**Q. Roles and Responsibility and Spring Boot Java Developer**

A Spring Boot developer primarily designs, develops, and maintains server-side applications using the Spring Boot framework. They are responsible for building and implementing microservices, designing RESTful APIs, and interacting with databases. They also collaborate with other developers and stakeholders to ensure the quality and performance of the application.

Here's a more detailed breakdown of the roles and responsibilities:

Development and Implementation:

* **Building Microservices:**

Spring Boot developers use the framework to create and implement microservices, which are small, independent, and reusable applications that can be deployed and scaled individually.

* **Designing RESTful APIs:**

They design and implement RESTful APIs, which are a standard way for different applications to communicate with each other.

* **Database Interaction:**

They interact with databases (SQL and NoSQL) to store and retrieve data, ensuring data integrity and efficient data management.

* **Working with the Software Development Lifecycle:**

They participate in all phases of the software development lifecycle, from requirements gathering to deployment and maintenance.

* **Writing Testable and Efficient Code:**

They write clean, well-documented, and efficient code that is easy to test and maintain.

* **Optimizing Application Performance:**

They optimize applications for maximum speed and scalability, ensuring efficient resource utilization.

Collaboration and Communication:

* **Working with Development Teams:**

They collaborate with other developers, including front-end developers, to integrate user-facing elements with server-side logic.

* **Communicating with Stakeholders:**

They communicate with product managers, business owners, and other stakeholders to understand requirements and ensure the application meets business goals.

* **Participating in Code Reviews:**

They participate in code reviews to ensure code quality and best practices.

* **Providing Technical Support:**

They provide production support for existing applications and troubleshoot issues related to API integrations.

Other Responsibilities:

* **Implementing Security Measures:**

They implement security measures to protect applications from potential threats and vulnerabilities.

* **Staying Updated with Technologies:**

They stay up-to-date with the latest Java frameworks and development trends to improve application quality and performance.

* **Documenting Technical Specifications:**

They document technical specifications, user manuals, and other relevant documentation to ensure effective communication and knowledge transfer.

* **Troubleshooting and Debugging:**

They troubleshoot and debug technical problems that arise during development and after deployment.

===========================================================================

**Q. List of checked and Unchecked exceptions**

**✅ Checked Exceptions (Compile-Time)**

These extend java.lang.Exception (but **not** RuntimeException):

| **Exception Class** | **Description** |
| --- | --- |
| IOException | Input/output failures |
| FileNotFoundException | File doesn't exist |
| SQLException | SQL database access errors |
| ParseException | Failure in parsing (e.g., date formats) |
| ClassNotFoundException | Class not found when loading |
| CloneNotSupportedException | If Object.clone() is called on non-cloneable |
| InterruptedException | Thread interruption |
| InvocationTargetException | Reflection method invocation failed |
| NoSuchMethodException | Method not found via reflection |
| InstantiationException | Object cannot be instantiated |
| TimeoutException | Operation timed out (e.g., in java.util.concurrent) |

You **must** handle or declare these using throws.

**❌ Unchecked Exceptions (Runtime)**

These extend java.lang.RuntimeException. Usually caused by programming bugs.

| **Exception Class** | **Description** |
| --- | --- |
| NullPointerException | Accessing null object |
| ArrayIndexOutOfBoundsException | Array index is invalid |
| IllegalArgumentException | Illegal argument passed |
| ArithmeticException | Division by zero, etc. |
| ClassCastException | Wrong type casting |
| IllegalStateException | Method called at the wrong time |
| NumberFormatException | Invalid number parsing |
| UnsupportedOperationException | Operation not supported |
| ConcurrentModificationException | Modification during iteration |
| StringIndexOutOfBoundsException | Invalid index in string |

These don’t need to be declared in method signatures.

**How to Remember?**

* **Checked** = **"expected issues"** (like file not found, DB issues)
* **Unchecked** = **"coding mistakes"** (like nulls, bad casts, logic bugs)

**==========================================================================**

**Q. Which Embedded Servers Does Spring Boot Support, and How to Change the Default?**

As of date, **Spring MVC supports Tomcat, Jetty, and Undertow.** Tomcat is the default application server supported by Spring Boot’s *web* starter.

**Spring WebFlux supports Reactor Netty, Tomcat, Jetty, and Undertow** with Reactor Netty as default.

In Spring MVC, to change the default, let’s say to Jetty, we need to exclude Tomcat and include Jetty in the dependencies:

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-web</**artifactId**>

<**exclusions**>

<**exclusion**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-tomcat</**artifactId**>

</**exclusion**>

</**exclusions**>

</**dependency**>

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-jetty</**artifactId**>

</**dependency**>Copy

Similarly, to change the default in WebFlux to UnderTow, we need to exclude Reactor Netty and include UnderTow in the dependencies.

===========================================================================

**Q. Use of @Primary, @Qualifier and other annotation with example**

In Spring, annotations like @Primary, @Qualifier, and others help resolve **ambiguity in dependency injection**, especially when multiple beans of the same type exist. Here's a clear explanation with examples:

**🔹 @Primary**

**✅ Use:**

* Marks a bean as the **default** when multiple candidates exist.

**🔧 Example:**

java

CopyEdit

public interface PaymentService {

void pay();

}

@Component

@Primary

public class CreditCardPaymentService implements PaymentService {

public void pay() {

System.out.println("Paid with Credit Card");

}

}

@Component

public class PaypalPaymentService implements PaymentService {

public void pay() {

System.out.println("Paid with PayPal");

}

}

@Component

public class PaymentProcessor {

@Autowired

private PaymentService paymentService; // Injects CreditCardPaymentService

}

**🔹 @Qualifier**

**✅ Use:**

* Explicitly specifies **which bean** to inject when multiple candidates are available.

**🔧 Example:**

java

CopyEdit

@Component

public class PaymentProcessor {

@Autowired

@Qualifier("paypalPaymentService")

private PaymentService paymentService;

}

**🔹 @Autowired (optional with Spring 4.3+)**

**✅ Use:**

* Injects dependencies automatically.
* Can be used on constructor, field, or setter.

**🔧 Constructor Injection:**

java

CopyEdit

@Component

public class PaymentProcessor {

private final PaymentService paymentService;

@Autowired

public PaymentProcessor(PaymentService paymentService) {

this.paymentService = paymentService;

}

}

**🔹 @Component, @Service, @Repository, @Controller**

**✅ Use:**

* Marks a class as a Spring-managed bean.
* Semantically different but functionally similar.

| **Annotation** | **Purpose** |
| --- | --- |
| @Component | Generic Spring bean |
| @Service | Business logic layer |
| @Repository | Persistence layer (DAO) |
| @Controller | MVC Controller |

**🔹 @Bean**

**✅ Use:**

* Declares a bean in a @Configuration class.

**🔧 Example:**

java

CopyEdit

@Configuration

public class AppConfig {

@Bean

public PaymentService creditCardPaymentService() {

return new CreditCardPaymentService();

}

}

**🔹 Summary Table**

| **Annotation** | **Purpose** |
| --- | --- |
| @Primary | Sets the default bean if multiple exist |
| @Qualifier | Explicitly chooses a bean |
| @Autowired | Injects dependencies automatically |
| @Component | Generic bean definition |
| @Service | Business logic annotation |
| @Repository | DAO bean with exception translation |
| @Controller | Web controller in Spring MVC |
| @Bean | Manual bean definition in config class |
|  |  |
| ======================================= |  |

**Q. Type of references in Java**In Java, **references** are how you access objects in memory. Java provides **four types of references**, each with different behaviors around **garbage collection**. These are defined in the java.lang.ref package.

**🔗 1. Strong Reference (default)**

**📌 Description:**

* Most common reference.
* As long as a strong reference exists, the object **is not eligible** for garbage collection.

**✅ Example:**

java

CopyEdit

Person p = new Person("John"); // Strong reference

**📌 GC Behavior:**

* Object p is not collected until the reference is nullified.

java

CopyEdit

p = null; // Now eligible for GC

**🌫 2. Soft Reference**

**📌 Description:**

* Used for **caching**.
* Garbage collector **may reclaim** the object if **memory is low**.

**✅ Example:**

java

CopyEdit

import java.lang.ref.SoftReference;

Person person = new Person("Amy");

SoftReference<Person> softRef = new SoftReference<>(person);

person = null; // Only soft reference remains

**📌 GC Behavior:**

* Object will remain in memory **until JVM needs space**.

**🍃 3. Weak Reference**

**📌 Description:**

* Object is collected on the **next GC cycle**, regardless of memory availability.
* Used in **memory-sensitive caches**, e.g., WeakHashMap.

**✅ Example:**

java

CopyEdit

import java.lang.ref.WeakReference;

Person person = new Person("Ben");

WeakReference<Person> weakRef = new WeakReference<>(person);

person = null;

**📌 GC Behavior:**

* Collected on the next GC, even if memory is sufficient.

**👻 4. Phantom Reference**

**📌 Description:**

* Used for **post-mortem cleanup**.
* Doesn’t prevent GC and always returns null when dereferenced.
* Must be used with a ReferenceQueue.

**✅ Example:**

java

CopyEdit

import java.lang.ref.PhantomReference;

import java.lang.ref.ReferenceQueue;

Person person = new Person("Lily");

ReferenceQueue<Person> refQueue = new ReferenceQueue<>();

PhantomReference<Person> phantomRef = new PhantomReference<>(person, refQueue);

person = null;

**📌 GC Behavior:**

* Enqueued after object is finalized but **before actual memory is reclaimed**.
* Used for managing **native resources**.

**🧠 Summary Table**

| **Reference Type** | **Cleared by GC?** | **Access Object?** | **Use Case** |
| --- | --- | --- | --- |
| Strong | ❌ Never (unless null) | ✅ Yes | Regular object usage |
| Soft | ✅ If memory is low | ✅ Yes | Caches |
| Weak | ✅ On next GC | ✅ Yes | WeakHashMap, internals |
| Phantom | ✅ After finalization | ❌ No | Native cleanup, tracking |

================================================================

**Q. If we are using a Serializable class and want it to be a Singleton, will deserializing the object result in multiple instances or a single instance?**

Great question! This dives into a common **pitfall** when combining **Singleton pattern** and **Java Serialization**.

**❓ Problem**

* You have a **singleton class** (only one instance should exist).
* You **serialize** and then **deserialize** the object.
* ❗️By default, deserialization creates a **new object**, breaking the singleton rule.

**🔴 So, Yes, you will get multiple objects unless you take special precautions.**

**✅ Solution: Use readResolve() Method**

To maintain singleton after deserialization, you must implement a readResolve() method.

**🔹 Example: Singleton with Serialization**

java

CopyEdit

import java.io.\*;

public class Singleton implements Serializable {

private static final Singleton INSTANCE = new Singleton();

private Singleton() {}

public static Singleton getInstance() {

return INSTANCE;

}

// Prevents creating new instance during deserialization

protected Object readResolve() {

return INSTANCE;

}

}

**🔹 Test Code**

java

CopyEdit

public class TestSingleton {

public static void main(String[] args) throws Exception {

Singleton instance1 = Singleton.getInstance();

// Serialize to file

ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("singleton.ser"));

out.writeObject(instance1);

out.close();

// Deserialize from file

ObjectInputStream in = new ObjectInputStream(new FileInputStream("singleton.ser"));

Singleton instance2 = (Singleton) in.readObject();

in.close();

System.out.println("instance1 hash: " + instance1.hashCode());

System.out.println("instance2 hash: " + instance2.hashCode());

}

}

**✅ Output with readResolve()**

bash

CopyEdit

instance1 hash: 12345678

instance2 hash: 12345678

**❌ Output without readResolve()**

bash

CopyEdit

instance1 hash: 12345678

instance2 hash: 87654321 <-- Different instance!

**✅ Summary**

| **Question** | **Answer** |
| --- | --- |
| Will deserialization create multiple instances of a singleton? | **Yes, by default.** |
| How to fix it? | **Implement readResolve()** in your singleton class. |

**===============================================================**

**Q. There is so many jar are there in your project then how you make sure your jar is safe any tool have you used?**

That's a great interview question focused on **security and dependency management**.

Here’s a more polished version of your question and how to answer it like a senior Java developer:

**✅ Interview-Ready Question (Corrected):**

**"In a large project with many third-party JARs, how do you ensure they are secure? Have you used any tools for dependency vulnerability scanning?"**

**✅ How to Answer:**

“As part of our development and DevSecOps pipeline, we use tools to ensure that the third-party dependencies (JARs) we use are secure and up to date. Specifically:

**🔹 Tools Commonly Used:**

| **Tool** | **Purpose** |
| --- | --- |
| **OWASP Dependency-Check** | Scans Maven/Gradle dependencies for known CVEs |
| **Snyk** | Detects and fixes security vulnerabilities in open-source dependencies |
| **Sonatype Nexus IQ / OSS Index** | Monitors libraries in Maven repositories for risks |
| **Black Duck (Synopsys)** | Full-spectrum license + vulnerability analysis |
| **JFrog Xray** | Deep binary analysis of JARs for known threats |
| **GitHub Dependabot** | Auto-updates vulnerable dependencies in GitHub projects |

To **implement security checks for JARs and dependencies** in your Java (Maven or Gradle) project, you can integrate tools like **OWASP Dependency-Check** or **Snyk** into your development workflow and CI/CD pipelines.

