

## LIST OF ALL IMPORTANT JAVA SPRINGBOOT QUESTIONS ASKED IN SERVICE BASED

### 1. Difference Between Aggregation and Composition

Aspect	Aggregation	Composition
Relationship	Weak "has-a"	Strong "has-a"
Dependency	Child can exist independently	Child cannot exist without parent
Lifetime	Independent lifetime	Bound to parent object's lifetime
Example	Use University and Students	House and Rooms

#### ► Aggregation Example:

```
java
CopyEdit
class Student {
    String name;
}

class Department {
    List<Student> students; // Aggregation: students can exist outside department
}
```

- Student can exist without Department.

#### ► Composition Example:

```
java
CopyEdit
class Room {
    private int number;
}

class House {
```

```
private Room room = new Room(); // Composition: Room is part of House  
}
```

- Room cannot meaningfully exist without House.
- 

◆ **2. Have You Used Caching in Your Projects? Explain.**

Yes, caching improves performance by storing frequent results in memory. I've used:

► **a. Spring Cache Abstraction:**

- Easy annotation-based caching using @Cacheable, @CacheEvict, etc.
- Backed by providers like **EhCache**, **Caffeine**, or **Redis**.

java

CopyEdit

```
@Cacheable("users")  
public User getUserById(Long id) {  
    return userRepository.findById(id).orElse(null);  
}
```

- On first call, result is cached.
- Later calls fetch from cache unless evicted.

► **b. Redis (Distributed Caching):**

- Useful in microservices for shared caching.
- Works well with @Cacheable using RedisCacheManager.

yaml

CopyEdit

```
spring.cache.type=redis  
spring.redis.host=localhost
```

► **c. Manual Caching using Map (Rare):**

- For specific cases in memory-heavy processing.

java

CopyEdit

```
Map<String, Product> cache = new HashMap<>();  
  
if (!cache.containsKey(id)) {  
  
    cache.put(id, fetchProduct(id));  
  
}  
  
return cache.get(id);
```

---

### ◆ 3. What is WeakHashMap?

**WeakHashMap<K, V>** is a map implementation that uses **weak references** for keys, meaning:

- Keys are eligible for **garbage collection** if there are **no strong references** to them.
- Entries may be automatically removed during GC.

### ► Why It's Used?

- Useful for memory-sensitive caches.
- Avoids memory leaks by cleaning unused keys.

### ► Code Example:

```
java  
  
CopyEdit  
  
Map<Object, String> map = new WeakHashMap<>();  
  
Object key = new Object();  
  
map.put(key, "Some Value");  
  
  
System.out.println("Before GC: " + map);  
  
  
key = null; // Remove strong reference  
  
System.gc(); // Suggest garbage collection  
  
  
Thread.sleep(1000); // Give time for GC
```

```
System.out.println("After GC: " + map); // Entry may be gone
```

► **Output (may vary):**

pgsql

CopyEdit

Before GC: {java.lang.Object@1a2b3c=Some Value}

After GC: {} // Entry removed after GC

► **Comparison:**

Map Type	Key Reference	Auto Removal	Use Case
HashMap	Strong	✗	General purpose
WeakHashMap	Weak	✓	Memory-sensitive

---

Let me know if you'd like me to update the PDF with these enhanced answers or continue with other topics.

**You said:**

what is n+1 query problem, and then how to tackle it

**ChatGPT said:**

◆ **What is the N+1 Query Problem?**

The **N+1 query problem** is a common performance issue in ORM frameworks like **Hibernate** or **JPA**, where:

- One query fetches the **main entity** (the "1").
- Then **N additional queries** are executed to fetch the related child entities for each result row (the "N").

► **Example Scenario:**

Suppose you have:

java

CopyEdit

```
class Department {
```

```
@OneToMany(mappedBy = "department")
private List<Employee> employees;
}
```

And you run:

java

CopyEdit

```
List<Department> departments = entityManager.createQuery("from Department",
Department.class).getResultList();
```

Now, for **each department**, if employees is lazily loaded:

java

CopyEdit

```
for (Department dept : departments) {
    System.out.println(dept.getEmployees().size()); // triggers 1 query per dept
}
```

This results in:

- 1 query for departments.
- N queries for N departments' employees.

➡ Total queries =  $N + 1$

---

#### ◆ Why It's a Problem

- **Performance degradation.**
  - Causes **database load** and **latency** spikes.
  - Especially problematic in large data sets.
- 

#### ◆ How to Fix or Avoid N+1 Problem

##### ✓ 1. Use JOIN FETCH (JPQL)

java

CopyEdit

```
List<Department> departments = entityManager.createQuery(  
    "SELECT d FROM Department d JOIN FETCH d.employees", Department.class  
).getResultList();
```

- Fetches departments and employees in **one query** using SQL JOIN.

## 2. @EntityGraph in Spring Data JPA

java

CopyEdit

```
@EntityGraph(attributePaths = {"employees"})  
List<Department> findAll(); // Loads both Department and Employees
```

- 

### 1. What is an index? Why is it used?

- Speeds up data retrieval.
- Similar to an index in a book.

### 2. What are the types of indexes in SQL?

- **Primary Index** (on primary key)
- **Unique Index**
- **Clustered Index**
- **Non-Clustered Index**
- **Composite Index** (on multiple columns)
- **Full-Text Index** (for searching text)
- **Bitmap Index** (used in data warehouses)

### 3. What is the difference between Clustered and Non-Clustered Indexes?

#### **Clustered Index      Non-Clustered Index**

Sorts actual data rows    Has separate structure with pointer

Only one per table      Can have multiple per table

Clustered Index	Non-Clustered Index
-----------------	---------------------

Faster for range queries	Faster for point queries
--------------------------	--------------------------

#### 4. Can a table have multiple indexes?

- Yes, a table can have **multiple non-clustered indexes**.
- Only **one clustered index** per table.

#### 5. What are the drawbacks of indexes?

- Slower INSERT, UPDATE, DELETE due to index maintenance.
- Additional disk space required.

#### 6. How does indexing affect query performance?

- Speeds up SELECT queries.
- Can slow down write-heavy operations.

#### 7. What is a Composite Index? When would you use it?

- Index on multiple columns.
- Useful when queries often filter/sort by a combination of those columns.

#### 8. What is a covering index?

- An index that contains all the columns needed by a query.
- Helps in **Index-Only Scan** (query doesn't go to the table).

#### 9. How do you decide which columns to index?

- Columns used in WHERE, JOIN, ORDER BY, GROUP BY.

#### 10. Can indexes be used on views?

- Yes, through **Indexed Views** or **Materialized Views**, depending on the DBMS.

---

#### Bonus: SQL Query Performance Tuning Question

*How would you optimize a slow query?*

- Use EXPLAIN or QUERY PLAN to analyze.
- Add indexes where necessary.
- Avoid using functions on indexed columns in WHERE.

- Reduce table scans.

## Components of JVM Architecture

### 1. Class Loader Subsystem

- **Role:** Loads .class files (compiled bytecode) into JVM memory.
- **Phases:**
  - **Loading:** Finds and loads class files.
  - **Linking:** Verifies and prepares classes.
  - **Initialization:** Initializes static variables and blocks.

