Infosys Java Developer Interview Question(3-8 Yrs Experience)

Q1. How do you perform a filter + map + reduce in one stream pipeline?

Ans- al.stream().filter(I - > I %2==0).map(I-> I * I) .reduce(0,Integer::sum);

Q2. What is the difference between reduce() and collect() in java streams?

Ans - 1. Purpose

- **`reduce()`** → Used for **mathematical-style reduction**: sum, min, max, product, concatenation, etc. Produces a **single value**.
- **`collect()`** → Used for **mutable reductions**: collecting into collections (`List`, `Set`, `Map`) or building complex results. Produces a **mutable container**.

2. Return Type

- **`reduce()`** → returns a **single value** (or an `Optional<T>` if no identity is given).
- **`collect()`** → returns a **container or aggregate** (e.g., `List`, `Set`, `Map`, or even a string).

Q3. What is the difference between Collectors.groupingBy() and Collectors.PartioningBy()?

Ans. 1. `Collectors.groupingBy()`

- **Definition**: Groups elements of the stream by a *classifier function* (any mapping function).
- **Return type**: `Map<K, List<T>>` (or `Map<K, Set<T>>`, etc. if you give a downstream collector).
- **Use case**: When you want to group data into multiple categories.
- **Example**:

```
List<String> words =
List.of("apple", "banana",
"cherry", "apricot", "blueberry");
```

```
Map<Character, List<String>>
groupedByFirstLetter =
    words.stream()
         .collect(Collectors.group
ingBy(word -> word.charAt(0)));

System.out.println(groupedByFirstL
etter);
// {a=[apple, apricot], b=[banana,
blueberry], c=[cherry]}
```

Here, elements are grouped into **many buckets**, depending on the key.

2. `Collectors.partitioningBy()`

- **Definition**: A *special case of groupingBy()* where the classifier is a **predicate** (boolean test).
- **Return type**: `Map<Boolean, List<T>>`.
- **Use case**: When you want to divide data into **two groups** (true/false).
- **Example**:

```
List<String> words =
List.of("apple", "banana",
"cherry", "apricot", "blueberry");

Map<Boolean, List<String>>
partitionedByLength =
    words.stream()
         .collect(Collectors.parti
tioningBy(word -> word.length() >
5));

System.out.println(partitionedByLe
ngth);
// {false=[apple], true=[banana,
cherry, apricot, blueberry]}
```

Here, elements are split into exactly **two buckets**:

- `true` → words with length > 5
- `false` → words with length ≤ 5

Q4. How you would convert a List<Employee> into a Map<Department,List<Empl

oyee>> using streams?

Ans.

Map<Department,List<Employee>> mapResult = al.stream().collect(Collectors.groupingBy(Employee::get dept));

Q5.What is the difference between run() and start()?

Ans .

1. `start()` method

- Declared in `Thread` class.
- Creates a **new thread of execution**.
- Calls the `run()` method **in a separate call stack**.
- Once you call `start()`, the JVM schedules the thread for execution.

Example:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Running in: "
+ Thread.currentThread().getName());
    }
}

public class Test {
    public static void main(String[]
args) {
        MyThread t1 = new MyThread();
        t1.start(); // starts a new
thread
        System.out.println("Main thread:
" + Thread.currentThread().getName());
    }
}
```

Possible Output:

```
Main thread: main
Running in: Thread-0
```

👉 Notice: `run()` is executed in a **different thread (Thread-0)**.

2. `run()` method

- Declared in `Runnable` interface (implemented/overridden in `Thread` subclass).
- Defines the code that will run when the thread starts.

- If you call `run()` directly, it is treated like a **normal method call**, **not** a separate thread.

Example:

```
MyThread t2 = new MyThread();
t2.run(); // just a normal method call
System.out.println("Main thread: " +
Thread.currentThread().getName());
```

Output:

```
Running in: main
Main thread: main
```

Here, `run()` executed in the **main thread**, not a new one.

Q6.What is synchronisation? Why is it needed?

Ans

In Java, **synchronization** is the process of **controlling access to shared resources** (like variables, objects, files) by multiple threads.

It ensures that **only one thread can access a shared resource at a time**, preventing inconsistent or corrupted data.

In simple terms:

Synchronization = "**one thread at a time**" rule for shared resources.

. Why is Synchronization Needed?

Because Java supports **multithreading**:

- Multiple threads may run **concurrently**.
- If threads modify the same shared resource **without control**, problems occur (called *race conditions*).

Example of Problem (Without Synchronization)

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

    public void increment() {
        count++; // not atomic → multiple
threads may interfere
    }

    public int getCount() { return count; }
}

public class Test {
    public static void main(String[] args)
throws InterruptedException {
        Counter counter = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++)
counter.increment();
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++)
counter.increment();
```

```
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Final count: " +
counter.getCount());
    }
}
```

Expected Output:

```
2000
```

But Actual Output may be:

```
1984, 1992, etc. (varies)
```

👉 Because both threads update `count` at the same time, overwriting each other.