Below are detailed implementation steps for both tools:

**✅ 1. Using OWASP Dependency-Check in Maven**

**🔹 Step 1: Add plugin to pom.xml**

xml

CopyEdit

<build>

<plugins>

<plugin>

<groupId>org.owasp</groupId>

<artifactId>dependency-check-maven</artifactId>

<version>8.4.0</version> <!-- Use latest stable version -->

<executions>

<execution>

<goals>

<goal>check</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

**🔹 Step 2: Run the scan**

bash

CopyEdit

mvn clean install

mvn dependency-check:check

It will generate a report like:

bash

CopyEdit

target/dependency-check-report.html

================================================================

**Q. Blue-Green Deployment: If you meant "Blue", this may refer to Blue-Green Deployment, which is a DevOps deployment strategy, not a design pattern**

**Blue-Green Deployment** is a **release strategy** used to **minimize downtime and risk** when deploying new versions of applications. It's widely used in **cloud environments, CI/CD pipelines**, and **microservices**.

**✅ What is Blue-Green Deployment?**

You maintain **two identical production environments**:  
**Blue (current version)** and **Green (new version)**.

**✔ Process:**

1. **Blue** is the current live environment.
2. Deploy the new version to the **Green** environment.
3. Run tests in the Green environment (pre-prod validation).
4. Switch traffic from Blue to Green (usually with a **load balancer**).
5. If something goes wrong, rollback by redirecting traffic back to Blue.

**🔄 Traffic Routing Diagram**

text

CopyEdit

┌────────────┐

│ Users │

└────┬───────┘

▼

┌──────────────────────┐

│ Load Balancer │

└────┬────────────┬────┘

▼ ▼

┌──────┐ ┌──────┐

│ Blue │ │ Green│

└──────┘ └──────┘

(Current) (New Version)

**✅ Benefits**

| **Feature** | **Advantage** |
| --- | --- |
| **Zero Downtime** | New version is fully ready before switch |
| **Easy Rollback** | Just re-route to old (Blue) environment |
| **Safe Testing** | Test Green with real-world conditions |

**🧱 How to Implement in Java (Spring Boot) Apps**

**🔹 Tools You Might Use:**

* **Kubernetes**: 2 deployments (blue, green), switch using services or Ingress
* **AWS Elastic Beanstalk** or **App Runner**
* **NGINX / HAProxy**: Manually route traffic
* **CI/CD**: GitHub Actions, Jenkins, GitLab CI

**🔧 Example with Kubernetes:**

**blue-deployment.yaml and green-deployment.yaml**

yaml

CopyEdit

apiVersion: apps/v1

kind: Deployment

metadata:

name: app-blue

spec:

replicas: 2

template:

spec:

containers:

- name: app

image: my-app:1.0

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: app-green

spec:

replicas: 2

template:

spec:

containers:

- name: app

image: my-app:2.0

**service.yaml – pointing to Blue or Green**

yaml

CopyEdit

apiVersion: v1

kind: Service

metadata:

name: app-service

spec:

selector:

app: app-blue # Later switch to app-green

Switch the label selector in the service to move traffic.

**🧪 Test the Green environment**

Before routing users, you can:

* Run **smoke tests**
* Compare logs/metrics
* Test critical flows (login, DB write, etc.)

**🛑 Rollback Strategy**

If anything fails:

* Route traffic back to Blue
* No re-deploy needed — just switch traffic again

**📌 Summary**

| **Feature** | **Description** |
| --- | --- |
| Environments | 2 (Blue: Live, Green: Staging) |
| Downtime | Zero during switch |
| Risk | Low (rollback is instant) |
| Tools | Kubernetes, CI/CD tools, Load balancers |
| Used For | Production releases with minimal risk |

===========================================================================

**Q. How you done the oneToOne, One to many and many to one and others.**

[**https://www.youtube.com/watch?v=H\_\_ELR1y3FQ&t=1044s**](https://www.youtube.com/watch?v=H__ELR1y3FQ&t=1044s)

**1. @OneToOne**

Each entity has exactly one related entity.

**Example:**

java

CopyEdit

@Entity

public class User {

@Id @GeneratedValue

private Long id;

private String name;

@OneToOne

private Profile profile;

}

@Entity

public class Profile {

@Id @GeneratedValue

private Long id;

private String bio;

}

**2. @OneToMany and @ManyToOne**

A one-to-many relationship from one side is a many-to-one from the other.

**Example:**

java

CopyEdit

@Entity

public class Author {

@Id @GeneratedValue

private Long id;

private String name;

@OneToMany(mappedBy = "author")

private List<Book> books;

}

@Entity

public class Book {

@Id @GeneratedValue

private Long id;

private String title;

@ManyToOne

@JoinColumn(name = "author\_id")

private Author author;

}

**3. @ManyToMany**

Many entities relate to many others.

**Example:**

java

CopyEdit

@Entity

public class Student {

@Id @GeneratedValue

private Long id;

private String name;

@ManyToMany

@JoinTable(

name = "student\_course",

joinColumns = @JoinColumn(name = "student\_id"),

inverseJoinColumns = @JoinColumn(name = "course\_id")

)

private List<Course> courses;

}

@Entity

public class Course {

@Id @GeneratedValue

private Long id;

private String title;

@ManyToMany(mappedBy = "courses")

private List<Student> students;

}

| **Cascade Type** | **Description** |
| --- | --- |
| ALL | Applies all cascade operations (PERSIST, MERGE, REMOVE, REFRESH, DETACH) |
| PERSIST | When you save the parent, save the child too |
| MERGE | When you update the parent, update the child |
| REMOVE | When you delete the parent, delete the child |
| REFRESH | When you refresh the parent, refresh the child from the database |
| DETACH | When you detach the parent from persistence context, detach the child |

**✅ Example Use Case:**

**Suppose:**

* One Customer has many Orders.
* When saving or deleting a Customer, you want to also persist/delete their orders.

java

CopyEdit

@Entity

public class Customer {

@OneToMany(mappedBy = "customer", cascade = CascadeType.ALL)

private List<Order> orders;

}

**🧠 Best Practices:**

* **Avoid CascadeType.REMOVE** on @ManyToOne — it can accidentally delete shared entities.
* Use **CascadeType.ALL** carefully — don’t apply it blindly.
* Prefer **manual save** for shared or reusable entities.

**Tips:**

* Always manage the owning side carefully: use mappedBy to indicate the inverse side.
* Use cascade and fetch attributes as needed (e.g., cascade = CascadeType.ALL).
* Use DTOs for complex objects when returning responses in REST APIs.

===========================================================================

**Q. WAP in java 8 to group by Employee age and salary.**

Here's a **Java 8** example that uses **Streams** to group Employee objects **by age** and then **by salary** within each age group:

**✅ Java 8 Code: Group by Age and Salary**

java

CopyEdit

import java.util.\*;

import java.util.stream.Collectors;

class Employee {

private String name;

private int age;

private double salary;

// Constructor

public Employee(String name, int age, double salary) {

this.name = name;

this.age = age;

this.salary = salary;

}

// Getters

public String getName() {

return name;

}

public int getAge() {

return age;

}

public double getSalary() {

return salary;

}

// toString for printing

@Override

public String toString() {

return name + " (" + salary + ")";

}

}

public class GroupByExample {

public static void main(String[] args) {

List<Employee> employees = Arrays.asList(

new Employee("Alice", 30, 5000),

new Employee("Bob", 25, 4000),

new Employee("Charlie", 30, 5000),

new Employee("David", 25, 4500),

new Employee("Eve", 30, 6000),

new Employee("Frank", 25, 4000)

);

Map<Integer, Map<Double, List<Employee>>> grouped =

employees.stream()

.collect(Collectors.groupingBy(

Employee::getAge,

Collectors.groupingBy(Employee::getSalary)

));

// Print the result

grouped.forEach((age, salaryMap) -> {

System.out.println("Age: " + age);

salaryMap.forEach((salary, empList) -> {

System.out.println(" Salary: " + salary);

empList.forEach(e -> System.out.println(" " + e));

});

});

}

}

**🧾 Output:**

yaml

CopyEdit

Age: 25

Salary: 4000.0

Bob (4000.0)

Frank (4000.0)

Salary: 4500.0

David (4500.0)

Age: 30

Salary: 5000.0

Alice (5000.0)

Charlie (5000.0)

Salary: 6000.0

Eve (6000.0)

===============================================================

**Q. I have factory code written, and now they want to add Hexagon to the existing factory design pattern implementation automatically. How can this be done?**

To automatically support new shapes like Hexagon in an existing **Factory Design Pattern** implementation, you'll need to **avoid manual changes in the factory every time** a new shape is added.

This can be done by using **dynamic registration** or **reflection-based discovery**, depending on the flexibility you're allowed.

**✅ Goal:**

* Add Hexagon without editing the existing factory logic.
* Allow automatic registration or discovery.

**✅ Solution 1: Dynamic Registration with Map**

Have each shape **register itself** in a static block. The factory simply looks it up.

**Step 1: Create a common interface**

java

CopyEdit

public interface Shape {

void draw();

}

**Step 2: Factory with a registry**

java

CopyEdit

import java.util.HashMap;

import java.util.Map;

import java.util.function.Supplier;

public class ShapeFactory {

private static final Map<String, Supplier<Shape>> registry = new HashMap<>();

public static void register(String name, Supplier<Shape> supplier) {

registry.put(name.toLowerCase(), supplier);

}

public static Shape createShape(String name) {

Supplier<Shape> supplier = registry.get(name.toLowerCase());

if (supplier != null) {

return supplier.get();

}

throw new IllegalArgumentException("No such shape: " + name);

}

}

**Step 3: Shapes register themselves**

java

CopyEdit

public class Circle implements Shape {

static {

ShapeFactory.register("circle", Circle::new);

}

@Override

public void draw() {

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

}

}

java

CopyEdit

public class Hexagon implements Shape {

static {

ShapeFactory.register("hexagon", Hexagon::new);

}

@Override

public void draw() {

System.out.println("Drawing Hexagon");

}

}

**✅ Usage**

java

CopyEdit

public class Main {

public static void main(String[] args) throws Exception {

Class.forName("Circle");

Class.forName("Hexagon");

Shape shape1 = ShapeFactory.createShape("circle");

shape1.draw();

Shape shape2 = ShapeFactory.createShape("hexagon");

shape2.draw();

}

}

Class.forName(...) triggers the static block for registration. You could also scan the classpath automatically using libraries like **Reflections** or **Spring's classpath scanner**, so even this is automated.

**✅ Output**

scss

CopyEdit

Drawing Circle

Drawing Hexagon

**✅ Summary: Automatic Shape Addition in Factory Pattern**

To support automatic addition of new shapes like Hexagon in a Factory Pattern **without modifying the factory code**, use the **dynamic registration approach**:

**🔧 How It Works:**

1. **Define a common interface**: Shape with a draw() method.
2. **Factory holds a registry** (Map<String, Supplier<Shape>>) to map shape names to constructors.
3. **Each shape class registers itself** in a static block:

java

CopyEdit

static { ShapeFactory.register("hexagon", Hexagon::new); }

1. **Factory creates shapes by name** using the registry:

java

CopyEdit

Shape shape = ShapeFactory.createShape("hexagon");

**✅ Benefits:**

* No changes needed in ShapeFactory when adding new shapes.
* Fully extensible and loosely coupled.
* Can be enhanced with classpath scanning for full automation.

=====================================================================

**Q. I have issue with one of microservice in production but wanted to enable dynamic logging for better operation how is it possible?**

Enabling **dynamic (runtime) logging** in a production **microservice** is a common and effective strategy to troubleshoot issues **without restarting the service**. Here's how you can achieve it, especially if your service is built with **Spring Boot** or similar frameworks:

**✅ Solutions to Enable Dynamic Logging in Production**

**1. Spring Boot Actuator + Loggers Endpoint**

If your microservice uses **Spring Boot**, the easiest way is through **Spring Boot Actuator**.

**➤ Steps:**

1. **Add Actuator to pom.xml**:

xml

CopyEdit

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

1. **Enable the logging endpoint in application.properties**:

properties

CopyEdit

management.endpoints.web.exposure.include=loggers

1. **Change log level at runtime using HTTP**: Send a POST request:

bash

CopyEdit

curl -X POST http://localhost:8080/actuator/loggers/com.your.package \

-H 'Content-Type: application/json' \

-d '{"configuredLevel": "DEBUG"}'

1. **Get current log level**:

bash

CopyEdit

curl http://localhost:8080/actuator/loggers/com.your.package

===========================================================================

**Q.** **WAP to check anagram**

Here's a **Java program** to check if two strings are **anagrams** (i.e., they contain the same characters in a different order):

**✅ Java Code: Anagram Checker**

java

CopyEdit

import java.util.Arrays;

public class AnagramChecker {

public static void main(String[] args) {

System.out.println(areAnagrams("listen", "silent")); // true

System.out.println(areAnagrams("triangle", "integral"));// true

System.out.println(areAnagrams("hello", "world")); // false

}

public static boolean areAnagrams(String str1, String str2) {

// Remove whitespace and convert to lowercase

str1 = str1.replaceAll("\\s", "").toLowerCase();

str2 = str2.replaceAll("\\s", "").toLowerCase();

// If lengths differ, they can't be anagrams

if (str1.length() != str2.length()) {

return false;

}

// Convert strings to char arrays and sort

char[] chars1 = str1.toCharArray();

char[] chars2 = str2.toCharArray();

Arrays.sort(chars1);

Arrays.sort(chars2);

// Compare sorted arrays

return Arrays.equals(chars1, chars2);

}

}

**✅ Sample Output:**

arduino

CopyEdit

true

true

false

**🔍 Logic:**

* Removes whitespace and normalizes case.
* Sorts both strings and checks if sorted versions match.

===========================================================================

**Q. How to avoid the deadlock in coding**

**✅ Best Practices to Avoid Deadlocks**

**1. Always Acquire Locks in a Fixed Global Order**

Ensure all threads acquire multiple locks in the **same order**.

**❌ Bad Example:**

java

CopyEdit

// Thread A

synchronized(lock1) {

synchronized(lock2) { ... }

}

// Thread B

synchronized(lock2) {

synchronized(lock1) { ... } // Deadlock risk!