 *It ensures classes are loaded only once using the parent delegation model.*

---

### 2. Runtime Data Areas (Memory Areas)

These are different memory segments created by JVM when a program runs:

#### a. Method Area (shared)

- Stores metadata like class names, method names, static variables, and constant pool.

#### b. Heap (shared)

- Stores objects and arrays.
- **Garbage Collector** works here.

#### c. Java Stack (per thread)

- Stores stack frames (one for each method call).
- Each frame contains:
  - **Local variables**
  - **Operand stack**
  - **Return address**

#### d. Program Counter (PC) Register (per thread)

- Keeps the address of the currently executing bytecode instruction.

#### e. Native Method Stack

- Used for native (non-Java) methods written in C/C++.
  - Works with JNI (Java Native Interface).
- 

### 3. Execution Engine

- Executes the bytecode using the following:
    - ◆ **a. Interpreter**
      - Interprets bytecode line by line.
      - Slower due to repetitive interpretation.
    - ◆ **b. JIT (Just-In-Time) Compiler**
      - Converts frequently used bytecode into **native machine code** for faster execution.
- Boosts performance by caching compiled native code.
- 

### 4. Native Interface (JNI)

- Allows Java code to interact with native applications/libraries (e.g., C, C++ code).
- 

### 5. Native Method Libraries

- Set of platform-specific libraries required for native method execution.
- 

#### Summary Diagram (Text Version)

sql

CopyEdit

JVM

|

-----  
Class Loader Subsystem

-----  
Runtime Data Areas



## 5. What is Idempotency?

- **Idempotent** operation = **same result no matter how many times it is called**.
- Common in **REST APIs** and **distributed systems**.

### Example in REST:

#### HTTP Method Idempotent? Reason

GET	✓	Reads data, no side effects
PUT	✓	Same data replaces existing data
DELETE	✓	Deleting same item again has no effect
POST	✗	Creates <b>new</b> resource on each call

### Real-life Example:

- Turning a light **ON**:
  - If it's already on, doing it again doesn't change anything → **idempotent**.

Let me know if you'd like a summarized PDF or if I should explain any of these with **code examples**.

2/2

what were the changes that came in internal implementation of the hashmap after java8

**ChatGPT said:**

In **Java 8**, significant **internal changes** were made to the **HashMap implementation** to improve its **performance in cases of high hash collisions**. Below are the **key changes** with explanations and examples:

---

◆ **1. Introduction of Tree-based Buckets**

✓ **Before Java 8:**

- All hash collisions were handled using **linked lists**.
- Lookup time for a bucket with many entries: **O(n)**.

✓ **After Java 8:**

- If the number of entries in a bucket **exceeds a threshold** (i.e., **8 elements**), and the total HashMap capacity is at least **64**, the **linked list is converted into a Red-Black Tree**.
- Tree-based lookup time: **O(log n)**.

📌 **Thresholds:**

Condition	Value
-----------	-------

Treeify Threshold	8
-------------------	---

Untreeify Threshold	6
---------------------	---

Minimum Capacity for Treeify 64

牢固树立 **Why Red-Black Tree?**

- Balanced tree → guarantees **logarithmic time** for search/insert/delete.
- Good performance under **poor hashCode() implementations**.

🧠 **Example Scenario:**

java

CopyEdit

```
Map<Integer, String> map = new HashMap<>();  
  
for (int i = 0; i < 100; i++) {  
  
    map.put(i % 10, "Value"); // Causes hash collisions (same bucket)  
  
}
```

In Java 8, once collisions in a bucket exceed 8 entries, it switches from a **linked list to a tree**, improving lookup performance.

---

## ◆ 2. Optimized Hash Function

- In Java 8, a **better hash spreading function** is used to avoid collisions:

java

CopyEdit

```
static final int hash(Object key) {  
  
    int h;  
  
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
  
}
```

- This XOR with high-order bits ensures a **better distribution of keys**, reducing collisions.
- 

## ◆ 3. Dynamic Treeification and Detreeification

- If entries in a bucket reduce below **6**, the tree is **converted back into a linked list** (called **untreeify**).
  - Prevents **tree overhead** when not needed.
- 

## ◆ 4. Concurrent Improvements (used in ConcurrentHashMap too)

- Though not directly part of HashMap, similar optimizations in **ConcurrentHashMap** include:
  - Lock splitting
  - TreeBin for heavily contended buckets

---

## Summary Table

Feature	Pre Java 8	Java 8+
Collision Handling	Linked List	Linked List → Red-Black Tree
Lookup Time (on collision)	$O(n)$	$O(\log n)$
Treeify Threshold	N/A	8 entries (bucket), capacity $\geq 64$
Untreeify Threshold	N/A	< 6 entries
Hash Spreading		Simple hashCode hashCode XOR shifted bits

---

Let me know if you'd like to see a [diagram](#) or [PDF version](#) of this explanation.

name some functional interfaces and their one abstract method what is the diff b/w content application bean factory and which one is lazy and which one is eager

**ChatGPT said:**

Sure! Here's a detailed answer for both your questions:

---

## 1. Functional Interfaces and Their Abstract Methods

In Java, a **Functional Interface** is an interface that has exactly **one abstract method**. These are commonly used in **lambda expressions**, **method references**, and **streams API**.

Below are some **commonly used functional interfaces** from `java.util.function` and others:

Functional Interface	Abstract Method	Description
Runnable	void run()	Used to execute code on a thread
Callable<V>	V call()	Like Runnable, but returns a result
Supplier<T>	T get()	Supplies a result
Consumer<T>	void accept(T t)	Consumes input and returns nothing

<b>Functional Interface Abstract Method</b>		<b>Description</b>
Function<T, R>	R apply(T t)	Transforms input T to output R
Predicate<T>	boolean test(T t)	Tests input and returns true/false
BiFunction<T, U, R>	R apply(T t, U u)	Takes 2 inputs and returns a result
BiConsumer<T, U>	void accept(T t, U u)	Takes 2 inputs, returns nothing
UnaryOperator<T>	T apply(T t)	Special case of Function (input = output)
BinaryOperator<T>	T apply(T t1, T t2)	Operates on 2 values of same type

 **Custom Functional Interface Example:**

```
java
```

```
CopyEdit
```

```
@FunctionalInterface
```

```
interface MyOperation {
```

```
    int operate(int a, int b);
```

```
}
```

```
MyOperation add = (a, b) -> a + b;
```

```
System.out.println(add.operate(2, 3)); // Output: 5
```

---

 **2. BeanFactory vs ApplicationContext**

Both BeanFactory and ApplicationContext are Spring **IoC containers**, responsible for **managing beans**, but there are key differences:

<b>Feature</b>	<b>BeanFactory</b>	<b>ApplicationContext</b>
<b>Type</b>	Basic Container	Advanced Container
<b>Bean Loading</b>	Lazy (loaded on demand)	Eager (loads all singleton beans at startup)
<b>Event Handling</b>	Not supported	Supported (ApplicationEventPublisher)

Feature	BeanFactory	ApplicationContext
<b>AOP Support</b>	Limited / Manual	Fully supported
<b>Internationalization (i18n)</b>	Not available	Available
<b>Bean PostProcessors</b>	Must be registered manually	Auto-detected
<b>Use Case</b>	Lightweight applications	Most Spring applications