3. Fix Using Synchronization

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

    public synchronized void increment() {
        count++; // synchronized ensures one
thread at a time
    }

    public int getCount() { return count; }
}
```

Now the result will **always** be:

```
Final count: 2000
```

Key Points

- **Synchronization ensures thread safety.**
- Prevents **race conditions** and data inconsistency.
- Achieved in Java via:
  - `synchronized` keyword (methods/blocks).
  - Locks (`ReentrantLock`, `ReadWriteLock`).
  - Concurrent utilities (`ConcurrentHashMap`, `AtomicInteger`).

Think of synchronization like a **traffic signal at a busy intersection**:

- Without it → chaos (accidents).
- With it → safe, orderly access, even if slower.

Q7. What is false sharing? Ans.

False sharing happens when multiple threads modify **different variables** that happen to lie **close together in memory** (same CPU cache line).

Even though the variables are *independent*, the CPU cache system treats them as one block

(cache line).
This causes **unnecessary cache invalidations** and **slows down performance**.

Why It Happens

- Modern CPUs don't load variables one by one.
- They load data in chunks called **cache lines** (usually 64 bytes).
- If two threads work on different variables that fall into the same cache line:
  - One thread's update invalidates the cache line for the other.
  - CPU keeps "bouncing" cache lines between cores.
  - This creates contention even though there's **no real data dependency**.

Illustration

Example (False Sharing Scenario):

```
class Data {
    volatile long x;  //
Thread 1 writes here
    volatile long y;  //
Thread 2 writes here
}

Data data = new Data();

// Thread 1
for (int i = 0; i < 1_000_000;
i++) {
    data.x++;
}

// Thread 2
for (int i = 0; i < 1_000_000;
i++) {
    data.y++;
}
```

- `x` and `y` are **different variables**.
- But they might be placed **in the same**

cache line.

- So both threads unintentionally slow each other down.

How to Avoid False Sharing

1. **Padding**: Add unused fields to separate hot variables into different cache lines.

```
class Data {
    volatile long x;
    // padding to push y into a different cache line
    long p1, p2, p3, p4, p5, p6, p7;
    volatile long y;
}
```

8. **2)@Contended annotation** (Java 8+, but

needs JVM option `-XX:-RestrictContended`):

```
import jdk.internal.vm.annotation.Contended;

9
10  class Data {
11      @Contended
12      volatile long x;
13
14      @Contended
15      volatile long y;
16  }
17
```

Tip : **Use concurrency-friendly data structures** like `LongAdder` or `ConcurrentHashMap` that internally avoid false sharing.

Key Takeaways

- **False sharing = performance bug**, not correctness bug.
- Happens when **independent variables share the same cache line**.
- Causes **unnecessary cache invalidation** between threads.
- Fixed by **padding**, `@Contended`, or better concurrency utilities.

Think of it like **two people trying to write on two pages of the same notebook**:

- They're not editing the same sentence, but the notebook keeps being taken away and passed back, slowing both down.

Q8. How would you implement a producer-consumer problem?

Ans.

Producer → generates data and puts it in a shared buffer.

Consumer → takes data from the buffer and processes it.

The buffer has limited capacity.

We must ensure:

- Producers don't add to a *full buffer*.
- Consumers don't take from an *empty buffer*.
- No race conditions (thread safety).

Q9.What is the difference

between synchronized and Reentrant lock?

Ans

1. `synchronized` (Keyword)

- **Language-level construct** (built into JVM).
- Used to lock a **method** or a **block**.
- Implicitly acquires and releases the lock.
- Once the thread exits the block/method, the lock is **automatically released** (even if an exception occurs).
- Always **reentrant** (a thread can acquire the same lock again without deadlock).

Example:

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

public void decrement() {
    synchronized (this) {
        count--;
    }
}
```

2. `ReentrantLock` (Class in `java.util.concurrent.locks`)

- **Explicit lock class** introduced in Java 5.
- Provides greater flexibility than `synchronized`.
- Locking/unlocking must be done **manually** with `lock()` and `unlock()`.
- Supports **fairness policy** (queue threads FIFO).
- Offers **tryLock**() (non-blocking attempt) and **lockInterruptibly**() (responds to interrupts).
- Must call `unlock()` in a `finally` block to avoid deadlocks.