}

**✅ Good Example:**

java

CopyEdit

// Both threads

synchronized(lock1) {

synchronized(lock2) { ... } // Consistent order

}

**2. Use Try-Lock with Timeout (Java ReentrantLock)**

This avoids indefinite waiting.

java

CopyEdit

ReentrantLock lock1 = new ReentrantLock();

ReentrantLock lock2 = new ReentrantLock();

if (lock1.tryLock(1, TimeUnit.SECONDS)) {

try {

if (lock2.tryLock(1, TimeUnit.SECONDS)) {

try {

// critical section

} finally {

lock2.unlock();

}

}

} finally {

lock1.unlock();

}

}

**3. Avoid Nested Locks If Possible**

Keep lock sections small and avoid acquiring a lock **inside another lock**.

**4. Use High-Level Concurrency Utilities**

Java's java.util.concurrent package provides tools that reduce the need for manual locking:

* ConcurrentHashMap (instead of synchronizing a map)
* BlockingQueue (instead of manually controlling queue access)
* CountDownLatch, Semaphore, CyclicBarrier – coordination tools

These abstractions **internally manage locks safely**.

**5. Use Thread-safe Data Structures**

Many collections and objects in java.util.concurrent are **non-blocking and thread-safe**.

**✅ Use instead of raw locks:**

* ConcurrentLinkedQueue
* CopyOnWriteArrayList
* AtomicInteger, AtomicReference

**6. Avoid Long-Running Operations Inside Locks**

Keep your synchronized blocks or locked sections **short and fast**.

**❌ Don’t:**

java

CopyEdit

synchronized(lock) {

// call network service or sleep here

}

**✅ Do:**

java

CopyEdit

Data data;

synchronized(lock) {

data = sharedObject.getData();

}

// process data outside lock

**7. Deadlock Detection (Advanced)**

In production or debugging, use:

**🧰 Tools:**

* jconsole, jvisualvm, YourKit, etc.
* ThreadMXBean in code:

java

CopyEdit

ThreadMXBean mbean = ManagementFactory.getThreadMXBean();

long[] deadlockedThreads = mbean.findDeadlockedThreads();

=============================================================

**Q. How to use GCP Cloud schedular in java code.**

To use **Google Cloud Scheduler** in Java, you generally interact with **Google Cloud Scheduler API** using the **Google Cloud Client Library for Java**. **Cloud Scheduler** allows you to schedule jobs (such as HTTP requests, Pub/Sub messages, or Cloud Function invocations) on Google Cloud.

Here’s how you can integrate **Google Cloud Scheduler** into a Java application:

**Steps to Use GCP Cloud Scheduler in Java**

**1. Set Up Your Google Cloud Project**

Make sure you have the following prerequisites:

* A **Google Cloud Project** with billing enabled.
* **Google Cloud SDK** installed (optional but helpful for setting up).
* **Cloud Scheduler API** enabled in the Google Cloud Console.
* **Google Cloud credentials** (Service Account with required permissions like roles/cloudscheduler.admin).

**2. Add Dependencies**

Add the necessary **Google Cloud Client Libraries** to your project.

**Maven Dependency (for Google Cloud Scheduler):**

xml

CopyEdit

<dependency>

<groupId>com.google.cloud</groupId>

<artifactId>google-cloud-scheduler</artifactId>

<version>3.0.0</version>

</dependency>

You may also need the **Google Cloud Auth** library to authenticate:

xml

CopyEdit

<dependency>

<groupId>com.google.cloud</groupId>

<artifactId>google-cloud-auth</artifactId>

<version>1.1.0</version>

</dependency>

**Gradle Dependency (for Google Cloud Scheduler):**

gradle

CopyEdit

implementation 'com.google.cloud:google-cloud-scheduler:3.0.0'

**3. Authenticate Using Google Cloud SDK**

You can authenticate using a **Service Account** that has the necessary permissions (roles/cloudscheduler.admin, roles/iam.serviceAccountUser, etc.). First, set up the **Google Cloud credentials** by exporting your service account's JSON key:

bash

CopyEdit

export GOOGLE\_APPLICATION\_CREDENTIALS="/path/to/your-service-account-file.json"

Alternatively, you can authenticate using **Google Cloud SDK** if you're running the application locally and have already logged in:

bash

CopyEdit

gcloud auth application-default login

**4. Create Cloud Scheduler Client in Java**

Create a client for interacting with Google Cloud Scheduler API in your Java application.

**Example Java Code to Create a Cloud Scheduler Job:**

java

CopyEdit

import com.google.cloud.scheduler.v1.CloudSchedulerClient;

import com.google.cloud.scheduler.v1.CreateJobRequest;

import com.google.cloud.scheduler.v1.Job;

import com.google.cloud.scheduler.v1.JobName;

import com.google.cloud.scheduler.v1.Schedule;

import com.google.cloud.scheduler.v1.HttpTarget;

import com.google.cloud.scheduler.v1.HttpMethod;

import com.google.protobuf.Duration;

import java.io.IOException;

public class CloudSchedulerExample {

public static void main(String[] args) throws IOException {

// Your Google Cloud project, location, and job ID

String projectId = "your-project-id";

String locationId = "us-central1"; // Cloud Scheduler location

String jobId = "your-job-id"; // Unique job ID

// Initialize the Cloud Scheduler Client

try (CloudSchedulerClient client = CloudSchedulerClient.create()) {

// Create a job with HTTP target

Job job = Job.newBuilder()

.setName(JobName.of(projectId, locationId, jobId).toString())

.setSchedule("every 24 hours") // Cron format for scheduling

.setTimeZone("UTC") // Optional time zone for job

.setHttpTarget(

HttpTarget.newBuilder()

.setUri("https://your-api-endpoint.com/endpoint")

.setHttpMethod(HttpMethod.POST) // or GET, PUT, etc.

.putHeaders("Content-Type", "application/json")

.setBody("Your request body here".getBytes())

.build())

.build();

// Create the job using the client

CreateJobRequest request = CreateJobRequest.newBuilder()

.setParent(String.format("projects/%s/locations/%s", projectId, locationId))

.setJob(job)

.build();

// Call the API to create the job

Job response = client.createJob(request);

System.out.println("Job created successfully: " + response.getName());

}

}

}

**5. Important Fields in the Job Configuration**

* **Job Name**: The full path of the job, including the project, location, and job ID.
* **Schedule**: A cron expression to determine when the job will run (e.g., "every 24 hours" or "0 9 \* \* \*").
* **HttpTarget**: Defines an HTTP request to trigger. You can configure:
  + **URI**: The URL to which the request is made.
  + **Method**: The HTTP method (GET, POST, PUT, etc.).
  + **Headers**: Optional headers like Content-Type.
  + **Body**: The body of the HTTP request.

**6. Deploy and Run**

Once the job is created successfully, it will be automatically scheduled to run based on the cron expression you’ve set. You can check the status of your jobs in the **Google Cloud Console** under **Cloud Scheduler**.

==========================================================================

**Q. Suppose we have used Serialization in parent but don't want to Serialize child class then how is it possible.**

In Java, if you want to **serialize the parent class** but **exclude the child class** from serialization, you can use the transient keyword. When a field is marked as transient, it is **not serialized**, even if it's part of a class that implements Serializable.

**Example Scenario:**

* The **parent class** implements Serializable.
* The **child class** extends the parent class, but you do not want the child class to be serialized.

**Steps to Achieve This:**

1. **Mark the child class field as transient**: This will ensure that the child class’s fields are not serialized.
2. **Ensure only the parent class implements Serializable**: The child class should not explicitly implement Serializable if you do not want it to be serialized.

**Example Code:**

java

CopyEdit

import java.io.\*;

class Parent implements Serializable {

private String parentName;

// Constructor, getters, setters

public Parent(String parentName) {

this.parentName = parentName;

}

public String getParentName() {

return parentName;

}

public void setParentName(String parentName) {

this.parentName = parentName;

}

}

class Child extends Parent {

private transient String childName; // This field will not be serialized

// Constructor, getters, setters

public Child(String parentName, String childName) {

super(parentName);

this.childName = childName;

}

public String getChildName() {

return childName;

}

public void setChildName(String childName) {

this.childName = childName;

}

}

public class SerializationExample {

public static void main(String[] args) {

Child child = new Child("Parent A", "Child B");

try {

// Serialize the object

FileOutputStream fileOut = new FileOutputStream("childObject.ser");

ObjectOutputStream out = new ObjectOutputStream(fileOut);

out.writeObject(child);

out.close();

fileOut.close();

System.out.println("Serialized data is saved in childObject.ser");

// Deserialize the object

FileInputStream fileIn = new FileInputStream("childObject.ser");

ObjectInputStream in = new ObjectInputStream(fileIn);

Child deserializedChild = (Child) in.readObject();

in.close();

fileIn.close();

// Output the deserialized object

System.out.println("Deserialized Parent Name: " + deserializedChild.getParentName());

System.out.println("Deserialized Child Name: " + deserializedChild.getChildName()); // Should be null, as it was transient

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

**Explanation:**

* **Parent class**: The Parent class implements Serializable and has a String field (parentName).
* **Child class**: The Child class extends Parent and adds a String field (childName). The childName field is marked as transient, which means it will not be serialized.

**What Happens During Serialization:**

* The **parent's state** (i.e., parentName) is serialized.
* The **child's transient field** (childName) is not serialized, so after deserialization, childName will be null.

**Example Output:**

pgsql

CopyEdit

Serialized data is saved in childObject.ser

Deserialized Parent Name: Parent A

Deserialized Child Name: null

**Key Points:**

* **transient keyword**: It prevents the field from being serialized.
* The **parent class** still serializes its state (non-transient fields).
* The **child class** can extend the parent, but if a field is marked as transient, it won’t be serialized, even if it’s in a subclass.
* The child class **does not need to implement Serializable** if you want to exclude it from serialization.

===========================================================================

**Q. What is ENUM and when will go with instead of variable.**

**What is an ENUM in Java?**

In Java, **enum** (short for **enumeration**) is a special **class** that represents a **group of constants** (unchangeable variables, like final variables). enum types are used to represent a fixed set of constants that can be logically grouped together.

An enum is a type of **reference data type**, which can hold a set of predefined constants. These constants are typically used for things like status codes, days of the week, months of the year, directions, etc.

**Syntax of enum:**

java

CopyEdit

public enum Day {

MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

}

**Example of Using an enum:**

java

CopyEdit

public class EnumExample {

// Define an enum to represent days of the week

public enum Day {

MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

}

public static void main(String[] args) {

// Using enum values

Day today = Day.MONDAY;

if (today == Day.MONDAY) {

System.out.println("Today is Monday!");

}

}

}

**Output:**

mathematica

CopyEdit

Today is Monday!

**Key Features of ENUM:**

1. **Fixed Set of Constants**: The values of an enum are fixed and cannot be changed. For example, days of the week (Monday, Tuesday, etc.) will always remain the same.
2. **Type Safety**: Unlike regular int or String values, enum provides type safety. For example, trying to assign an invalid day name (like "Funday") will result in a compile-time error.
3. **Can Have Fields, Methods, and Constructors**: Enums can have fields, methods, and constructors just like any other class. This makes enums more powerful than simple constants.
4. **Enumerated Types Can Be Iterated**: You can iterate over the values of an enum using the values() method.

**When to Use ENUM Instead of Variables:**

1. **When You Have a Fixed Set of Constants**: If you have a predefined set of values that are known ahead of time, such as days of the week, months, states in a process, etc., enum is a better choice than using int or String constants.

**Example:**

* + **Days of the Week**: Instead of using integers like 1 for Monday, 2 for Tuesday, and so on, you can use an enum to represent days more safely and clearly:

java

CopyEdit

public enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY; }

1. **Avoiding Magic Numbers/Strings**: Magic numbers or strings (unexplained constants like 3 or "RED") can be hard to understand and maintain. Enums provide meaningful names and reduce confusion in your code.

**Example:**

* + **Traffic Light Status**: Using enum is much more readable and maintainable than using int or String values.

java

CopyEdit

public enum TrafficLight { RED, YELLOW, GREEN; }

1. **When You Need to Add Behavior to Constants**: Enums can have methods and behavior associated with each constant. If your constants need logic, enums are ideal.

**Example:**

java

CopyEdit

public enum TrafficLight {

RED(30), YELLOW(5), GREEN(45);

private int duration; // Duration in seconds

TrafficLight(int duration) {

this.duration = duration;

}

public int getDuration() {

return this.duration;

}

}

Now, you can easily retrieve the duration associated with each traffic light:

java

CopyEdit

System.out.println("Green Light Duration: " + TrafficLight.GREEN.getDuration());

1. **When You Need Type Safety**: Enums enforce type safety because they are treated as a special type. You cannot assign a value that is not part of the enum.

**Example:**

java

CopyEdit

Day today = Day.MONDAY; // Valid

today = "SUNDAY"; // Compile-time error

1. **When You Need to Use Enums in Switch Statements**: Enums work perfectly in switch statements and help make the code cleaner and more readable.

**Example:**

java

CopyEdit

Day today = Day.TUESDAY;

switch (today) {

case MONDAY:

System.out.println("Start of the week!");

break;

case TUESDAY:

System.out.println("Second day!");

break;

case WEDNESDAY:

System.out.println("Mid-week!");

break;

// other cases

default:

System.out.println("Weekend!");

break;

}

**Advantages of Using Enums Over Variables:**

| **Feature** | **Enum** | **Variables** |
| --- | --- | --- |
| **Type Safety** | Yes, prevents invalid values | No, allows any value |
| **Code Readability** | High, meaningful names | Low, needs comments or documentation |
| **Value Restrictions** | Fixed, predefined set | No restrictions, prone to errors |
| **Behavior with Constants** | Can include fields, methods, and constructors | No behavior, just values |
| **Switch Statements** | Works naturally | Not ideal for switch |
| **Error Prevention** | Helps avoid invalid constants | Prone to logical errors |
| **Ease of Use** | Easy to use and understand | Harder to maintain for large sets |

**When Not to Use Enums:**

* **When the Set of Constants is Unknown**: If the set of constants can change dynamically (e.g., new values can be added at runtime), enum is not suitable.
* **When You Need to Use Numbers/Strings Instead of Specific Constants**: If you need a more flexible or lightweight approach (e.g., if you don't need type safety or additional behavior), a simple int or String constant may suffice.