 **Example Usage:**

```
java
CopyEdit
// ApplicationContext (commonly used)
ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
MyBean obj = context.getBean(MyBean.class);

// BeanFactory (used in resource-constrained scenarios)
BeanFactory factory = new XmlBeanFactory(new ClassPathResource("beans.xml"));
MyBean obj2 = factory.getBean(MyBean.class);
```

 **Which one is lazy or eager?**

- BeanFactory: **Lazy Initialization** (Beans are created when requested)
- ApplicationContext: **Eager Initialization** (Singleton beans created at startup)
- **Cascading** in JPA means that an operation (like persist, merge, remove) applied to one entity is **automatically propagated** to its related entities.
-  **Why Use It?**
- To **avoid repetitive operations** on child entities when working with relationships like @OneToOne, @OneToMany, @ManyToOne, or @ManyToMany.
-  **Cascade Types in JPA:**

- **Cascade Type**
- CascadeType.PERSIST
- CascadeType.MERGE
- CascadeType.REMOVE
- CascadeType.REFRESH
- CascadeType.DETACH
- CascadeType.ALL
- **Description**
- When you persist parent, child is persisted too.
- When you merge parent, child is merged.
- When you delete parent, child is also deleted.
- When you refresh parent, child is also refreshed from DB.
- When you detach parent, child is also detached.
- Applies all of the above cascade types.

- **◆ Example:**

- java
  - CopyEdit
  - @Entity
  - public class Employee {
  - @OneToOne(cascade = CascadeType.ALL)
  - private Address address;
  - }
  - Now when you save the Employee, its Address will also be saved automatically:
  - java
  - CopyEdit
  - Employee emp = new Employee();
  - emp.setAddress(new Address(...));
  - entityManager.persist(emp); // This also persists Address
- 

- **2. Difference Between Space Complexity and Time Complexity**

- | • <b>Aspect</b>     | • <b>Time Complexity</b>                      | • <b>Space Complexity</b>                      |
|---------------------|---|--|
| • <b>Definition</b> | • Amount of <b>time</b> taken by an algorithm | • Amount of <b>memory</b> used by an algorithm |

- **Aspect**
- **Focus**
- **Goal**
- **Unit**
- **Example**
- **Time Complexity**
- Execution steps or operations
- Optimize speed
- Big O notation (e.g.,  $O(n)$ ,  $O(\log n)$ )
- Searching in array →  $O(n)$  time
- **Space Complexity**
- Memory consumption (variables, call stack, etc.)
- Optimize memory usage
- Big O notation (e.g.,  $O(1)$ ,  $O(n)$ )
- Storing input in array →  $O(n)$  space

- **◆ Example:**

- Consider this code snippet:

- java
- CopyEdit
- int sum(int[] arr) {
- int total = 0;                 // O(1) space
- for (int i = 0; i < arr.length; i++) {
- total += arr[i];            // O(n) time
- }
- return total;
- }

- **Time Complexity:**  $O(n)$  — loop runs  $n$  times

- **Space Complexity:**  $O(1)$  — uses one variable (total)

- But if you stored a new array of prefix sums, space complexity would become  $O(n)$ .

- **Cascading** in JPA means that an operation (like persist, merge, remove) applied to one entity is **automatically propagated** to its related entities.

- **◆ Why Use It?**

- To **avoid repetitive operations** on child entities when working with relationships like @OneToOne, @OneToMany, @ManyToOne, or @ManyToMany.

- **◆ Cascade Types in JPA:**

- **Cascade Type**
- CascadeType.PERSIST
- CascadeType.MERGE
- CascadeType.REMOVE
- CascadeType.REFRESH
- CascadeType.DETACH
- CascadeType.ALL
- **Description**
- When you persist parent, child is persisted too.
- When you merge parent, child is merged.
- When you delete parent, child is also deleted.
- When you refresh parent, child is also refreshed from DB.
- When you detach parent, child is also detached.
- Applies all of the above cascade types.

- **◆ Example:**

- java
  - CopyEdit
  - @Entity
  - public class Employee {
  - @OneToOne(cascade = CascadeType.ALL)
  - private Address address;
  - }
  - Now when you save the Employee, its Address will also be saved automatically:
  - java
  - CopyEdit
  - Employee emp = new Employee();
  - emp.setAddress(new Address(...));
  - entityManager.persist(emp); // This also persists Address
- 

- **2. Difference Between Space Complexity and Time Complexity**

- | • <b>Aspect</b>     | • <b>Time Complexity</b>                      | • <b>Space Complexity</b>                      |
|---------------------|---|--|
| • <b>Definition</b> | • Amount of <b>time</b> taken by an algorithm | • Amount of <b>memory</b> used by an algorithm |

- **Aspect**
  - **Time Complexity**
  - Execution steps or operations
  - Optimize speed
  - Big O notation (e.g.,  $O(n)$ ,  $O(\log n)$ )
  - Searching in array →  $O(n)$  time
- **Focus**
  - Memory consumption (variables, call stack, etc.)
- **Goal**
  - Optimize memory usage
- **Unit**
  - Big O notation (e.g.,  $O(1)$ ,  $O(n)$ )
- **Example**
  - Storing input in array →  $O(n)$  space
- **◆ Example:**
  - Consider this code snippet:
  - java
  - CopyEdit
  - ```
int sum(int[] arr) {  
    int total = 0;           // O(1) space  
    for (int i = 0; i < arr.length; i++) {  
        total += arr[i];     // O(n) time  
    }  
    return total;  
}
```
  - **Time Complexity:**  $O(n)$  — loop runs  $n$  times
  - **Space Complexity:**  $O(1)$  — uses one variable (total)
  - But if you stored a new array of prefix sums, space complexity would become  $O(n)$ .
  - Absolutely, let's go deep into the **difference between Lock and Monitor** in Java. Many developers confuse these two because **monitors** are implicitly used with the synchronized keyword, while **locks** are explicitly managed through classes like ReentrantLock.
  - ---
  -  **MONITOR vs LOCK in Java (Detailed Explanation)**

- \_\_\_\_\_
- **What is a Monitor?**
- A **monitor** is a synchronization mechanism that allows only **one thread at a time** to execute a block of code protected by the monitor. In Java:
- Every **object** has an **intrinsic monitor** (a.k.a. intrinsic lock).
- When a thread enters a synchronized block or method, it acquires the object's monitor.
- Only **one thread can hold the monitor** at a time; others are blocked until it's released.
- **Example of Monitor via synchronized:**
  - java
  - CopyEdit
  - public class SharedResource {
  - public synchronized void printData() {
  - // Thread must acquire the monitor of 'this' to enter
  - System.out.println("Inside synchronized method");
  - }
  - }
  - public void anotherWay() {
  - synchronized (this) {
  - // synchronized block also uses monitor
  - System.out.println("Inside synchronized block");
  - }
  - }
  - }
- **Key Characteristics of Monitor:**

| • Feature          | • Description                                |
|--------------------|----------------------------------------------|
| • Built-in locking | • Associated with every object automatically |

- **Feature**
  - Automatic acquire/release
  - No timeout
  - Not interruptible
  - No fairness
  - Description
  - Acquired when entering synchronized, released when exiting
  - You cannot wait with a timeout on a monitor
  - A thread blocked on a monitor cannot be interrupted
  - No way to ensure first-come-first-serve thread access
- 

- **What is a Lock?**

- Java introduced the `java.util.concurrent.locks` package in **Java 5** with more powerful and flexible locking constructs, especially **Lock** interface and **ReentrantLock** implementation.

