuture.supplyAsync(() ->
getProfile(user.id))
        );
```

✔ Used when second call depends on first result

7️⃣ Using `thenCombine()` (Combine Two Futures)

```
CompletableFuture<UserOrderInfo> future =
    userFuture.thenCombine(orderFuture,
        (user, orders) -> new UserOrderInfo(user, orders)
    );
```

8️⃣ Best Practices (Interview Gold ⭐)

✔ Use **custom Executor** (avoid common ForkJoinPool)
✔ Block **only at controller boundary**
✔ Use `join()` instead of `get()`
✔ Handle exceptions explicitly
✔ Avoid nested futures

9️⃣ When NOT to Use CompletableFuture ❌

- CPU-bound tasks → use ExecutorService
- Very simple sequential logic
- Reactive systems (use WebFlux)

Q38. Write code using streams to print the employee who works in Product Team and Is located in bangalore

```
employees.stream()
         .filter(e -> "Product".equalsIgnoreCase(e.getTeam())
                   && "Bangalore".equalsIgnoreCase(e.getLocation()))
         .forEach(System.out::println);
```

Q39. What is deadlock, how to avoid it?

Deadlock occurs when multiple threads wait indefinitely for resources held by each other, creating a circular dependency.

Classic Deadlock Example

```
class DeadlockExample {
    private static final Object lock1 = new Object();
    private static final Object lock2 = new Object();

    public static void main(String[] args) {

        Thread t1 = new Thread(() -> {
            synchronized (lock1) {
                System.out.println("Thread 1 acquired lock1");
                try { Thread.sleep(100); } catch (Exception e)
{}
                synchronized (lock2) {
                    System.out.println("Thread 1 acquired
lock2");
                }
            }
        });
```

```
        Thread t2 = new Thread(() -> {
            synchronized (lock2) {
                System.out.println("Thread 2 acquired lock2");
                try { Thread.sleep(100); } catch (Exception e)
{}
                synchronized (lock1) {
                    System.out.println("Thread 2 acquired
lock1");
                }
            }
        });

        t1.start();
        t2.start();
    }
}
```

❌ Result

- Thread-1 holds `lock1`, waiting for `lock2`
- Thread-2 holds `lock2`, waiting for `lock1`
- **Deadlock happens**

Four Necessary Conditions for Deadlock (Must Know)

Deadlock occurs **only if all four conditions are true**:

Condition	Meaning

Mutual Exclusion	Resource held by only one thread
Hold and Wait	Thread holds one resource and waits for another
No Preemption	Resource can't be forcibly taken
Circular Wait	Thread-1 → Thread-2 → Thread-1

How to Avoid Deadlock (Important Part)

✅ 1. Avoid Circular Lock Ordering (Best Solution) ⭐⭐⭐

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

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

✔ Breaks circular wait

✅ 2. Use `tryLock()` with Timeout

```
if (lock1.tryLock(1, TimeUnit.SECONDS)) {
    try {
        if (lock2.tryLock(1, TimeUnit.SECONDS)) {
            try {
                // work
            } finally {
                lock2.unlock();
```

```
            }
        }
    } finally {
        lock1.unlock();
    }
}
```

✔ Prevents infinite waiting
✔ Allows recovery

✅ 3. Minimize Synchronized Blocks

```
// Bad
synchronized(this) {
    heavyLogic();
}

// Good
prepareData();
synchronized(this) {
    updateSharedData();
}
```

✔ Reduces lock holding time

✅ 4. Avoid Nested Locks

```
// Risky
synchronized(lock1) {
    synchronized(lock2) {
```

```
    }
}
```

✔ Use single lock if possible

✅ 5. Use High-Level Concurrency Utilities

Prefer:

- `ExecutorService`
- `ConcurrentHashMap`
- `Semaphore`
- `BlockingQueue`

These are **deadlock-safe by design**.

Q40. finally and finalize the difference?

`finally` is a block used for guaranteed cleanup after exception handling, whereas `finalize()` is a GC-invoked method for object cleanup that is unpredictable and deprecated.

1️⃣ `finally` (Keyword)

What is it?

- A **block** used with `try-catch`
- Executes **always**, whether an exception occurs or not

Purpose

- To **clean up resources**
    - Close files
    - Close DB connections
    - Release locks

Example