**Example with Enums vs. Strings:**

**Using Strings:**

java

CopyEdit

public class TrafficControl {

public static void controlTraffic(String lightStatus) {

if (lightStatus.equals("RED")) {

System.out.println("Stop");

} else if (lightStatus.equals("GREEN")) {

System.out.println("Go");

}

}

}

**Using Enums:**

java

CopyEdit

public enum TrafficLight { RED, GREEN, YELLOW; }

public class TrafficControl {

public static void controlTraffic(TrafficLight lightStatus) {

switch (lightStatus) {

case RED:

System.out.println("Stop");

break;

case GREEN:

System.out.println("Go");

break;

case YELLOW:

System.out.println("Slow down");

break;

}

}

}

In the enum-based approach, there’s no risk of typo errors like "RED" vs. "red", and the code is much easier to maintain.

**Summary:**

* **Use enum** when you have a **fixed set of constants** that need type safety, clarity, and possible behavior (methods).
* **Avoid variables** like int or String for representing constants in cases where you need **type safety**, clear intent, and the possibility of adding logic to constants.

===========================================================================

**Q. Java code for below input and output**

package com.interview;

import java.util.Arrays;

import java.util.List;

import java.util.Map;

import java.util.stream.Collectors;

public class EY\_Interview06052025 {

public static void main(String[] args)

//Input provided by Interviewer

List<String> input = Arrays.asList("one", "two", "three", "four", "five");

/\*

\* Output:

3, [one, two]

4, [five, four]

5, [three]

\*/

// Grouping by string length

Map<Integer, List<String>> grouped = input.stream()

.collect(Collectors.groupingBy(String::length));

System.out.println("grouped : "+grouped);

// Sorting and printing by key

grouped.entrySet().stream()

.sorted(Map.Entry.comparingByKey())

.forEach(entry -> System.out.println(entry.getKey() + ", " + entry.getValue()));

}

}

===========================================================================

**Q. Countdown latch and Cyclic barrier.**

**CountdownLatch vs CyclicBarrier in Java**

Both **CountDownLatch** and **CyclicBarrier** are synchronization utilities in Java that are used to coordinate the execution of threads, but they are suited for different purposes.

Here's a detailed explanation and comparison of both:

**1. CountdownLatch**

**CountDownLatch** is a **one-time synchronization aid** that allows one or more threads to wait until a set of operations performed by other threads is completed. It is typically used to ensure that one or more threads wait for a set of tasks to finish before proceeding.

* **Main Purpose**: To wait for a set of threads to finish their tasks before continuing execution.
* **Behavior**: The latch is initialized with a given count, and each call to countDown() reduces the count by 1. Once the count reaches 0, all waiting threads are released and can continue their execution.
* **Once the count reaches zero**: The latch cannot be reset or reused. It's one-time use.

**Key Methods:**

* countDown(): Decreases the count of the latch by one.
* await(): Causes the current thread to wait until the count reaches zero.

**Example: CountDownLatch**

java

CopyEdit

import java.util.concurrent.CountDownLatch;

public class CountdownLatchExample {

public static void main(String[] args) throws InterruptedException {

// Creating a CountdownLatch with a count of 3

CountDownLatch latch = new CountDownLatch(3);

// Thread 1

Thread t1 = new Thread(new Task(latch), "Thread 1");

// Thread 2

Thread t2 = new Thread(new Task(latch), "Thread 2");

// Thread 3

Thread t3 = new Thread(new Task(latch), "Thread 3");

t1.start();

t2.start();

t3.start();

// Main thread waits for the latch to reach zero

latch.await();

System.out.println("All tasks are finished, main thread can proceed!");

}

}

class Task implements Runnable {

private CountDownLatch latch;

public Task(CountDownLatch latch) {

this.latch = latch;

}

@Override

public void run() {

try {

// Simulate task completion

Thread.sleep((long) (Math.random() \* 1000));

System.out.println(Thread.currentThread().getName() + " completed!");

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

// Countdown after task completion

latch.countDown();

}

}

}

**How it Works:**

* In this example, the main thread waits for the other three threads to finish their tasks before it proceeds.
* Each of the worker threads (t1, t2, t3) calls latch.countDown() once their work is complete.
* The main thread calls latch.await() and only proceeds once the count reaches 0 (i.e., once all three worker threads have finished).

**2. CyclicBarrier**

**CyclicBarrier** is a **synchronization aid** that allows a set of threads to wait for each other to reach a common barrier point. It is used when you want multiple threads to wait for each other to reach a common point (called a barrier), and then they can all proceed together. Unlike CountDownLatch, it can be reused after the barrier is released.

* **Main Purpose**: To make threads wait for others to reach a common point and then continue together.
* **Behavior**: When a thread reaches the barrier, it waits until all other threads reach the barrier. Once all threads have arrived, they are released together to continue execution. The barrier can be reused after all threads have crossed it.
* **Reusability**: CyclicBarrier can be reset after all threads have passed the barrier, making it suitable for repeated use.

**Key Methods:**

* await(): Causes the current thread to wait until all threads have arrived at the barrier point.
* getNumberWaiting(): Returns the number of threads currently waiting at the barrier.
* reset(): Resets the barrier to allow reuse.

**Example: CyclicBarrier**

java

CopyEdit

import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierExample {

public static void main(String[] args) {

// Number of threads to wait for

int numberOfThreads = 3;

// Create a CyclicBarrier with a count of 3

CyclicBarrier barrier = new CyclicBarrier(numberOfThreads, () -> {

System.out.println("All threads have reached the barrier, proceeding!");

});

// Creating threads that will work concurrently and then wait at the barrier

for (int i = 0; i < numberOfThreads; i++) {

new Thread(new Task(barrier), "Thread " + (i + 1)).start();

}

}

}

class Task implements Runnable {

private CyclicBarrier barrier;

public Task(CyclicBarrier barrier) {

this.barrier = barrier;

}

@Override

public void run() {

try {

// Simulating some work before reaching the barrier

Thread.sleep((long) (Math.random() \* 1000));

System.out.println(Thread.currentThread().getName() + " reached the barrier!");

// Wait at the barrier

barrier.await();

} catch (Exception e) {

e.printStackTrace();

}

}

}

**How it Works:**

* In this example, three threads reach a CyclicBarrier.
* Once all threads reach the barrier, a callback action (which prints "All threads have reached the barrier, proceeding!") is triggered.
* After that, all threads proceed with their execution.
* The barrier is **reusable** and can be used for multiple rounds of synchronization.

**Comparison: CountdownLatch vs CyclicBarrier**

| **Feature** | **CountDownLatch** | **CyclicBarrier** |
| --- | --- | --- |
| **Purpose** | To wait for a set of operations to complete (one-time use). | To make threads wait for each other at a common point (reusable). |
| **Usage** | Useful when you need to wait for a fixed number of threads to finish (before proceeding). | Useful when you need multiple threads to synchronize at common points (can be reused). |
| **Reusability** | **Non-reusable**: Once the count reaches 0, it cannot be reset. | **Reusable**: After all threads reach the barrier, it can be reset and reused. |
| **Synchronization** | One-time synchronization after the count reaches 0. | Repeated synchronization at each barrier point. |
| **Main Method** | countDown() (decrements the count). await() (waits for count to reach 0). | await() (waits until all threads reach the barrier). |
| **Trigger Point** | The trigger is when the count reaches zero. | The trigger is when all threads have arrived at the barrier. |
| **Callback Action** | No callback. | Allows a **Runnable** callback action to be performed after all threads reach the barrier. |
| **Use Case Example** | Waiting for multiple threads to finish processing before proceeding (e.g., loading resources, waiting for multiple computations to finish). | Ensuring multiple threads reach the same point in the process, such as in simulations, parallel processing, or coordinating phases in a process. |

**When to Use Which?**

* **Use CountDownLatch** when:
  + You need to wait for a specific number of threads to finish execution before proceeding.
  + It's a one-time wait (e.g., waiting for services to start, waiting for all parts of a process to complete).
* **Use CyclicBarrier** when:
  + You need to wait for multiple threads to synchronize at a common point and then continue together.
  + You want the barrier to be **reused** for repeated synchronizations (e.g., in simulations, game rounds, or parallel processing).

==========================================================================

**Q. What is Lock interface and it's implementation**

**Lock Interface in Java**

The **Lock** interface in Java provides a more flexible and powerful mechanism for thread synchronization than the traditional synchronized keyword. It allows for more control over concurrency and gives advanced features like **try-locking**, **timed locking**, and the ability to **interrupt** thread waits.

The Lock interface is part of the **java.util.concurrent** package, and it provides methods for managing synchronization in multi-threaded applications.

**Key Methods of the Lock Interface:**

1. **lock()**: Acquires the lock. If the lock is not available, the current thread will be blocked until the lock is acquired.
2. **unlock()**: Releases the lock. This must be called after the lock is acquired in a finally block to ensure the lock is always released, preventing deadlock situations.
3. **tryLock()**: Tries to acquire the lock without blocking. It returns true if the lock was successfully acquired, and false if it was not.
4. **tryLock(long time, TimeUnit unit)**: Tries to acquire the lock within a specified time period. If the lock is not available within the given time, it returns false.
5. **lockInterruptibly()**: Acquires the lock, but the current thread can be interrupted while waiting for the lock.
6. **newCondition()**: Creates a Condition associated with this lock. Conditions are used for communication between threads, like notifying waiting threads.

**Implementations of Lock Interface:**

**1. ReentrantLock**

The most commonly used implementation of the Lock interface is **ReentrantLock**. It provides the basic lock functionality, along with additional features such as:

* **Reentrancy**: A thread that holds the lock can acquire it again without being blocked.
* **Fairness**: A fair lock gives preference to threads that are waiting for a lock (i.e., first-come-first-serve). By default, ReentrantLock is non-fair, but fairness can be enabled.
* **Interruptible locking**: ReentrantLock allows a thread to be interrupted while waiting for the lock.

**Example with ReentrantLock:**

java

CopyEdit

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class LockExample {

private static final Lock lock = new ReentrantLock();

public static void main(String[] args) {

// Starting multiple threads to demonstrate the lock usage

Thread t1 = new Thread(new Task());

Thread t2 = new Thread(new Task());

t1.start();

t2.start();

}

static class Task implements Runnable {

@Override

public void run() {

lock.lock(); // Acquire the lock

try {

// Critical section

System.out.println(Thread.currentThread().getName() + " is executing.");

Thread.sleep(1000); // Simulate work

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

lock.unlock(); // Ensure the lock is released

}

}

}

}

**Explanation:**

* In this example, two threads (t1 and t2) try to execute the critical section of code.
* The ReentrantLock ensures that only one thread can execute the critical section at a time.
* Even if a thread already holds the lock, it can acquire the lock again without causing a deadlock (reentrancy).
* The lock.unlock() is called in a finally block to ensure that the lock is always released, even if an exception occurs.

**2. ReentrantReadWriteLock**

**ReentrantReadWriteLock** is another implementation of the Lock interface that allows **readers** to access a shared resource concurrently but **excludes writers**. It provides two locks:

* **Read lock**: Multiple threads can acquire the read lock simultaneously, as long as no thread holds the write lock.
* **Write lock**: Only one thread can acquire the write lock, and no other threads can acquire the read lock while the write lock is held.

This implementation is useful in situations where you have a shared resource that is mostly read and occasionally updated.

**Example with ReentrantReadWriteLock:**

java

CopyEdit

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantReadWriteLock;

public class ReadWriteLockExample {

private static final ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock();

private static final Lock readLock = rwLock.readLock();

private static final Lock writeLock = rwLock.writeLock();

public static void main(String[] args) {

// Starting multiple threads to demonstrate the read/write locks

Thread reader1 = new Thread(new Reader());

Thread reader2 = new Thread(new Reader());

Thread writer = new Thread(new Writer());

reader1.start();

reader2.start();

writer.start();

}

static class Reader implements Runnable {

@Override

public void run() {

readLock.lock(); // Acquire the read lock

try {

System.out.println(Thread.currentThread().getName() + " is reading.");

Thread.sleep(1000); // Simulate reading

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

readLock.unlock(); // Release the read lock

}

}

}

static class Writer implements Runnable {

@Override

public void run() {

writeLock.lock(); // Acquire the write lock

try {

System.out.println(Thread.currentThread().getName() + " is writing.");

Thread.sleep(1000); // Simulate writing

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

writeLock.unlock(); // Release the write lock

}

}

}

}

**Explanation:**

* The ReentrantReadWriteLock allows multiple threads to read the resource concurrently (as long as no thread is writing).
* When a writer thread is active, no reader threads can acquire the read lock.
* The reader threads acquire the readLock and the writer thread acquires the writeLock.

**3. Other Implementations**

There are other, more specialized Lock implementations in the java.util.concurrent.locks package, including:

* **StampedLock**: A more advanced lock with an optimistic reading feature, providing better scalability in certain read-heavy situations.
* **ReadWriteLock**: The interface that ReentrantReadWriteLock implements, although it's less commonly used directly.