- With a **Lock**:

- You **manually acquire and release** the lock.
- You can **check for lock availability, interrupt, timeout**, and even ensure **fair access**.

- **Example of Lock using ReentrantLock:**

- `java`
- `CopyEdit`
- `import java.util.concurrent.locks.ReentrantLock;`
- 
- `public class SharedResource {`
- `private final ReentrantLock lock = new ReentrantLock();`
- 
- `public void printData() {`
- `lock.lock(); // manual acquire`
- `try {`
- `System.out.println("Inside lock-protected section");`

- } finally {
- lock.unlock(); // must release manually
- }
- }
- }

- ◆ **Key Features of Lock:**

| • <b>Feature</b>    | • <b>Description</b>                                  |
|---------------------|-------------------------------------------------------|
| • Manual control    | • You control when to acquire and release             |
| • Interruptible     | • Can interrupt waiting threads (lockInterruptibly()) |
| • Timeout support   | • tryLock(timeout, TimeUnit)                          |
| • Fairness policy   | • Can use fair locking (FIFO)                         |
| • Condition support | • Supports multiple wait conditions (newCondition())  |

---

-  **Side-by-Side Comparison: Monitor vs Lock**

|                           |                                       |                                                |
|---------------------------|---------------------------------------|------------------------------------------------|
| • <b>Feature</b>          | • <b>Monitor<br/>(synchronized)</b>   | • <b>Lock (ReentrantLock or<br/>others)</b>    |
| • How it's used           | • synchronized keyword                | • Explicit lock() / unlock()                   |
| • Lock release            | • Automatic (on method/block exit)    | • Manual (must call unlock() in finally)       |
| • Interruptible lock wait | • ✗ No                                | • ✓ Yes (lockInterruptibly())                  |
| • Timeout on wait         | • ✗ No                                | • ✓ Yes (tryLock(timeout))                     |
| • Fairness                | • ✗ No guarantee                      | • ✓ Yes (constructor: new ReentrantLock(true)) |
| • Condition variables     | • ✗ Only one per object (wait/notify) | • ✓ Multiple via Condition                     |

- **Feature**
  - Monitor (synchronized)
  - Lock (ReentrantLock or others)
  - Read/Write lock support
  - ✗ Not available
  - Yes (ReentrantReadWriteLock)
  - Performance
  - ✗ Lightweight (JVM optimized)
  - Heavier but more flexible
  - Usage Complexity
  - ✗ Simple
  - More complex, needs manual handling
- 

- **When to Use What?**

- **Situation**
  - Recommended
  - Simple mutual exclusion
  - synchronized (monitor)
  - Need for timeout or interrupt
  - Lock
  - Need for read/write separation
  - ReentrantReadWriteLock
  - Need for fairness
  - ReentrantLock(true)
  - Multiple conditions on same lock
  - Lock + Condition
- 

- **Extra: Condition Variables (wait/notify vs Condition)**

- In a monitor (i.e., with synchronized), we use wait(), notify(), and notifyAll() for condition synchronization.
- In Lock, we can create multiple Condition objects, providing better separation for multiple wait-sets.

- **Example with Condition:**

- java
- CopyEdit
- ConditionnotEmpty = lock.newCondition();
- 
- lock.lock();

- try{
- while (queue.isEmpty()) {
- notEmpty.await(); // waits for condition
- }
- // consume
- } finally{
- lock.unlock();
- }

#### • 4. What is Semaphore and Mutex?

##### • ◆ **Semaphore:**

- Controls access to a resource using a **permit count**.
- Allows multiple threads to acquire it (like a gate with tokens).
- java
- CopyEdit
- Semaphore sem = new Semaphore(3); // max 3 threads allowed
- sem.acquire();
- try{
- // critical section
- } finally{
- sem.release();
- }

##### • ◆ **Mutex (Mutual Exclusion):**

- Special case of Semaphore with **only one permit**.
- Allows only **one thread** at a time.
- java
- CopyEdit
- Semaphore mutex = new Semaphore(1);
- Or use ReentrantLock as a mutex:

- java
  - CopyEdit
  - Lock lock = new ReentrantLock();
  - lock.lock();
  - try{
  - // only one thread at a time
  - }finally{
  - lock.unlock();
  - }
- 

- **5. How to Tackle Race Conditions?**

- **Race condition** happens when multiple threads access shared data and try to modify it simultaneously without proper synchronization.

- **Ways to Prevent Race Conditions:**

- | • Technique                   | • Explanation                                  |
|-------------------------------|------------------------------------------------|
| • synchronized                | • Lock critical section                        |
| • ReentrantLock               | • Explicit locking                             |
| • Atomic classes              | • Like AtomicInteger, AtomicReference          |
| • Thread-safe data structures | • Like ConcurrentHashMap, CopyOnWriteArrayList |
| • volatile keyword            | • Ensures visibility (but not atomicity)       |
| • Immutable objects           | • No shared mutable state                      |
| • Avoid sharing state         | • Design without shared mutable data           |

- **Example:**

- java
- CopyEdit
- AtomicInteger counter = new AtomicInteger(0);

- 
- Runnable task = () -> {
  - for (int i = 0; i < 1000; i++) {
  - counter.incrementAndGet();
  - }
  - };
- Or using synchronized:
- java
- CopyEdit
- synchronized void increment()
- Here are some **frequently asked output-based String questions** in Java that are commonly used to test understanding of **immutability, interning, == vs .equals(), string pool, and concatenation behavior**:
- ---
- **◆ 1. String Interning and == vs .equals()**
- java
- CopyEdit
- public class Test {
- public static void main(String[] args) {
- String s1 = "hello";
- String s2 = "hello";
- String s3 = new String("hello");
- }
- System.out.println(s1 == s2); // ?
- System.out.println(s1 == s3); // ?
- System.out.println(s1.equals(s3)); // ?
- }
- }

- **Expected Output:**

- arduino
- CopyEdit
- true
- false
- true

- **Why:**

- s1 and s2 point to the same object in the String pool.
- s3 is a new object on the heap.
- .equals() compares content, == compares reference.

- ---

- **◆ 2. String Immutability Test**

- java
- CopyEdit
- public class Test {
- public static void main(String[] args) {
- String str = "abc";
- str.concat("def");
- System.out.println(str);
- }
- }

- **Output:**

- nginx
- CopyEdit
- abc

- **Why:**

concat() returns a new string, original string remains unchanged unless reassigned.