```
try {
    int a = 10 / 0;   // Exception
} catch (ArithmeticException e) {
    System.out.println("Exception caught");
} finally {
    System.out.println("Finally block executed");
}
```

Output

```
Exception caught
Finally block executed
```

✔ Runs even if exception occurs
✔ Runs even if exception is not caught

When `finally` does NOT execute

- `System.exit(0)`
- JVM crash

2️⃣ `finalize()` (Method)

What is it?

- A **method** of `java.lang.Object`
- Called by **Garbage Collector (GC)** before destroying an object

```
protected void finalize() throws Throwable
```

Purpose

- Cleanup before object removal (old approach)

Example

```
class Test {
    @Override
    protected void finalize() throws Throwable {
        System.out.println("Finalize called");
    }
}

Test t = new Test();
t = null;
System.gc();
```

⚠️ Output is **NOT guaranteed**

Important Notes About `finalize()`

❌ Not predictable
❌ No guarantee it will be called
❌ Performance issues
❌ Deprecated in Java 9

3️⃣ Key Differences (Interview Table)

Aspect	`finally`	`finalize()`
Type	Block	Method
Used with	try–catch	Garbage Collection
Execution	Always (mostly)	Not guaranteed
Called by	JVM immediately	GC
Purpose	Resource cleanup	Object cleanup
Frequency	Each try block	Once per object
Status	Actively used	Deprecated

4️⃣ Why `finalize()` Is Deprecated?

- Unpredictable execution
- GC delays

- Can cause memory leaks
- Blocks GC thread

5️⃣ Modern Alternative to `finalize()`

✔ `try-with-resources`

```
try (FileInputStream fis = new FileInputStream("file.txt")) {
    // use file
}
```

✔ Safe
✔ Deterministic
✔ Recommended

Q41. garbage collector method, can we call it garbage collector?

What Is the Garbage Collector (GC)?

The **Garbage Collector** is a JVM component that:

- **Automatically frees memory**
- Removes objects that are **no longer reachable**
- Prevents memory leaks

Is There a Garbage Collector Method?

❌ No direct "garbage collector method" exists

But Java provides **ways to REQUEST GC**, not force it.

Can We Call the Garbage Collector?

✔ YES — but only as a request, not a guarantee

1️⃣ Using `System.gc()`

```
System.gc();
```

- Requests JVM to run GC
- JVM **may ignore** the request

2️⃣ Using `Runtime.getRuntime().gc()`

```
Runtime.getRuntime().gc();
```

- Same as `System.gc()`
- Only a suggestion

⚠️ Important Interview Point

> **Garbage Collection is controlled by the JVM, not the programmer.**

Calling GC:

- ❌ Does NOT guarantee immediate execution
- ❌ Does NOT guarantee object destruction

- ❌ Should NOT be relied upon

Why We Should NOT Manually Call GC

- Performance overhead
- JVM knows better when to run GC
- Modern GCs are optimized
- Can pause application threads (STW)

What Happens When GC Runs?

1. Marks reachable objects
2. Removes unreachable objects
3. Reclaims memory
4. May compact heap

What About `finalize()`?

```
protected void finalize() throws Throwable
```

- Called by GC **before object removal**
- ❌ Not guaranteed

- ❌ Deprecated since Java 9

Best Practices Instead of Calling GC

✔ Use **try-with-resources**
✔ Close resources explicitly
✔ Avoid memory leaks
✔ Use proper object scope

Interview Q&A

Q: Can we force GC?

❌ No

Q: Does `System.gc()` always run GC?

❌ No

Q: Who controls GC?

✔ JVM

Q: Should we call GC in production?

❌ No