Example:

```
import
java.util.concurrent.locks.ReentrantL
ock;

ReentrantLock lock = new
ReentrantLock();

public void increment() {
    lock.lock();
    try {
        count++;
    } finally {
```

```
            lock.unlock(); // must
release manually
        }
}
```

When to Use What?

- Use **`synchronized`**:
  - When synchronization needs are simple.
  - For most common thread-safe methods/blocks.
  - Cleaner and less error-prone.
- Use **`ReentrantLock`**:
  - When you need **fairness**, **tryLock()**, or **interruptible locks**.
  - When you need **multiple condition variables** for more complex producer–consumer logic.
  - When fine-grained control is required.

👉 Think of it like this:

- **`synchronized` = automatic car** 🚗 (easy, but less flexible).
- **`ReentrantLock` = manual car** 🚙 (more control,

but you must manage it carefully).

Q10. What is the volatile keyword?
Ans

- `volatile` is a **Java keyword** used with **variables**.
- It tells the **JVM and CPU** that the variable's value **may be modified by multiple threads**.
- Ensures that **reads/writes to that variable are directly done in main memory**, not cached in CPU registers or thread-local caches.

In simple words: `volatile` guarantees **visibility** of changes to variables across threads.

2. Key Points About `volatile`

1. **Visibility Guarantee**
   - If Thread A updates a `volatile` variable, Thread B will see the updated value immediately.
   - Without `volatile`, Thread B may see a **stale cached value**.
2. **Atomicity**
   - `volatile` **does NOT guarantee atomicity** for compound operations like `count++`.
   - Only guarantees visibility of single reads/writes.

3 **When to Use**

- For **flags or state variables** shared between threads.
- Example: `boolean running = true;` used to stop a thread.

Q11. What is N+1 select problem?

Ans

The **N+1 Select Problem** is a common performance issue in **ORM frameworks** like **Hibernate** or **JPA** when fetching related entities from a database. It can cause **a large number of SQL queries** and drastically slow down your application.

Let me break it down step by step:

1. The Problem

Suppose you have two entities: `Author` and

`Book`, where an `Author` has many `Book`s.

```
@Entity
class Author {
    @Id
    private Long id;
    private String name;

    @OneToMany(mappedBy = "author")
    private List<Book> books;
}

@Entity
class Book {
    @Id
    private Long id;
    private String title;

    @ManyToOne
    private Author author;
}
```

Now, you want to fetch all authors **and their books**:

```
List<Author> authors = authorRepository.findAll();
for (Author author : authors) {
    System.out.println(author.getBooks().size());
}
```

2. What Actually Happens

1. The ORM executes **1 query** to fetch all authors:

```
SELECT * FROM author;
```

2

3 Then, for each author, it executes **1 query** to fetch their books:

```
SELECT * FROM book WHERE author_id = 1;
```

```
4  SELECT * FROM book WHERE author_id = 2;
5  SELECT * FROM book WHERE author_id = 3;
6  ...
7
```

If you have **N authors**, you run **1 + N queries**. Hence the name **N+1 select problem**.

3. Why It's Bad

- If `N` is large, it can cause **hundreds or thousands of queries**, drastically slowing down the application.

- Wastes database resources and increases latency.

4. How to Solve It

Solution 1: Eager Fetch / Join Fetch

Use `JOIN FETCH` to load authors and books in **one query**:

```
@Query("SELECT a FROM Author a JOIN FETCH
a.books")
List<Author> findAllAuthorsWithBooks();
```

Generates a single SQL query:

```
SELECT a.*, b.*
FROM author a
JOIN book b ON a.id = b.author_id;
```

Solution 2: Batch Fetching

Configure the ORM to fetch collections in batches instead of one by one:

```
@BatchSize(size = 10)
private List<Book> books;
```

Key Takeaway: The N+1 select problem happens when your ORM executes **1 query for the main entity + N queries for each related entity**, which can be avoided with **join fetches, batch fetching, or projections**.

Q12. How do you implement inheritance in JPA?