- \_\_\_\_\_
- ◆ **3. String Interning with intern()**
- java
- CopyEdit
- public class Test {
- public static void main(String[] args) {
- String s1 = new String("hello");
- String s2 = "hello";
- String s3 = s1.intern();
- }
- System.out.println(s1 == s2); // ?
- System.out.println(s3 == s2); // ?
- }
- }
- **Output:**
- arduino
- CopyEdit
- false
- true
- \_\_\_\_\_
- ◆ **4. Compile-Time vs Runtime Concatenation**
- java
- CopyEdit
- public class Test {
- public static void main(String[] args) {
- String s1 = "ab";
- String s2 = "cd";
- String s3 = "abcd";

- String s4 = s1 + s2;
  - 
  - System.out.println(s3 == s4); // ?
  - }
  - }
  - **Output:**
  - arduino
  - CopyEdit
  - false
  - **Why:**
- s3 is interned at compile-time. s4 is created at **runtime**, hence a different object.
- 

- 
- ◆ **5. StringBuilder vs String Concatenation**

- java
- CopyEdit
- public class Test {
- public static void main(String[] args) {
- StringBuilder sb = new StringBuilder("hello");
- sb.append("world");
- System.out.println(sb);
- }
- }

- **Output:**

- nginx
- CopyEdit
- helloworld

- 
- - ◆ **6. Tricky Immutability**

- java
- CopyEdit
- public class Test {
- public static void main(String[] args) {
- String s = "abc";
- s.toUpperCase();
- System.out.println(s);
- }
- }
- **Output:**
- nginx
- CopyEdit
- abc
- **Why:**

toUpperCase() returns a new object, s remains unchanged.

- ---
- **◆ 7. Null String Comparison**

- java
- CopyEdit
- public class Test {
- public static void main(String[] args) {
- String s = null;
- if ("test".equals(s)) {
- System.out.println("Match");
- } else {
- System.out.println("No Match");
- }
- }

- }
  - **Output:**
  - sql
  - CopyEdit
  - No Match
  - **Safe comparison:** "test".equals(s) avoids NullPointerException.
- 

- **◆ 8. String hashCode Consistency**

- java
- CopyEdit
- public class Test {
- public static void main(String[] args){
- String a = "abc";
- String b = "abc";
- System.out.println(a.hashCode() == b.hashCode()); // ?
- }
- }

- **Output:**

- arduino
- CopyEdit
- true

---

- **◆ 9. String vs StringBuffer**

- java
- CopyEdit
- public class Test {
- public static void main(String[] args){
- StringBuffer sb = new StringBuffer("Hello");

- sb.append("World");
- System.out.println(sb);
- }
- }

- **Output:**

- nginx
- CopyEdit
- HelloWorld

- \_\_\_\_\_

- **◆ 10. Unicode in String**

- java
- CopyEdit
- public class Test {
- public static void main(String[] args) {
- String s = "\u0041"; // Unicode for 'A'
- System.out.println(s);
- }
- }

- **Output:**

- CSS
- CopyEdit
- A

- \_\_\_\_\_

- Would you like me to turn these into a practice quiz or add a few more with edge cases like null, trimming, splitting, or regex behavior?

- \_\_\_\_\_

- **◆ Interview Question: Mixing Strings and Numbers in Expressions**

- **Code Example:**

- java
  - CopyEdit
  - public class Test {
  - public static void main(String[] args) {
  - System.out.println(10 + 20 + "30");
  - System.out.println("10" + 20 + 30);
  - }
  - }
- 

-  **Expected Output:**

- yaml
  - CopyEdit
  - 3030
  - 102030
- 

-  **Explanation:**

- ► **Line 1: System.out.println(10 + 20 + "30");**
  - Evaluated **left to right**
  - $10 + 20 = 30$  (integer addition)
  - $30 + "30" \rightarrow "30" + "30" = "3030"$  (String concatenation)
  -  Output: "3030"
- 

- ► **Line 2: System.out.println("10" + 20 + 30);**

- Evaluated **left to right**
- $"10" + 20 \rightarrow "10" + "20" = "1020"$  (String)
- $"1020" + 30 \rightarrow "1020" + "30" = "102030"$  (String)
-  Output: "102030"

- \_\_\_\_\_
-  **Concepts Tested:**
- **Operator associativity:** + is left-associative
- **String concatenation vs numeric addition**
- Java promotes the expression to **String concatenation** once a String is encountered
- which is the best suited data structure for LR is singleton pattern breakable ,how? how can we modify the response coming or request going to be returned by the controller in java
- **ChatGPT said:**
- Here are **detailed answers** to your three questions with examples and interview-relevant explanations:
- \_\_\_\_\_
-  **1. Which is the Best-Suited Data Structure for LRU?**
- **LRU (Least Recently Used)** cache is used to discard the least recently used items first when the cache reaches its limit.
-  **Best Data Structure:**
- **LinkedHashMap** is most suited in Java for implementing LRU.
- It maintains insertion order **OR** access order (when configured), making it ideal for LRU eviction.
- \_\_\_\_\_
-  **Example: Custom LRU using LinkedHashMap**
- java
- CopyEdit
- class LRUCache<K, V> extends LinkedHashMap<K, V> {
- private final int capacity;
- 
- public LRUCache(int capacity) {
- super(capacity, 0.75f, true); // true = accessOrder

- this.capacity = capacity;
- }
- 
- @Override
- protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
- return size() > capacity;
- }
- }
- ◆ Usage
- java
- CopyEdit
- LRUCache<Integer, String> cache = new LRUCache<>(3);
- cache.put(1, "A");
- cache.put(2, "B");
- cache.put(3, "C");
- cache.get(1); // Access 1 => now 2 is least recently used
- cache.put(4, "D"); // Evicts 2
- ✓ Alternatives:
- **Caffeine Cache:** High-performance LRU-like Java cache.
- **Guava Cache:** Google's caching library.
- 

---

- ✓ 2. Is Singleton Pattern Breakable? How?
- Yes, the Singleton pattern **can be broken** using several techniques. This is a common interview question.
- ◆ Ways Singleton Can Be Broken:
- **Using Reflection**
- java
- CopyEdit

- Constructor<Singleton> cons = Singleton.class.getDeclaredConstructor();
- cons.setAccessible(true);
- Singleton obj2 = cons.newInstance(); // Creates another instance
- **Serialization/Deserialization**
- java
- CopyEdit
- ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("obj"));
- oos.writeObject(Singleton.getInstance());
- 
- ObjectInputStream ois = new ObjectInputStream(new FileInputStream("obj"));
- Singleton obj2 = (Singleton) ois.readObject(); // New instance
- **Cloning**
- java
- CopyEdit
- Singleton obj2 = (Singleton) obj1.clone(); // Creates new instance
- 
- ---
- **Solutions:**
- **Enum Singleton** (recommended):
- java
- CopyEdit
- public enum Singleton {
- INSTANCE;
- }
- For serialization: implement readResolve()
- java
- CopyEdit
- protected Object readResolve() {

- return getInstance();
- }
- For reflection: throw exception from constructor if instance already exists
- 
- ---
- **3. How to Modify Request or Response in Spring Controller?**
- Spring provides interceptors and response advice to modify the request/response.
- 
- ---
- **◆ Modify Request: Using HandlerInterceptor**
- java
- CopyEdit
- @Component
- public class CustomRequestInterceptor implements HandlerInterceptor {
- @Override
- public boolean preHandle(HttpServletRequest request,
- HttpServletResponse response, Object handler) {
- System.out.println("Intercepted Request URI: " + request.getRequestURI());
- return true; // continue to controller
- }
- }
- Register it with WebMvcConfigurer:
- java
- CopyEdit
- @Configuration
- public class WebConfig implements WebMvcConfigurer {
- public void addInterceptors(InterceptorRegistry registry) {
- registry.addInterceptor(new CustomRequestInterceptor());
- }