**Key Differences Between Lock and synchronized (Keyword)**

| **Feature** | **Lock Interface** | **synchronized Keyword** |
| --- | --- | --- |
| **Lock Acquisition** | Manual, requires lock() and unlock() | Automatic, the block or method is locked when entered |
| **Interruptible** | Yes, can use lockInterruptibly() | No, synchronized cannot be interrupted |
| **Fairness** | Can be fair (ReentrantLock supports fairness) | Not applicable, no fairness control |
| **Multiple Locks** | Can lock on multiple resources at once (multiple Lock instances) | Only one lock per block or method |
| **Reentrancy** | ReentrantLock allows reentrancy | Automatically reentrant |
| **Condition Variables** | Supports Condition for waiting threads | Uses wait(), notify(), and notifyAll() |
| **Reusability** | Locks can be reused (ReentrantLock, CyclicBarrier) | Not reusable, automatically released |

**Conclusion**

* **Lock** is a more flexible and advanced way of handling synchronization compared to synchronized. It allows greater control over thread synchronization (e.g., interruptible locks, try-lock, timed locks) and provides additional functionality like fair locking and condition variables.
* **ReentrantLock** is the most commonly used implementation of the Lock interface and is ideal for scenarios where you need manual control over locking, fairness, or interruptibility.
* **ReentrantReadWriteLock** is useful for scenarios where you have multiple readers and fewer writers, allowing concurrent reads and exclusive writes.

==========================================================================

**Q. How can prevent a Singleton class from getting cloned-**

**we need to implement/override the clone() method and throw an exception CloneNotSupportedException from the clone method**

You're absolutely right — to prevent **cloning of a Singleton class**, you must **override the clone() method** and **throw CloneNotSupportedException**. This ensures no copy of the Singleton instance can be made using the clone() method, which would otherwise break the Singleton pattern.

**✅ Why?**

Cloning creates a new object, which **violates the Singleton principle** (only one instance must exist).

**✅ Implementation Example**

java

CopyEdit

public class Singleton implements Cloneable {

private static final Singleton INSTANCE = new Singleton();

private Singleton() {

// private constructor to prevent instantiation

}

public static Singleton getInstance() {

return INSTANCE;

}

@Override

protected Object clone() throws CloneNotSupportedException {

throw new CloneNotSupportedException("Cloning of Singleton is not allowed");

}

}

**🧪 Test Code:**

java

CopyEdit

public class Main {

public static void main(String[] args) {

Singleton s1 = Singleton.getInstance();

try {

Singleton s2 = (Singleton) s1.clone(); // Throws exception

} catch (CloneNotSupportedException e) {

System.out.println("Clone prevented: " + e.getMessage());

}

}

}

**🔐 Output:**

csharp

CopyEdit

Clone prevented: Cloning of Singleton is not allowed

===========================================================================

**Q. How will you run the Docker image and deploy..tell the commands**

To **run a Docker image** and **deploy it on Kubernetes**, here’s a step-by-step guide with **commands**:

**✅ Step 1: Build Docker Image**

Make sure your Java (Spring Boot, etc.) app has a Dockerfile like:

Dockerfile

CopyEdit

FROM openjdk:17-jdk

COPY target/myapp.jar app.jar

ENTRYPOINT ["java", "-jar", "/app.jar"]

**🔹 Build the image:**

bash

CopyEdit

docker build -t myapp:latest .

**✅ Step 2: Run Docker Image Locally (Optional)**

To test it locally before deploying to Kubernetes:

bash

CopyEdit

docker run -p 8080:8080 myapp:latest

**✅ Step 3: Push to Docker Registry**

Tag and push to a registry like Docker Hub or GCR:

bash

CopyEdit

docker tag myapp:latest <your-dockerhub-username>/myapp:latest

docker push <your-dockerhub-username>/myapp:latest

**✅ Step 4: Create Deployment YAML (or use CLI)**

**Option 1: Using kubectl command:**

bash

CopyEdit

kubectl create deployment myapp --image=<your-dockerhub-username>/myapp:latest

**Option 2: With YAML:**

yaml

CopyEdit

# deployment.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: myapp

spec:

replicas: 2

selector:

matchLabels:

app: myapp

template:

metadata:

labels:

app: myapp

spec:

containers:

- name: myapp

image: <your-dockerhub-username>/myapp:latest

ports:

- containerPort: 8080

Then apply it:

bash

CopyEdit

kubectl apply -f deployment.yaml

**✅ Step 5: Expose Service**

To access your app:

bash

CopyEdit

kubectl expose deployment myapp --type=LoadBalancer --port=80 --target-port=8080

Check service URL:

bash

CopyEdit

kubectl get services

**✅ Step 6: Verify Deployment**

bash

CopyEdit

kubectl get pods

kubectl logs <pod-name>

=========================================================================

**Q. Avg salary of each department using stream**

To calculate the **average salary of each department** using **Java Streams**, you can use the Collectors.groupingBy along with Collectors.averagingDouble.

**✅ Sample Java Implementation**

**📦 Classes:**

java

CopyEdit

class Employee {

String name;

String department;

double salary;

public Employee(String name, String department, double salary) {

this.name = name;

this.department = department;

this.salary = salary;

}

public String getDepartment() {

return department;

}

public double getSalary() {

return salary;

}

}

**💡 Stream Logic:**

java

CopyEdit

import java.util.\*;

import java.util.stream.\*;

public class AvgSalaryByDept {

public static void main(String[] args) {

List<Employee> employees = Arrays.asList(

new Employee("Alice", "HR", 60000),

new Employee("Bob", "IT", 80000),

new Employee("Charlie", "HR", 70000),

new Employee("David", "IT", 90000),

new Employee("Eve", "Sales", 50000)

);

Map<String, Double> avgSalaryByDept = employees.stream()

.collect(Collectors.groupingBy(

Employee::getDepartment,

Collectors.averagingDouble(Employee::getSalary)

));

avgSalaryByDept.forEach((dept, avgSalary) ->

System.out.println(dept + ": " + avgSalary));

}

}

**✅ Output:**

makefile

CopyEdit

HR: 65000.0

IT: 85000.0

Sales: 50000.0

===========================================================================

**Q. How can you convert the list into Map**

You can convert a List into a Map in Java using the **Stream API** and the Collectors.toMap() collector.

**✅ Basic Syntax:**

java

CopyEdit

Map<KeyType, ValueType> map = list.stream()

.collect(Collectors.toMap(

item -> keyExtractor(item), // keyMapper

item -> valueExtractor(item) // valueMapper

));

**🎯 Example 1: List of Employees → Map of Name to Salary**

**📦 Employee class:**

java

CopyEdit

class Employee {

String name;

double salary;

public Employee(String name, double salary) {

this.name = name;

this.salary = salary;

}

public String getName() { return name; }

public double getSalary() { return salary; }

}

**💡 Conversion:**

java

CopyEdit

List<Employee> employees = Arrays.asList(

new Employee("Alice", 60000),

new Employee("Bob", 75000),

new Employee("Charlie", 50000)

);

Map<String, Double> nameToSalary = employees.stream()

.collect(Collectors.toMap(

Employee::getName,

Employee::getSalary

));

=========================================================================

**Q. 2nd highest salary using sql**

To get the **second highest salary** in SQL, you have multiple ways to do it. Below are the most commonly used and optimized approaches.

**✅ 1. Using LIMIT and OFFSET (MySQL, PostgreSQL)**

sql

CopyEdit

SELECT DISTINCT salary

FROM employees

ORDER BY salary DESC

LIMIT 1 OFFSET 1;

* ORDER BY salary DESC: highest first
* OFFSET 1: skip the highest
* LIMIT 1: take the second highest

✅ Works in: **MySQL**, **PostgreSQL**, **SQLite**

**✅ 2. Using Subquery with MAX (Standard SQL, Oracle, etc.)**

sql

CopyEdit

SELECT MAX(salary) AS second\_highest

FROM employees

WHERE salary < (

SELECT MAX(salary)

FROM employees

);

* Gets the **maximum salary less than the highest** salary.
* Works in **most RDBMS** including **Oracle**, **SQL Server**, etc.

===========================================================================

**Q.**

===========================================================================

1. Java implementation of the **Producer-Consumer problem** using **BlockingQueue** for thread synchronization. The **Producer** thread will produce items and add them to the queue, while the **Consumer** thread will consume those items from the queue.

**Java Producer-Consumer Program**

java

Copy

import java.util.concurrent.ArrayBlockingQueue;

import java.util.concurrent.BlockingQueue;

class Producer implements Runnable {

private BlockingQueue<Integer> queue;

public Producer(BlockingQueue<Integer> queue) {

this.queue = queue;

}

@Override

public void run() {

try {

for (int i = 1; i <= 10; i++) {

System.out.println("Produced: " + i);

queue.put(i); // Put the item in the queue (blocks if full)

Thread.sleep(500); // Simulate time taken to produce an item

}

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

}

class Consumer implements Runnable {

private BlockingQueue<Integer> queue;

public Consumer(BlockingQueue<Integer> queue) {

this.queue = queue;

}

@Override

public void run() {

try {

for (int i = 1; i <= 10; i++) {

int item = queue.take(); // Take an item from the queue (blocks if empty)

System.out.println("Consumed: " + item);

Thread.sleep(1000); // Simulate time taken to consume an item

}

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

}

public class ProducerConsumerExample {

public static void main(String[] args) {

// Create a blocking queue with a capacity of 5

BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(5);

// Create producer and consumer threads

Thread producerThread = new Thread(new Producer(queue));

Thread consumerThread = new Thread(new Consumer(queue));

// Start the threads

producerThread.start();

consumerThread.start();

}

}

**Explanation:**

* **BlockingQueue**: The BlockingQueue interface is used to implement thread-safe data structures that handle synchronization automatically. In this case, we are using an ArrayBlockingQueue, which has a fixed size. When the queue is full, the producer thread will block until space becomes available. Similarly, if the queue is empty, the consumer thread will block until an item is produced.
* **Producer**: The Producer class implements Runnable and produces integers (from 1 to 10) and places them in the queue. The queue.put(item) method blocks the producer if the queue is full.
* **Consumer**: The Consumer class implements Runnable and consumes the integers from the queue. The queue.take() method blocks the consumer if the queue is empty.
* **Threads**: The main method creates the producer and consumer threads and starts them.

**Execution:**

* The producer adds items (1 to 10) to the queue every 500 milliseconds.
* The consumer takes items from the queue and consumes them every 1000 milliseconds.
* The threads synchronize via the BlockingQueue, ensuring that the producer doesn’t overwhelm the queue, and the consumer doesn’t consume when the queue is empty.

1. **API versioning in java**

**Common API Versioning Strategies in Java**

**1. URI Versioning (Path Versioning)**

* Add the version number to the URL.
* Example:

java

CopyEdit

@RestController

@RequestMapping("/api/v1/products")

public class ProductControllerV1 {

@GetMapping

public String getProducts() {

return "Product list v1";

}

}

* **Pros:** Simple, clear versioning.
* **Cons:** Changes the URL structure, requiring clients to update their endpoints.

**2. Request Parameter Versioning**

* Pass the version as a request parameter.
* Example:

java

CopyEdit

@RestController

@RequestMapping("/api/products")

public class ProductController {

@GetMapping(params = "version=1")

public String getProductsV1() {

return "Product list v1";

}

@GetMapping(params = "version=2")

public String getProductsV2() {

return "Product list v2";

}

}

* **Pros:** No need to change the URL.
* **Cons:** Less intuitive, as versioning is hidden in parameters.

**3. Header Versioning**

* Versioning is done using a custom header.
* Example:

java

CopyEdit

@RestController

@RequestMapping("/api/products")

public class ProductController {

@GetMapping

public String getProducts(@RequestHeader("API-Version") String version) {

if ("1".equals(version)) {

return "Product list v1";

} else if ("2".equals(version)) {

return "Product list v2";

}

return "Unknown version";

}

}

* **Pros:** Clean URLs.
* **Cons:** Harder for clients to implement.

**4. Content Negotiation (Accept Header Versioning)**

* Use the Accept header for versioning.
* Example:

java

CopyEdit

@RestController

@RequestMapping("/api/products")

public class ProductController {

@GetMapping(produces = "application/vnd.company.v1+json")

public String getProductsV1() {

return "Product list v1";

}

@GetMapping(produces = "application/vnd.company.v2+json")

public String getProductsV2() {

return "Product list v2";

}

}

* **Pros:** Clean URL, follows REST principles.
* **Cons:** Clients need to set headers correctly.

**5. API Versioning Using Spring API Versioning Library**

* Use **Spring API Versioning** with annotations.
* Add **Maven dependency**:

xml

CopyEdit

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>2.9.2</version>

</dependency>

* Use version annotations:

java

CopyEdit

@RestController

@RequestMapping("/api/products")

@ApiVersion("1")

public class ProductControllerV1 {

@GetMapping

public String getProducts() {

return "Product list v1";

}

}

**Best Practices for API Versioning**

1. **Use Semantic Versioning (v1, v1.1, v2, etc.)**
2. **Deprecate Old Versions Gradually** before removing them.
3. **Document API Changes Properly** using Swagger/OpenAPI.
4. **Ensure Backward Compatibility** to avoid breaking existing clients.

===========================================================================

1. **Out of Memory (OOM) Analysis & Code Optimization in Java**

Out of memory analysis and optimising the code.

Java applications can run out of memory due to inefficient memory usage, excessive object allocation, memory leaks, or incorrect JVM configurations. Here’s how you can **analyze, debug, and optimize** Java applications to prevent OOM errors.