Ans

In **JPA**, inheritance allows you to map an **object-oriented class hierarchy** to database tables. JPA provides three main strategies for inheritance mapping. Let's go step by step.

1. Single Table Inheritance (`SINGLE_TABLE`)

- **All classes in the hierarchy are stored in a single table.**
- A **discriminator column** distinguishes the type of each row.

Example:

```
@Entity
@Inheritance(strategy =
InheritanceType.SINGLE_TABLE)
```

```
@DiscriminatorColumn(name = "vehicle_type")
public abstract class Vehicle {
    @Id
    private Long id;
    private String manufacturer;
}

@Entity
public class Car extends Vehicle {
    private int seatingCapacity;
}

@Entity
public class Bike extends Vehicle {
    private boolean hasCarrier;
}
```

Generated Table: `Vehicle`

i
d

Pros: Simple, single table, good performance
Cons: Many nullable columns for subclasses

2. Joined Table Inheritance (`JOINED`)

- Each class has its **own table**.
- The **child table references the parent table** via primary key.

Example:

```
@Entity
@Inheritance(strategy =
InheritanceType.JOINED)
public abstract class Vehicle {
```

```
    @Id
    private Long id;
    private String manufacturer;
}

@Entity
public class Car extends Vehicle {
    private int seatingCapacity;
}

@Entity
public class Bike extends Vehicle {
    private boolean hasCarrier;
}
```

Generated Tables:

- `Vehicle` → `id`, `manufacturer`
- `Car` → `id` (FK to Vehicle), `seatingCapacity`
- `Bike` → `id` (FK to Vehicle), `hasCarrier`

Pros: Normalized, no null columns
Cons: Joins needed to fetch child entities (slower)

3. Table Per Class (`TABLE_PER_CLASS`)

- Each concrete class has its **own table** with all fields of parent included.
- No parent table is used for storing data.

Example:

```
@Entity
@Inheritance(strategy =
InheritanceType.TABLE_PER_CLASS)
```

```
public abstract class Vehicle {
    @Id
    private Long id;
    private String manufacturer;
}

@Entity
public class Car extends Vehicle {
    private int seatingCapacity;
}

@Entity
public class Bike extends Vehicle {
    private boolean hasCarrier;
}
```

Generated Tables:

- `Car` → `id`, `manufacturer`, `seatingCapacity`
- `Bike` → `id`, `manufacturer`, `hasCarrier`

Pros: Simple, no joins
Cons: Queries across hierarchy are complex; IDs must be unique across all tables

Key Annotations

- `@Inheritance(strategy = ...)` → defines the inheritance strategy
- `@DiscriminatorColumn` → used in SINGLE_TABLE for type differentiation
- `@DiscriminatorValue("Car")` → optional, specify value for child type

✅ **Tip:**

- **SINGLE_TABLE** → best for performance if subclasses have few additional fields
- **JOINED** → best for normalized schema
- **TABLE_PER_CLASS** → rarely used, mainly for abstract parent with very different subclasses

Q13. How do you write a native SQL query in JPA?

Ans

1. Using `@Query` with `nativeQuery = true`

When using **Spring Data JPA**:

```
public interface
UserRepository extends
JpaRepository<User, Long> {

    @Query(
        value = "SELECT * FROM
users u WHERE u.email
```

```
= :email",
        nativeQuery = true
    )
    User
findByEmail(@Param("email")
String email);
}
```

👉 Here:

- `value` = the SQL query
- `nativeQuery = true` tells JPA it's raw SQL, not JPQL

2. Using `@NamedNativeQuery` (on Entity)

You can define a named native query on your entity class:

```
@Entity
@NamedNativeQuery(
    name =
"User.findByStatus",
    query = "SELECT * FROM
```

```
users WHERE status = ?1",
    resultClass = User.class
)
public class User {
    @Id
    private Long id;
    private String email;
    private String status;
}
```

Then call it like:

```
List<User> users =
entityManager
        .createNamedQuery("Use
r.findByStatus", User.class)
        .setParameter(1,
"ACTIVE")
        .getResultList();
```

3. Using `EntityManager.createNativeQuery()`

If you want more flexibility (manual use):

```
List<Object[]> results =
entityManager
    .createNativeQuery("SELECT
u.id, u.email FROM users u
WHERE u.status = ?")
    .setParameter(1, "ACTIVE")
    .getResultList();

for (Object[] row : results) {
    Long id = ((Number)
row[0]).longValue();
    String email = (String)
row[1];
}
```