- }
- 
- **◆ Modify Response: Using ResponseBodyAdvice**
- java
- CopyEdit
- @RestControllerAdvice
- public class CustomResponseAdvice implements ResponseBodyAdvice<Object> {
  - 
  - @Override
  - public boolean supports(MethodParameter returnType, Class converterType) {
    - return true; // apply to all responses
}
  - }
  - 
  - @Override
  - public Object beforeBodyWrite(Object body,
    - MethodParameter returnType,
    - MediaType selectedContentType,
    - Class selectedConverterType,
    - ServerHttpRequest request,
    - ServerHttpResponse response) {
      - 
      - // Modify the response body
      - return Map.of("status", "success", "data", body);
}
    - }
    - }
  - ✓ 1. What is GuardDuty and Inspector (EC2) in AWS?

- **Tool** • **Description**
- **GuardDuty** • A **threat detection service** that continuously monitors for **malicious or unauthorized behavior** like crypto mining, brute force, port scanning etc. It uses **VPC Flow Logs**, DNS logs, and CloudTrail events.
- **Inspector (EC2 Inspector)** • An **automated vulnerability management** service that scans **EC2 instances, Lambda and ECR containers** for OS vulnerabilities and security misconfigurations.
- ◇ **GuardDuty Use Cases:**
  - Detect suspicious API calls
  - Detect abnormal network activity
  - Identify compromised instances
- ◇ **Inspector Use Cases:**
  - Find CVEs (vulnerabilities) in EC2
  - Scan software packages
  - Monitor for best practices (e.g., SSH open to world)
- ---
- **2. Which Logging Service Will You Use from AWS?**
- **AWS CloudWatch Logs** is the **primary logging service** used for:
  - **EC2 logs**
  - **Lambda logs**
  - **Application logs**
  - **Custom logs**
- ◇ **Features:**
  - Log aggregation
  - Metrics and alarms
  - Log retention and filtering
  - Insights query engine
  - java

- CopyEdit
  - // Lambda logs can be seen in CloudWatch:
  - System.out.println("This will appear in CloudWatch Logs");
  - Other AWS Logging Tools:
  - **CloudTrail** – Logs API calls (auditing)
  - **X-Ray** – Distributed tracing
  - **OpenSearch** – For complex search-based log analytics
- 

- **3. What is Cold Start in AWS?**

- **Cold start** refers to the **delay** in response when a Lambda function is **invoked for the first time or after being idle**.
  - **Causes:**
  - Lambda has to create a **new container** → download code, initialize runtime and dependencies
  - **Affects:**
  - Serverless apps
  - Real-time APIs using Lambda
  - **How to Reduce It?**
  - Use **provisioned concurrency**
  - Minimize package size
  - Use lighter runtimes (e.g., Node.js over Java)
- 

- **4. Difference Between EntityManager and TransactionManager (JPA)**

- |                                                                                              |                                                                                                                                                          |                                                                                                                         |
|----------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>• <b>Feature</b></li> <li>• <b>Purpose</b></li> </ul> | <ul style="list-style-type: none"> <li>• <b>EntityManager</b></li> <li>• Interface to interact with DB (CRUD, query, manage entity lifecycle)</li> </ul> | <ul style="list-style-type: none"> <li>• <b>TransactionManager</b></li> <li>• Manages transaction boundaries</li> </ul> |
|----------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------|

- **Feature**
- **EntityManager**
- Scope
- JPA specific
- Example
- entityManager.persist(obj);
- **EntityManager Example:**
  - java
  - CopyEdit
  - @Autowired
  - private EntityManager entityManager;
  - 
  - public void saveData(MyEntity entity) {
  - entityManager.persist(entity);
  - }
- **TransactionManager Example:**
  - java
  - CopyEdit
  - @Transactional
  - public void saveInTransaction() {
  - repo.save(...);
  - }
- ---
- **5. How to Debug Lambda Expressions in Java?**
- Lambda expressions are **anonymous functions**, so debugging needs a slightly different approach.
- **Techniques:**
  - **Log inside Lambda**
  - java

- CopyEdit
- list.forEach(item -> {
  - System.out.println("Processing: " + item);
  - });
- **Use Breakpoints in IDE**  
Assign lambda to a variable so you can set a breakpoint:
- java
- CopyEdit
- Consumer<String> printer = s -> {
  - System.out.println("Value: " + s); // <-- breakpoint here
  - };
- printer.accept("abc");
- **Avoid deeply nested lambdas** for easier debugging
- **Use method references** when possible for readability
- java
- CopyEdit
- list.forEach(System.out::println);
- **Convert lambda to traditional method temporarily:**
- java
- CopyEdit
- public void process(String s) {
  - System.out.println("Debugging: " + s);
  - }
- list.forEach(this::process);
- The volatile keyword in Java is used to mark a variable **so that it is always read from and written to main memory**, not from a thread's local cache.
- ---
- **◆ Definition**

- java
- CopyEdit
- volatile int sharedValue;
- ---
-  **What it does:**
- **Ensures visibility:** Changes made by one thread to a volatile variable are immediately visible to other threads.
- **Prevents caching:** Each read of the variable is done directly from main memory, and each write is flushed to main memory.
- **Does NOT provide atomicity:** volatile does not prevent race conditions for compound actions like count++.
- ---
-  **Example Without volatile (Incorrect Behavior)**
- java
- CopyEdit
- class Flag {
- static boolean stop = false;
- 
- public static void main(String[] args) throws InterruptedException {
- new Thread(() -> {
- while (!stop) {
- // do something
- }
- }).start();
- 
- Thread.sleep(1000);
- stop = true; // may not be seen by the other thread!
- }
- }

- Without volatile, stop might be cached, and the new value true may not be visible to the thread running the loop.
- 

- **Correct Example With volatile**

- java
- CopyEdit
- class Flag {
- static volatile boolean stop = false;
- 
- public static void main(String[] args) throws InterruptedException {
- new Thread(() -> {
- while (!stop) {
- // do something
- }
- }).start();
- 
- Thread.sleep(1000);
- stop = true; // now other thread will see this change
- }
- }

---

- **Limitations:**

- volatile is **not** a replacement for synchronized when:
  - You need **atomic operations** (like incrementing a counter).
  - You need **mutual exclusion** (only one thread allowed in critical section).
- 

- **Use volatile when:**

- One thread updates a flag, others read it.

- You want lightweight visibility guarantees without locking.
- \_\_\_\_\_
- Would you like an example comparing volatile with synchronized or atomic classes like AtomicInteger?
- Ask ChatGPT
- \_\_\_\_\_

Databases Questions:

I was asked that , our tAAble employee has 3 columns name, dept, salary , and I have to ensure that no entry happens where salary is less than 10 k ,so how can i do it from db side

**ChatGPT said:**

To **ensure at the database level** that no employee record is inserted or updated with a **salary less than ₹10,000**, you should use a **CHECK constraint** on the salary column.