**1. Understanding Out of Memory (OOM) Errors**

**Common Types of OOM Errors**

1. **java.lang.OutOfMemoryError: Java heap space**
   * Happens when objects occupy more memory than allocated in the **heap**.
   * Causes:
     + Large object allocation
     + Memory leaks (e.g., unclosed resources)
     + Insufficient heap size
2. **java.lang.OutOfMemoryError: GC overhead limit exceeded**
   * Occurs when **Garbage Collection (GC)** runs excessively but cannot free up enough memory.
3. **java.lang.OutOfMemoryError: Metaspace**
   * Happens when too many classes are loaded dynamically, exceeding the metaspace limit.
4. **java.lang.OutOfMemoryError: Direct buffer memory**
   * Occurs when **NIO (Non-blocking I/O)** direct buffers exceed allocated memory.

**2. Analyzing Out of Memory Errors**

**Step 1: Check JVM Memory Usage**

* Use **JVM options** to monitor memory:

sh

CopyEdit

java -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xloggc:gc.log -jar myapp.jar

* + -XX:+PrintGCDetails: Prints detailed GC logs
  + -Xloggc:gc.log: Saves logs to a file

**Step 2: Use Heap Dump Analysis**

* Generate a heap dump when OOM occurs:

sh

CopyEdit

java -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=heapdump.hprof -jar myapp.jar

* Analyze with **Eclipse Memory Analyzer (MAT)** or **VisualVM**:
  + MAT command:

sh

CopyEdit

mat heapdump.hprof

* + Look for:
    - **Large objects** consuming memory
    - **Unclosed connections**
    - **Memory leaks** (e.g., unreferenced but retained objects)

**Step 3: Profile Memory Usage**

* Use tools like:
  + **VisualVM** (built into JDK)
  + **JProfiler** (commercial)
  + **YourKit** (commercial)
  + **Eclipse MAT** (free)

**3. Optimizing Code to Prevent OOM Errors**

**1. Optimize Data Structures**

* Use **efficient collections**:
  + Replace ArrayList with LinkedList for frequent insertions/deletions.
  + Use HashMap instead of Hashtable for better concurrency.
  + Use ConcurrentHashMap for multi-threaded access.

Example:

java

CopyEdit

// Inefficient

List<String> list = new ArrayList<>();

// Better

List<String> list = new LinkedList<>();

**2. Avoid Memory Leaks**

**a) Close Resources**

* Always close **database connections, streams, and sockets**.
* Use **try-with-resources**:

java

CopyEdit

try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {

String line = br.readLine();

} catch (IOException e) {

e.printStackTrace();

}

**b) Use Weak References for Caching**

* Prevent memory leaks in **caches** using WeakReference or SoftReference:

java

CopyEdit

Map<String, SoftReference<byte[]>> cache = new HashMap<>();

**c) Remove Unused Listeners**

* Detach event listeners when not needed:

java

CopyEdit

someComponent.removeActionListener(listener);

**3. Tune JVM Memory Settings**

* Increase heap size:

sh

CopyEdit

java -Xms512m -Xmx2g -jar myapp.jar

* + -Xms: Initial heap size
  + -Xmx: Maximum heap size
* Optimize GC settings:

sh

CopyEdit

java -XX:+UseG1GC -XX:MaxGCPauseMillis=200 -XX:+UnlockExperimentalVMOptions -jar myapp.jar

* + UseG1GC: Use **G1 Garbage Collector** for large applications.

**4. Optimize Loops and Object Creation**

**a) Use StringBuilder Instead of String Concatenation**

java

CopyEdit

// Inefficient

String result = "";

for (int i = 0; i < 1000; i++) {

result += "Hello";

}

// Optimized

StringBuilder sb = new StringBuilder();

for (int i = 0; i < 1000; i++) {

sb.append("Hello");

}

**b) Avoid Creating Unnecessary Objects**

java

CopyEdit

// Inefficient

Integer a = new Integer(10);

// Optimized

Integer a = Integer.valueOf(10);

**c) Use Streams Efficiently**

java

CopyEdit

// Avoid excessive object creation

List<Integer> numbers = IntStream.range(0, 1000)

.mapToObj(Integer::valueOf)

.collect(Collectors.toList());

**4. Debugging Real-World OOM Scenarios**

**Example 1: Memory Leak in a Long-Lived Collection**

**Issue: Holding objects in a static collection without removing them.**

java

CopyEdit

public class MemoryLeakExample {

private static final List<Object> cache = new ArrayList<>();

public void addToCache(Object obj) {

cache.add(obj);

}

}

**Solution: Use a WeakHashMap to allow GC to reclaim memory.**

java

CopyEdit

private static final Map<Object, String> cache = new WeakHashMap<>();

**Example 2: Large Object Allocation**

**Issue: Reading a huge file into memory at once.**

java

CopyEdit

byte[] data = Files.readAllBytes(Paths.get("largefile.txt"));

**Solution: Use Buffered Streams.**

java

CopyEdit

try (BufferedReader reader = new BufferedReader(new FileReader("largefile.txt"))) {

String line;

while ((line = reader.readLine()) != null) {

System.out.println(line);

}

}

**5. Preventive Measures**

✅ **Use Proper Data Structures** (HashMap, LinkedList, etc.)  
✅ **Optimize Object Creation** (Use StringBuilder, avoid redundant objects)  
✅ **Close Resources Properly** (try-with-resources)  
✅ **Tune JVM Settings** (-Xmx, -XX:+UseG1GC)  
✅ **Use Profiling Tools** (VisualVM, Eclipse MAT, JProfiler)  
✅ **Implement Caching Wisely** (WeakReference, SoftReference)  
✅ **Avoid Static References Holding Large Objects**

===========================================================================

1. **How to optimize the database query?**

**1. Use Indexing for Faster Lookups**

**🔹 What is Indexing?**

Indexes help the database find rows faster, avoiding full table scans.

**✅ Best Practices for Indexing**

1. **Index frequently queried columns** (especially in WHERE, JOIN, ORDER BY, GROUP BY).
2. **Use composite indexes** for multiple-column queries.
3. **Avoid over-indexing** (too many indexes slow down INSERT, UPDATE, DELETE).
4. **Use EXPLAIN (MySQL/PostgreSQL) or EXPLAIN ANALYZE to check index usage.**

**Example**

sql

CopyEdit

CREATE INDEX idx\_user\_email ON users(email);

java

CopyEdit

// Query using indexed column

String sql = "SELECT \* FROM users WHERE email = ?";

PreparedStatement stmt = conn.prepareStatement(sql);

stmt.setString(1, "test@example.com");

**2. Use SELECT Wisely**

**❌ Bad Practice**

sql

CopyEdit

SELECT \* FROM orders;

* Fetches unnecessary columns, increasing memory and network usage.

**✅ Optimized Query**

sql

CopyEdit

SELECT order\_id, order\_date FROM orders;

* Fetch only required columns.

**Java Code**

java

CopyEdit

String sql = "SELECT order\_id, order\_date FROM orders WHERE customer\_id = ?";

PreparedStatement stmt = conn.prepareStatement(sql);

stmt.setInt(1, customerId);

**3. Use Pagination for Large Data Sets**

**❌ Bad Practice**

sql

CopyEdit

SELECT \* FROM products;

* Loads all records at once, causing performance issues.

**✅ Optimized Query (Using LIMIT and OFFSET)**

sql

CopyEdit

SELECT \* FROM products ORDER BY product\_id LIMIT 10 OFFSET 20;

* Fetches **10 records** starting from the **21st record**.

**Java Code**

java

CopyEdit

String sql = "SELECT \* FROM products ORDER BY product\_id LIMIT ? OFFSET ?";

PreparedStatement stmt = conn.prepareStatement(sql);

stmt.setInt(1, 10); // Limit

stmt.setInt(2, 20); // Offset

* **Use keyset pagination** (WHERE id > last\_fetched\_id) for better performance.

**4. Use Prepared Statements to Prevent SQL Injection**

**❌ Bad Practice**

java

CopyEdit

String sql = "SELECT \* FROM users WHERE username = '" + username + "'";

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery(sql);

* Vulnerable to **SQL Injection**.

**✅ Optimized Query**

java

CopyEdit

String sql = "SELECT \* FROM users WHERE username = ?";

PreparedStatement stmt = conn.prepareStatement(sql);

stmt.setString(1, username);

ResultSet rs = stmt.executeQuery();

* **Precompiled SQL statements** improve performance and security.

**5. Use Joins Efficiently**

**❌ Bad Practice**

sql

CopyEdit

SELECT \* FROM orders, customers WHERE orders.customer\_id = customers.customer\_id;

* **Uses cross join (cartesian product)**, which is inefficient.

**✅ Optimized Query**

sql

CopyEdit

SELECT o.order\_id, c.name

FROM orders o

JOIN customers c ON o.customer\_id = c.customer\_id;

* **Use JOIN instead of subqueries** for better performance.

**6. Avoid N+1 Query Problem**

**Problem**

Fetching data in a loop increases DB calls, slowing down performance.

java

CopyEdit

List<Order> orders = new ArrayList<>();

for (int customerId : customerIds) {

String sql = "SELECT \* FROM orders WHERE customer\_id = ?";

PreparedStatement stmt = conn.prepareStatement(sql);

stmt.setInt(1, customerId);

ResultSet rs = stmt.executeQuery();

while (rs.next()) {

orders.add(new Order(rs.getInt("order\_id"), rs.getDate("order\_date")));

}

}

**Solution: Use IN Clause**

sql

CopyEdit

SELECT \* FROM orders WHERE customer\_id IN (1, 2, 3, 4);

* Fetch all data **in a single query**.

**Java Code**

java

CopyEdit

String sql = "SELECT \* FROM orders WHERE customer\_id IN (?)";

PreparedStatement stmt = conn.prepareStatement(sql);

stmt.setString(1, "1,2,3,4");

ResultSet rs = stmt.executeQuery();

**7. Optimize Bulk Inserts & Updates**

**❌ Bad Practice**

java

CopyEdit

for (Order order : orders) {

String sql = "INSERT INTO orders (order\_id, amount) VALUES (?, ?)";

PreparedStatement stmt = conn.prepareStatement(sql);

stmt.setInt(1, order.getId());

stmt.setDouble(2, order.getAmount());

stmt.executeUpdate();

}

* Multiple DB calls **slow performance**.

**✅ Optimized Batch Insert**

java

CopyEdit

String sql = "INSERT INTO orders (order\_id, amount) VALUES (?, ?)";

PreparedStatement stmt = conn.prepareStatement(sql);

for (Order order : orders) {

stmt.setInt(1, order.getId());

stmt.setDouble(2, order.getAmount());

stmt.addBatch();

}

stmt.executeBatch(); // Execute all queries at once

* **Batch processing reduces DB calls** and improves performance.

**8. Use Caching to Reduce DB Calls**

1. **Use Application-Level Caching** (e.g., Redis, Ehcache) for frequently accessed data.
2. **Enable Query Caching** in Hibernate.
3. **Cache Reference Data** (e.g., country codes, product categories).

**Example: Using Redis for Caching**

java

CopyEdit

Jedis jedis = new Jedis("localhost");

String cachedValue = jedis.get("user\_123");

if (cachedValue == null) {

String sql = "SELECT name FROM users WHERE id = ?";

PreparedStatement stmt = conn.prepareStatement(sql);

stmt.setInt(1, 123);

ResultSet rs = stmt.executeQuery();

if (rs.next()) {

cachedValue = rs.getString("name");

jedis.set("user\_123", cachedValue); // Store in cache

}

}

**9. Optimize Transactions**

* Use transactions **only when necessary** to avoid locking resources.
* **Commit early** to free up locks.

java

CopyEdit

Connection conn = dataSource.getConnection();

conn.setAutoCommit(false);

try {

PreparedStatement stmt1 = conn.prepareStatement("UPDATE account SET balance = balance - ? WHERE id = ?");

stmt1.setDouble(1, 100);

stmt1.setInt(2, 1);

stmt1.executeUpdate();

PreparedStatement stmt2 = conn.prepareStatement("UPDATE account SET balance = balance + ? WHERE id = ?");

stmt2.setDouble(1, 100);

stmt2.setInt(2, 2);

stmt2.executeUpdate();

conn.commit(); // Commit transaction

} catch (Exception e) {

conn.rollback(); // Rollback on failure

}

**10. Monitor Query Performance**

* Use **EXPLAIN** to analyze query execution plans.

sql

CopyEdit

EXPLAIN ANALYZE SELECT \* FROM orders WHERE customer\_id = 10;

* Use **Database Monitoring Tools**:
  + MySQL: SHOW PROCESSLIST;
  + PostgreSQL: pg\_stat\_statements
  + Oracle: V$SQL

**Summary: Best Practices for Query Optimization**

✅ **Use Indexing** (CREATE INDEX idx\_name ON table(column);)  
✅ **Select Only Needed Columns** (SELECT id, name FROM users;)  
✅ **Use Pagination** (LIMIT 10 OFFSET 20)  
✅ **Use Prepared Statements** (Precompile queries)  
✅ **Avoid N+1 Queries** (Use IN clause)  
✅ **Use Batch Processing** (executeBatch())  
✅ **Use Caching** (Redis, Ehcache)  
✅ **Optimize Transactions** (Commit early)  
✅ **Monitor Queries** (EXPLAIN ANALYZE)

=======================================================

1. **Java code to get first unique character**

**Approach:**

1. **Use a HashMap** to store the frequency of each character.
2. **Iterate the string again** to find the first character with a count of 1.