---

 **Option 1: Using a CHECK Constraint (Recommended)**

- ◆ If you're using Oracle, PostgreSQL, MySQL 8+, or most modern RDBMS:

sql

CopyEdit

ALTER TABLE employee

ADD CONSTRAINT chk\_salary\_min

CHECK (salary >= 10000);

- This constraint will **block any insert or update** where salary < 10000.
  - Ensures **data integrity at the schema level**.
  - No need for app-side checks.
- 

 **If you're creating the table from scratch:**

sql

CopyEdit

CREATE TABLE employee (

```

name VARCHAR(100),
dept VARCHAR(100),
salary NUMBER(10, 2),
CONSTRAINT chk_salary_min CHECK (salary >= 10000)
);

```

Question: write a query to change the status field of the table employee from 0 to 1 if the status is 0 and from 1 to 0 if the status is 1

◆ **Using CASE (Works in all SQL databases):**

```

sql
CopyEdit
UPDATE employee
SET status = CASE
    WHEN status = 0 THEN 1
    WHEN status = 1 THEN 0
END;

```

What is the difference between WHERE and HAVING? What is the difference between INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL JOIN? How do GROUP BY and ORDER BY differ? What is the use of DISTINCT? When does it impact performance? What are aggregate functions in SQL? Name at least 5.

**ChatGPT said:**

Great! Let's walk through **the first five SQL interview questions** with **detailed explanations, examples, and performance notes**.

---

**1. What is the difference between WHERE and HAVING?**

| Feature             | WHERE                                  | HAVING                                                       |
|---------------------|----------------------------------------|--------------------------------------------------------------|
| Applies to          | Individual rows (before grouping)      | Grouped rows (after GROUP BY)                                |
| Can use aggregates? | <input checked="" type="checkbox"/> No | <input checked="" type="checkbox"/> Yes (SUM(), AVG(), etc.) |

| Feature | WHERE | HAVING |
|---------|-------|--------|
|---------|-------|--------|

Used with GROUP BY No Yes

#### 🔍 Example:

sql

CopyEdit

-- Get departments where the total salary is more than 100000

-- ✗ Invalid: SUM() cannot be used in WHERE

-- SELECT dept, SUM(salary)

-- FROM employee

-- WHERE SUM(salary) > 100000

-- GROUP BY dept;

-- ✅ Correct usage

SELECT dept, SUM(salary) as total\_salary

FROM employee

GROUP BY dept

HAVING SUM(salary) > 100000;

## ✓ 2. What is the difference between INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL JOIN?

#### ◆ Joins Overview:

**Join Type   Returns...**

INNER JOIN Only matching rows in both tables

LEFT JOIN All rows from the left table + matching rows from right (if any)

RIGHT JOIN All rows from the right table + matching rows from left (if any)

**Join Type    Returns...**

FULL JOIN All rows from both tables; unmatched rows are filled with NULLs

 **Example:**

sql

CopyEdit

-- Assume: employee(dept\_id) and department(id)

-- INNER JOIN: only employees with departments

SELECT \* FROM employee e

INNER JOIN department d ON e.dept\_id = d.id;

-- LEFT JOIN: all employees, even if they have no department

SELECT \* FROM employee e

LEFT JOIN department d ON e.dept\_id = d.id;

-- RIGHT JOIN: all departments, even if no employees assigned

SELECT \* FROM employee e

RIGHT JOIN department d ON e.dept\_id = d.id;

-- FULL JOIN (only in some DBs like PostgreSQL or using UNION in Oracle)

SELECT \* FROM employee e

FULL OUTER JOIN department d ON e.dept\_id = d.id;

---

 **3. How do GROUP BY and ORDER BY differ?**

| <b>Feature</b> | <b>GROUP BY</b> | <b>ORDER BY</b> |
|----------------|-----------------|-----------------|
|----------------|-----------------|-----------------|

|         |                                 |                   |
|---------|---------------------------------|-------------------|
| Purpose | Aggregates data (e.g., by dept) | Sorts result rows |
|---------|---------------------------------|-------------------|

|                                             |                          |
|---------------------------------------------|--------------------------|
| Required for Aggregate functions like SUM() | Displaying sorted output |
|---------------------------------------------|--------------------------|

| Feature  | GROUP BY                  | ORDER BY                            |
|----------|---------------------------|-------------------------------------|
| Works on | Columns used for grouping | Any selected columns or expressions |

 **Example:**

sql

CopyEdit

-- GROUP BY: find total salary per department

SELECT dept, SUM(salary)

FROM employee

GROUP BY dept;

-- ORDER BY: sort employees by salary descending

SELECT \* FROM employee

ORDER BY salary DESC;

 You can combine both: GROUP BY dept ORDER BY SUM(salary) DESC;

---

 **4. What is the use of DISTINCT? When does it impact performance?**

 **Purpose:**

- Removes **duplicate rows** from the result set.
- Useful when you want **unique combinations** of columns.

 **Example:**

sql

CopyEdit

-- Get all unique departments

SELECT DISTINCT dept FROM employee;

-- Get unique name-department pairs

SELECT DISTINCT name, dept FROM employee;

 **Performance Impact:**

- DISTINCT forces a **sort or hash** internally to remove duplicates.
  - On large datasets, it can **slow down** queries, especially with joins or subqueries.
  - Use only when **uniqueness is necessary**.
- 

 **5. What are aggregate functions in SQL? Name at least 5.**

Aggregate functions **compute a single result from a group of rows**.

◆ **Common Aggregate Functions:**

**Function Description**

SUM() Adds values

AVG() Averages values

COUNT() Counts rows

MAX() Returns maximum value

MIN() Returns minimum value

 **Example:**

sql

CopyEdit

SELECT

COUNT(\*) AS total\_employees,

SUM(salary) AS total\_salary,

AVG(salary) AS average\_salary,

MAX(salary) AS highest\_salary,

MIN(salary) AS lowest\_salary

FROM employee;

 Aggregate functions are usually used with GROUP BY, but COUNT(\*) can also be used alone.

 2. DML & DDL What is the difference between DELETE, TRUNCATE, and DROP? Can we rollback after TRUNCATE? What is the difference between CHAR, VARCHAR, and VARCHAR2? How do you rename a column in Oracle? How do you add a new column with default value to an existing table?

**ChatGPT said:**

Great! Let's now cover **Section 2: DML & DDL** operations in detail — with answers, examples, and Oracle-specific notes where applicable.

---

 **1. What is the difference between DELETE, TRUNCATE, and DROP?**

| Feature          | DELETE                                                          | TRUNCATE                                                          | DROP                                    |
|------------------|-----------------------------------------------------------------|-------------------------------------------------------------------|-----------------------------------------|
| Type             | DML (Data Manipulation Language)                                | DDL (Data Definition Language)                                    | DDL                                     |
| Removes          | Specific rows (with WHERE)                                      | All rows (no WHERE allowed)                                       | Entire table (structure + data)         |
| Can rollback?    | <input checked="" type="checkbox"/> Yes (if inside transaction) | <input type="checkbox"/> No (commits immediately in Oracle)       | <input type="checkbox"/> No             |
| Affects triggers | <input checked="" type="checkbox"/> Yes                         | <input type="checkbox"/> No                                       | <input type="checkbox"/> No             |
| Resets identity? | <input type="checkbox"/> No                                     | <input checked="" type="checkbox"/> Yes (auto-increment counters) | <input checked="" type="checkbox"/> Yes |

 **Examples:**

sql

CopyEdit

-- DELETE: Remove only sales employees

DELETE FROM employee WHERE dept = 'Sales';

-- TRUNCATE: Remove all records quickly

TRUNCATE TABLE employee;

-- DROP: Delete table entirely

```
DROP TABLE employee;
```

---

## 2. Can we rollback after TRUNCATE?

- ◆ Answer:

 No, you cannot rollback a TRUNCATE in Oracle.

- TRUNCATE is a **DDL** command and **commits automatically**.
- Once executed, the data is **permanently removed** unless backups or flashback options are used.

 TRUNCATE bypasses the undo log — unlike DELETE.

---

## 3. What is the difference between CHAR, VARCHAR, and VARCHAR2?

| Feature         | CHAR                                                                                          | VARCHAR ( <i>Deprecated in Oracle</i> )                                                                      | VARCHAR2                                                                                                      |
|-----------------|-----------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------|
| Storage         | Fixed-length (padded with spaces)                                                             | Same as VARCHAR2                                                                                             | Variable-length                                                                                               |
| Max Length      | Up to 2000 bytes                                                                              | Same as VARCHAR2                                                                                             | Up to 4000 bytes (Oracle)                                                                                     |
| Trailing spaces | Preserved                                                                                     | Removed on comparison                                                                                        | Removed on comparison                                                                                         |
| Oracle Use      |  Supported |  Supported but deprecated |  Recommended for strings |

### Example:

sql

CopyEdit

-- CHAR(10) stores "Hi" as "Hi "

-- VARCHAR2(10) stores "Hi" as "Hi"

- ◆ Always prefer VARCHAR2 in Oracle — it's the standard for variable-length strings.
-

## 4. How do you rename a column in Oracle?

Use the RENAME COLUMN clause in an ALTER TABLE statement (Oracle 12c+):

sql

CopyEdit

```
ALTER TABLE employee
```

```
RENAME COLUMN emp_name TO full_name;
```

 This is supported in **Oracle 12c (12.2)** and above.

For older Oracle versions (before 12c), you must:

1. Create a new column.
  2. Copy data.
  3. Drop old column.
  4. Rename the new one.
- 

## 5. How do you add a new column with a default value to an existing table?

Use ALTER TABLE with ADD and DEFAULT:

sql

CopyEdit

```
ALTER TABLE employee
```

```
ADD status NUMBER(1) DEFAULT 1;
```

- This adds a new column status with default value 1.
- Future inserts will use 1 if no value is provided.
- For existing rows, Oracle automatically fills it with the default value unless NOT NULL is enforced.

 You can also make it NOT NULL:

sql

CopyEdit

```
ALTER TABLE employee
```

```
ADD status NUMBER(1) DEFAULT 1 NOT NULL;
```

---

## 1. What are the types of constraints in SQL?

SQL constraints are rules applied to table columns to **ensure data integrity**.

- ◆ **Common Types of Constraints:**

| <b>Constraint</b> | <b>Description</b> |
|-------------------|--------------------|
|-------------------|--------------------|

|          |                                         |
|----------|-----------------------------------------|
| NOT NULL | Ensures column cannot store NULL values |
|----------|-----------------------------------------|

|        |                                                                 |
|--------|-----------------------------------------------------------------|
| UNIQUE | Ensures all values in a column (or combo of columns) are unique |
|--------|-----------------------------------------------------------------|

|             |                                                            |
|-------------|------------------------------------------------------------|
| PRIMARY KEY | Uniquely identifies each row, implicitly NOT NULL + UNIQUE |
|-------------|------------------------------------------------------------|

|             |                                                      |
|-------------|------------------------------------------------------|
| FOREIGN KEY | Ensures referential integrity between related tables |
|-------------|------------------------------------------------------|

|       |                                          |
|-------|------------------------------------------|
| CHECK | Ensures values meet a specific condition |
|-------|------------------------------------------|

|         |                                                                         |
|---------|-------------------------------------------------------------------------|
| DEFAULT | Assigns default value to a column if no value is provided during INSERT |
|---------|-------------------------------------------------------------------------|

 These constraints can be applied **at column level** or **table level**.

---

## 2. How to add a CHECK constraint to an existing column?

Use the ALTER TABLE statement:

sql

CopyEdit

ALTER TABLE employee

ADD CONSTRAINT chk\_salary CHECK (salary >= 10000);

- chk\_salary is the **name of the constraint**.
- Ensures that only salaries  $\geq 10000$  can be inserted or updated.

 If any existing row violates this condition, Oracle will **reject** the operation.

---

## 3. How to define a NOT NULL constraint on an existing column?

Unlike other constraints, NOT NULL is **not added with ADD CONSTRAINT**, but with MODIFY:

sql

CopyEdit

ALTER TABLE employee

MODIFY name NOT NULL;

- You **must ensure** no NULL values exist in that column before applying this.

- ◆ You can combine with data type:

sql

CopyEdit

ALTER TABLE employee

MODIFY name VARCHAR2(100) NOT NULL;

---

#### 4. How to remove a constraint from a table?

Use ALTER TABLE ... DROP CONSTRAINT for named constraints.

sql

CopyEdit

ALTER TABLE employee

DROP CONSTRAINT chk\_salary;

- To find constraint names:

sql

CopyEdit

SELECT constraint\_name, constraint\_type

FROM user\_constraints

WHERE table\_name = 'EMPLOYEE';

 For NOT NULL, use:

sql

CopyEdit

ALTER TABLE employee

MODIFY name NULL;

---

## 5. What is a UNIQUE constraint and how is it different from a PRIMARY KEY?

| Feature              | UNIQUE                                                                          | PRIMARY KEY                                                               |
|----------------------|---------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| Uniqueness enforced? | <input checked="" type="checkbox"/> Yes                                         | <input checked="" type="checkbox"/> Yes                                   |
| Allows NULLs?        | <input checked="" type="checkbox"/> Yes (but only one NULL per column)          | <input checked="" type="checkbox"/> No (implicitly NOT NULL)              |
| One per table?       | <input checked="" type="checkbox"/> Can have <b>multiple</b> UNIQUE constraints | <input checked="" type="checkbox"/> Only <b>one</b> PRIMARY KEY per table |
| Index created?       | <input checked="" type="checkbox"/> Yes                                         | <input checked="" type="checkbox"/> Yes (automatically)                   |

### Example:

sql

CopyEdit

-- Unique constraint on email

ALTER TABLE employee

ADD CONSTRAINT uniq\_email UNIQUE (email);

-- Primary key on employee\_id

ALTER TABLE employee

ADD CONSTRAINT pk\_emp PRIMARY KEY (employee\_id);

Use UNIQUE for columns that must be unique **but can be NULL**, like email or phone.

Use PRIMARY KEY to **identify rows uniquely and non-null**.

q