**Optimized Java Code**

java

CopyEdit

import java.util.LinkedHashMap;

import java.util.Map;

public class FirstUniqueCharacter {

public static char findFirstUniqueChar(String str) {

Map<Character, Integer> charCount = new LinkedHashMap<>();

// Step 1: Count frequency of each character

for (char ch : str.toCharArray()) {

charCount.put(ch, charCount.getOrDefault(ch, 0) + 1);

}

// Step 2: Find first character with frequency 1

for (Map.Entry<Character, Integer> entry : charCount.entrySet()) {

if (entry.getValue() == 1) {

return entry.getKey();

}

}

return '\0'; // Return null character if no unique character exists

}

public static void main(String[] args) {

String input = "swiss";

char uniqueChar = findFirstUniqueChar(input);

if (uniqueChar != '\0') {

System.out.println("First unique character: " + uniqueChar);

} else {

System.out.println("No unique character found.");

}

}

}

**Example Runs**

**✅ Input: "swiss" → Output: 'w'**

**✅ Input: "racecar" → Output: 'e'**

**✅ Input: "aabb" → Output: "No unique character found"**

===================================================

1. **How to run aur thread in managed way**

**Running a Thread in a Managed Way in Java 🚀**

To efficiently manage threads in Java, avoid creating raw threads (new Thread()). Instead, use **Executor Framework** (thread pools) for better performance, scalability, and resource management.

**1. Problems with Raw Threads**

java

CopyEdit

new Thread(() -> System.out.println("Task running")).start();

❌ **Issues:**

* No control over thread lifecycle.
* Can lead to too many threads, causing performance issues.
* Hard to handle exceptions and failures.

**2. Solution: Use Executor Framework**

**✅ Using ExecutorService**

The best way to manage threads is by using **thread pools**.

java

CopyEdit

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class ManagedThreadExample {

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(5); // 5 worker threads

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

executor.submit(() -> {

System.out.println(Thread.currentThread().getName() + " is executing");

});

}

executor.shutdown(); // Graceful shutdown

}

}

**✅ Thread Pool Types**

1. **Fixed Thread Pool**: Best for stable workloads.

java

CopyEdit

ExecutorService executor = Executors.newFixedThreadPool(5);

1. **Cached Thread Pool**: Best for short-lived tasks.

java

CopyEdit

ExecutorService executor = Executors.newCachedThreadPool();

1. **Single Thread Executor**: Ensures sequential execution.

java

CopyEdit

ExecutorService executor = Executors.newSingleThreadExecutor();

1. **Scheduled Thread Pool**: Runs tasks periodically.

java

CopyEdit

ScheduledExecutorService executor = Executors.newScheduledThreadPool(3);

executor.schedule(() -> System.out.println("Delayed Task"), 2, TimeUnit.SECONDS);

**Best Practices**

✅ **Use ExecutorService Instead of new Thread()**  
✅ **Shutdown Executors Properly (shutdown(), shutdownNow())**  
✅ **Handle Exceptions with ThreadFactory or UncaughtExceptionHandler**

**=====================================================================**

1. **How you will test integration testing**

**Integration Testing in Java 🚀**

Integration testing ensures that different modules of your application work together correctly. This typically involves testing **database interactions, REST APIs, messaging queues, and external services**.

**1. Types of Integration Testing**

✔ **Database Integration Testing** - Verify database interactions.  
✔ **REST API Integration Testing** - Test API endpoints.  
✔ **Spring Boot Integration Testing** - Test end-to-end app flow.  
✔ **Messaging Queue Testing** - Test Kafka/RabbitMQ interactions.

**2. Setting Up Integration Testing in Java**

**✅ Using JUnit + Spring Boot for Integration Testing**

For Spring Boot apps, we use **JUnit 5, Testcontainers, and MockMvc**.

**📌 Example: REST API Integration Test**

java

CopyEdit

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

@AutoConfigureMockMvc

public class UserControllerIT {

@Autowired

private MockMvc mockMvc;

@Test

public void testGetUser() throws Exception {

mockMvc.perform(get("/users/1"))

.andExpect(status().isOk())

.andExpect(jsonPath("$.name").value("John Doe"));

}

}

🔹 **MockMvc**: Mocks API calls without needing a real server.  
🔹 **@SpringBootTest**: Loads the Spring context.  
🔹 **@AutoConfigureMockMvc**: Configures MockMvc for testing.

**3. Database Integration Testing**

**✅ Using Testcontainers for Real DB Testing**

Testcontainers provide lightweight, real database instances for tests.

**📌 Example: PostgreSQL Integration Test**

java

CopyEdit

@Testcontainers

@SpringBootTest

public class UserRepositoryIT {

@Container

static PostgreSQLContainer<?> postgreSQLContainer = new PostgreSQLContainer<>("postgres:latest")

.withDatabaseName("testdb")

.withUsername("test")

.withPassword("test");

@Autowired

private UserRepository userRepository;

@Test

public void testSaveUser() {

User user = new User("Alice");

User savedUser = userRepository.save(user);

assertNotNull(savedUser.getId());

}

}

🔹 **Testcontainers** start a real PostgreSQL database inside Docker.  
🔹 **No need for an in-memory database (H2)**—ensures production-like testing.

**4. Mocking External Services**

**✅ Using WireMock to Mock APIs**

If your service calls external APIs, **mock them** instead of hitting real servers.

**📌 Example: Mocking an External API**

java

CopyEdit

@ExtendWith(SpringExtension.class)

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

@AutoConfigureWireMock(port = 8081)

public class ExternalApiServiceIT {

@Test

public void testExternalApi() throws Exception {

stubFor(get(urlEqualTo("/external-api"))

.willReturn(aResponse()

.withStatus(200)

.withBody("{\"message\": \"Success\"}")));

String response = externalApiService.callExternalApi();

assertEquals("Success", response);

}

}

🔹 **WireMock** simulates an external API response.  
🔹 **Prevents network failures affecting tests.**

**5. Running Tests in CI/CD**

* **Use Maven/Gradle for automation**

sh

CopyEdit

mvn test

* **Run integration tests in GitHub Actions**

yaml

CopyEdit

jobs:

test:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Run Tests

run: mvn test

**✅ Best Practices**

✔ **Use Testcontainers instead of in-memory DBs for realistic tests**  
✔ **Mock external dependencies using WireMock**  
✔ **Use @DirtiesContext to reset Spring context when needed**  
✔ **Run integration tests in CI/CD pipeline**

=====================================================================

1. **Security is a critical concern in microservices. Discuss a scenario where sensitive information needs to be exchanged between microservices. How would you implement secure communication and ensure data integrity? What Java technologies would you leverage for this purpose?**

**Securing Communication Between Microservices in Java 🔐**

Security is critical in microservices, especially when exchanging sensitive information like **user credentials, financial transactions, or health records**.

**📌 Scenario: Secure Payment Processing Between Microservices**

Imagine an **E-commerce System** with the following microservices:

* **Order Service**: Handles customer orders.
* **Payment Service**: Processes payments.
* **Notification Service**: Sends payment confirmation emails.

The **Order Service** needs to send sensitive payment details to the **Payment Service** securely.

**🔐 Security Challenges**

✔ **Data Confidentiality** – Prevent unauthorized access.  
✔ **Data Integrity** – Ensure data is not tampered with.  
✔ **Authentication & Authorization** – Ensure only authorized services communicate.  
✔ **Replay Attack Protection** – Prevent attackers from reusing valid requests.

**1️⃣ Secure Communication Using Mutual TLS (mTLS)**

Mutual TLS ensures **both services authenticate each other** before exchanging data.

**✅ How to Implement in Spring Boot**

**Step 1: Generate SSL Certificates**

Use OpenSSL to generate server & client certificates:

sh

CopyEdit

openssl req -newkey rsa:4096 -nodes -keyout server.key -x509 -days 365 -out server.crt

openssl req -newkey rsa:4096 -nodes -keyout client.key -x509 -days 365 -out client.crt

**Step 2: Configure SSL in application.yml**

yaml

CopyEdit

server:

ssl:

key-store: classpath:server.p12

key-store-password: secret

key-store-type: PKCS12

client-auth: need

**Step 3: Enable HTTPS in Spring Boot**

java

CopyEdit

@Bean

public RestTemplate restTemplate(RestTemplateBuilder builder) {

return builder

.requestFactory(() -> new HttpComponentsClientHttpRequestFactory())

.build();

}

✅ **Result**: Only trusted microservices can communicate using TLS encryption.

**2️⃣ Secure Data Transmission with JWT Authentication**

Each request should carry a **JWT token** signed by a trusted authority.

**Step 1: Add JWT Dependency**

xml

CopyEdit

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt</artifactId>

<version>0.11.5</version>

</dependency>

**Step 2: Generate JWT Token**

java

CopyEdit

public String generateToken(String serviceName) {

return Jwts.builder()

.setSubject(serviceName)

.setIssuedAt(new Date())

.setExpiration(new Date(System.currentTimeMillis() + 3600000)) // 1 hour

.signWith(SignatureAlgorithm.HS256, "secretKey")

.compact();

}

**Step 3: Validate JWT in API Gateway**

java

CopyEdit

public Claims validateToken(String token) {

return Jwts.parser()

.setSigningKey("secretKey")

.parseClaimsJws(token)

.getBody();

}

✅ **Result**: Only services with valid JWT tokens can access the API.

**4️⃣ Secure Microservices with OAuth 2.0 & Keycloak**

Use **OAuth 2.0** for centralized authentication and authorization.

**Step 1: Configure Keycloak**

* Create a **client** in Keycloak.
* Obtain an **access token** using client credentials.

**Step 2: Secure Spring Boot Endpoints**

java

CopyEdit

@EnableResourceServer

public class SecurityConfig extends ResourceServerConfigurerAdapter {

@Override

public void configure(HttpSecurity http) throws Exception {

http.authorizeRequests()

.antMatchers("/payment/\*\*").authenticated()

.anyRequest().permitAll();

}

}

✅ **Result**: Only authenticated services can access protected endpoints.

**🔹 Best Practices for Microservices Security**

✔ **Use HTTPS and mTLS for secure communication**  
✔ **Sign requests with JWT for authentication**  
✔ **Leverage OAuth 2.0 for centralized security**  
✔ **Monitor logs for suspicious activities using ELK or Prometheus**

**=======================================================**

1. **Tell me real example of dynamic object creation where you used reflection API.**

**Real-World Example of Dynamic Object Creation Using Reflection API 🚀**

**📌 Scenario: Dynamic DTO Mapping in a REST API**

Imagine we are building a **generic API response handler** that dynamically **maps database entities to DTOs (Data Transfer Objects)**. Instead of writing a separate mapping logic for each DTO, we use **Reflection API** to dynamically create and populate objects at runtime.

**1️⃣ Problem Statement**

We have multiple DTO classes (UserDTO, OrderDTO, etc.) and want to dynamically create instances **based on API request parameters**.

**✅ Without Reflection (Manual Mapping)**

java

CopyEdit

UserDTO userDTO = new UserDTO();

userDTO.setId(user.getId());

userDTO.setName(user.getName());

🔹 This approach is **tedious** when handling multiple DTOs dynamically.

**2️⃣ Solution: Dynamic Object Creation with Reflection API**

**Step 1: Define DTO Classes**

java

CopyEdit

public class UserDTO {

private Long id;

private String name;

// Getters and Setters

}

java

CopyEdit

public class OrderDTO {

private Long orderId;

private Double amount;

// Getters and Setters

}

**Step 2: Create a Generic Method to Instantiate DTOs**

We use **Reflection API** to **create an object dynamically** and **set fields** based on entity data.

java

CopyEdit

import java.lang.reflect.Field;

public class DTOMapper {

public static <T> T mapToDTO(Object entity, Class<T> dtoClass) {

try {

// Create instance of DTO dynamically

T dtoInstance = dtoClass.getDeclaredConstructor().newInstance();

// Get all fields from DTO class

for (Field field : dtoClass.getDeclaredFields()) {

field.setAccessible(true);

// Get corresponding field from entity

Field entityField;

try {

entityField = entity.getClass().getDeclaredField(field.getName());

entityField.setAccessible(true);

// Copy value from entity to DTO

field.set(dtoInstance, entityField.get(entity));

} catch (NoSuchFieldException ignored) {

// Skip if the field does not exist in the entity

}

}

return dtoInstance;

} catch (Exception e) {

throw new RuntimeException("DTO Mapping failed!", e);

}

}

}

**Step 3: Use Reflection to Create DTO at Runtime**

java

CopyEdit

User user = new User();

user.setId(1L);

user.setName("John Doe");

// Dynamically create UserDTO

UserDTO userDTO = DTOMapper.mapToDTO(user, UserDTO.class);

System.out.println(userDTO.getName()); // Output: John Doe

**3️⃣ Real-World Usage: Generic Response Handler in REST APIs**

We can use this approach in **Spring Boot REST Controllers** for **dynamic object mapping**.

java

CopyEdit

@RestController

@RequestMapping("/users")

public class UserController {

@Autowired

private UserService userService;

@GetMapping("/{id}")

public ResponseEntity<?> getUser(@PathVariable Long id, @RequestParam String dtoType) throws ClassNotFoundException {

User user = userService.findById(id);

// Convert String to Class Type (Dynamic DTO)

Class<?> dtoClass = Class.forName("com.example.dto." + dtoType);

// Use reflection to create DTO dynamically

Object dto = DTOMapper.mapToDTO(user, dtoClass);

return ResponseEntity.ok(dto);

}

}

**📌 API Call Example**

sh

CopyEdit

GET /users/1?dtoType=UserDTO

===========================================================================

1. **How to implement where Data transactional and data analytical is running same time and how to handle real scenario in java microservices.**

Handling Concurrent Transactional & Analytical Workloads in Java Microservices 🚀

In modern microservices architectures, applications often need to handle both:  
1️⃣ Transactional Workloads (OLTP - Online Transaction Processing): Real-time CRUD operations on databases.  
2️⃣ Analytical Workloads (OLAP - Online Analytical Processing): Aggregations, reports, and complex queries for insights.

📌 Real-World Scenario: E-commerce Order Processing & Analytics

Imagine an E-commerce System with the following microservices:

| Microservice | Function | Workload Type |
| --- | --- | --- |
| Order Service | Handles real-time customer orders | Transactional (OLTP) |
| Inventory Service | Updates stock after each order | Transactional (OLTP) |
| Analytics Service | Generates reports on sales trends | Analytical (OLAP) |

Problem: Running OLAP queries on the same OLTP database can cause:  
✔ Performance bottlenecks – Expensive queries slow down transactions.  
✔ Deadlocks – Competing read/write operations lock the database.  
✔ Scalability issues – High traffic affects user experience.

✅ Solution: Separate OLTP & OLAP with Event-Driven Architecture

To efficiently handle real-time transactions & analytics, we decouple OLTP and OLAP using:  
✔ CQRS (Command Query Responsibility Segregation)  
✔ Event Streaming (Kafka, RabbitMQ)  
✔ Read-Optimized Databases (Data Warehouses like Snowflake, BigQuery, ClickHouse)

1️⃣ Approach: CQRS (Command Query Responsibility Segregation)

CQRS separates write (transactional) operations from read (analytical) operations.

🛠 Implementation in Java (Spring Boot)

✅ Order Service (Transactional OLTP) - Uses MySQL for Order Processing

java

CopyEdit

@Entity

public class Order {

@Id @GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String productName;

private Double price;

}

java

CopyEdit

@Service

public class OrderService {

@Autowired private OrderRepository orderRepository;

@Autowired private KafkaTemplate<String, Order> kafkaTemplate;

@Transactional

public Order createOrder(Order order) {

Order savedOrder = orderRepository.save(order);

// Publish event to Kafka for analytics processing

kafkaTemplate.send("order-events", savedOrder);

return savedOrder;

}

}

✅ Analytics Service (Analytical OLAP) - Uses NoSQL for Reports

java

CopyEdit

@Component

@KafkaListener(topics = "order-events", groupId = "analytics-group")

public class OrderAnalyticsConsumer {

@Autowired private MongoTemplate mongoTemplate;

public void processOrder(Order order) {

// Save processed data to MongoDB for fast analytical queries

mongoTemplate.save(order, "order\_analytics");

}

}

🔹 Transactional DB: MySQL (for fast writes)  
🔹 Analytical DB: MongoDB / ElasticSearch / Snowflake (for reporting)  
🔹 Event Streaming: Kafka decouples OLTP & OLAP

✅ Benefits:  
✔ No OLAP load on OLTP DB  
✔ Scalable analytics with real-time data  
✔ Prevents locking & bottlenecks

**====================================================================**

1. **Have you worked on event driven architecture and how to implement it.**

**pub sub model and explain how to communicate each other**

📌 What is Event-Driven Architecture (EDA)?

In EDA, microservices communicate by publishing and consuming events asynchronously rather than making direct REST API calls.

✅ When to use?  
✔ High-performance & low-latency applications  
✔ Real-time data processing (e.g., fraud detection, IoT, stock trading)  
✔ Microservices that need loose coupling

✅ Common Tech Stack

* Message Brokers: Kafka, RabbitMQ, AWS SQS/SNS
* Event Storage: Event Sourcing with PostgreSQL, MongoDB, Kafka Streams
* Processing Frameworks: Spring Boot, Apache Flink, Debezium

1️⃣ Implementing Event-Driven Microservices with Kafka (Spring Boot)

Scenario: E-commerce Order Processing

Imagine an Order Service that processes purchases and an Inventory Service that updates stock. Instead of a direct API call, the Order Service emits an "Order Created" event, which the Inventory Service listens to.

🛠 Step 1: Add Kafka Dependencies

xml

CopyEdit

<dependency>

<groupId>org.springframework.kafka</groupId>

<artifactId>spring-kafka</artifactId>

</dependency>

🛠 Step 2: Configure Kafka in application.yml

yaml

CopyEdit

spring:

kafka:

bootstrap-servers: localhost:9092

consumer:

group-id: inventory-group

auto-offset-reset: earliest

producer:

key-serializer: org.apache.kafka.common.serialization.StringSerializer

value-serializer: org.apache.kafka.common.serialization.StringSerializer

consumer:

key-deserializer: org.apache.kafka.common.serialization.StringDeserializer

value-deserializer: org.apache.kafka.common.serialization.StringDeserializer

🛠 Step 3: Publish Events from Order Service

java

CopyEdit

@Service

public class OrderService {

@Autowired private KafkaTemplate<String, String> kafkaTemplate;

public void placeOrder(String orderId) {

System.out.println("Order placed: " + orderId);

// Publish event

kafkaTemplate.send("order-events", orderId);

}

}

🔹 Event is published asynchronously instead of making an API call to Inventory Service.

🛠 Step 4: Consume Events in Inventory Service

java

CopyEdit

@Component

public class InventoryListener {

@KafkaListener(topics = "order-events", groupId = "inventory-group")

public void handleOrderEvent(String orderId) {

System.out.println("Updating inventory for order: " + orderId);

// Process inventory update

}

}

✅ Result:

* Order Service emits order-events.
* Inventory Service listens & updates stock asynchronously.
* No direct API call → Loose Coupling → Better Scalability! 🚀

**====================================================**

1. **Saga design pattern**

**📌 Saga Design Pattern in Microservices**

**🔹 What is the Saga Pattern?**

The **Saga Pattern** is a **distributed transaction management** technique for microservices where a long-running business transaction is broken into multiple **smaller transactions** that execute **in a sequence** or **in parallel**.

Since **distributed transactions** (like 2PC - Two-Phase Commit) are not scalable in microservices, the **Saga Pattern** ensures **eventual consistency** without locking resources.

**🔹 Types of Saga Pattern**

1️⃣ **Choreography-Based Saga**

* Each microservice **listens for events** and **reacts accordingly**.
* No central coordinator.
* Best for **simpler workflows**.

2️⃣ **Orchestration-Based Saga**

* A **central orchestrator** (Saga Coordinator) manages the workflow.
* Best for **complex transactions** with strict **order control**.

**1️⃣ Choreography-Based Saga (Event-Driven)**

**📌 Scenario: E-commerce Order Processing**

* **Order Service** creates an order 🛒
* **Payment Service** processes the payment 💳
* **Inventory Service** reserves stock 📦
* **Notification Service** sends confirmation 📩

**🛠 Step 1: Add Kafka Dependencies**

xml

CopyEdit

<dependency>

<groupId>org.springframework.kafka</groupId>

<artifactId>spring-kafka</artifactId>

</dependency>

**🛠 Step 2: Publish "Order Created" Event**

java

CopyEdit

@Service

public class OrderService {

@Autowired private KafkaTemplate<String, String> kafkaTemplate;

public void createOrder(String orderId) {

System.out.println("Order Created: " + orderId);

kafkaTemplate.send("order-events", orderId);

}

}

**🛠 Step 3: Payment Service Listens & Publishes "Payment Processed"**

java

CopyEdit

@Component

public class PaymentListener {

@KafkaListener(topics = "order-events", groupId = "payment-group")

public void processPayment(String orderId) {

System.out.println("Processing Payment for Order: " + orderId);

// Publish next event

kafkaTemplate.send("payment-events", orderId);

}

}

**🛠 Step 4: Inventory Service Listens & Publishes "Stock Reserved"**

java

CopyEdit

@Component

public class InventoryListener {

@KafkaListener(topics = "payment-events", groupId = "inventory-group")

public void reserveStock(String orderId) {

System.out.println("Stock Reserved for Order: " + orderId);

kafkaTemplate.send("inventory-events", orderId);

}

}

**🛠 Step 5: Notification Service Listens & Sends Confirmation**

java

CopyEdit

@Component

public class NotificationListener {

@KafkaListener(topics = "inventory-events", groupId = "notification-group")

public void sendNotification(String orderId) {

System.out.println("Order Confirmation Sent for: " + orderId);

}

}

✅ **Result:**  
✔ Services execute **independently** with **eventual consistency**  
✔ No **direct API calls**, reducing dependencies

**2️⃣ Orchestration-Based Saga (Central Coordinator)**

**📌 How It Works?**

* A **Saga Orchestrator** centrally **manages** transactions.
* Each microservice executes a step and **notifies** the orchestrator.
* If a step **fails**, the orchestrator triggers **compensating actions**.

**🛠 Step 1: Define Saga Orchestrator**

java

CopyEdit

@Service

public class SagaOrchestrator {

@Autowired private KafkaTemplate<String, String> kafkaTemplate;

public void startSaga(String orderId) {

System.out.println("Saga Started for Order: " + orderId);

kafkaTemplate.send("order-saga-events", "START\_ORDER:" + orderId);

}

}

**🛠 Step 2: Order Service Processes Order**

java

CopyEdit

@Component

public class OrderService {

@KafkaListener(topics = "order-saga-events", groupId = "order-group")

public void handleSagaEvent(String message) {

if (message.startsWith("START\_ORDER")) {

String orderId = message.split(":")[1];

System.out.println("Order Created: " + orderId);

// Notify next step

kafkaTemplate.send("order-saga-events", "PROCESS\_PAYMENT:" + orderId);

}

}

}

**🛠 Step 3: Payment Service Processes Payment**

java

CopyEdit

@Component

public class PaymentService {

@KafkaListener(topics = "order-saga-events", groupId = "payment-group")

public void handleSagaEvent(String message) {

if (message.startsWith("PROCESS\_PAYMENT")) {

String orderId = message.split(":")[1];

System.out.println("Payment Processed for Order: " + orderId);

// Notify next step

kafkaTemplate.send("order-saga-events", "RESERVE\_STOCK:" + orderId);

}

}

}

**🛠 Step 4: Inventory Service Reserves Stock**

java

CopyEdit

@Component

public class InventoryService {

@KafkaListener(topics = "order-saga-events", groupId = "inventory-group")

public void handleSagaEvent(String message) {

if (message.startsWith("RESERVE\_STOCK")) {

String orderId = message.split(":")[1];

System.out.println("Stock Reserved for Order: " + orderId);

// Notify next step

kafkaTemplate.send("order-saga-events", "SEND\_NOTIFICATION:" + orderId);

}

}

}

**🛠 Step 5: Notification Service Sends Confirmation**

java

CopyEdit

@Component

public class NotificationService {

@KafkaListener(topics = "order-saga-events", groupId = "notification-group")

public void handleSagaEvent(String message) {

if (message.startsWith("SEND\_NOTIFICATION")) {

String orderId = message.split(":")[1];

System.out.println("Order Confirmation Sent for: " + orderId);

}

}

}

**3️⃣ Handling Failures in Saga**

**🔹 Compensating Transactions**

If **Stock Reservation Fails**, the orchestrator can trigger a **rollback**:

java

CopyEdit

@Component

public class InventoryService {

@KafkaListener(topics = "order-saga-events", groupId = "inventory-group")

public void handleSagaEvent(String message) {

if (message.startsWith("RESERVE\_STOCK")) {

String orderId = message.split(":")[1];

if (checkStock(orderId)) {

kafkaTemplate.send("order-saga-events", "SEND\_NOTIFICATION:" + orderId);

} else {

kafkaTemplate.send("order-saga-events", "CANCEL\_PAYMENT:" + orderId);

}

}

}

}

**4️⃣ Choosing Between Choreography & Orchestration**

| **Feature** | **Choreography-Based Saga** | **Orchestration-Based Saga** |
| --- | --- | --- |
| **Complexity** | Simple (Event-Driven) | More Complex (Central Coordinator) |
| **Scalability** | High (Decentralized) | Moderate (Centralized) |
| **Flexibility** | Harder to change workflows | Easy to modify transactions |
| **Failure Handling** | Difficult (Each service handles failure) | Easier (Coordinator manages failures) |

**📌 Conclusion**

✅ Use **Choreography Saga** if microservices are **loosely coupled** & event-driven.  
✅ Use **Orchestration Saga** if you need **centralized transaction control**.

====================================================================

1. **Use of circuit breaker in microservices.**

**Circuit Breaker Pattern in Java Microservices**

The **Circuit Breaker Pattern** is used in **distributed systems** to prevent **cascading failures** by stopping repeated requests to a **failing service**. Instead of **constantly retrying**, the circuit breaker detects failures and **opens the circuit**, allowing the system to recover.

**🔹 Why Use a Circuit Breaker?**

In a microservices environment, one service might depend on many others. If one service **becomes slow or fails**, it can cause a **chain reaction**, slowing down the entire system.

✔ **Prevents excessive load** on a failing service  
✔ **Improves system resilience**  
✔ **Prevents thread exhaustion** by failing fast  
✔ **Allows automatic recovery** when the service becomes available

**🔹 How It Works?**

1. **CLOSED** (Normal State)
   * Requests go through **as usual**.
   * Failures are monitored.
2. **OPEN** (Service is Failing)
   * Requests **fail immediately** for a certain time.
   * The system avoids overloading the failing service.
3. **HALF-OPEN** (Check Recovery)
   * Some requests are **allowed through**.
   * If they **succeed**, the circuit **closes**.
   * If they **fail**, it stays **open**.

**🔹 Implementation Using Resilience4j**

Resilience4j is a Java library that provides an easy way to implement **Circuit Breaker**, **Rate Limiting**, **Retry**, and more.

**📌 Step 1: Add Dependencies**

Add the following to your pom.xml:

xml

CopyEdit

<dependency>

<groupId>io.github.resilience4j</groupId>

<artifactId>resilience4j-spring-boot2</artifactId>

<version>1.7.1</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-aop</artifactId>

</dependency>

**📌 Step 2: Configure Circuit Breaker in application.yml**

yaml

CopyEdit

resilience4j.circuitbreaker:

instances:

myServiceCircuitBreaker:

slidingWindowSize: 10

failureRateThreshold: 50

waitDurationInOpenState: 5000ms

permittedNumberOfCallsInHalfOpenState: 3

slowCallDurationThreshold: 2000ms

✔ **failureRateThreshold: 50** → Opens circuit if 50% of calls fail  
✔ **waitDurationInOpenState: 5000ms** → Circuit remains open for 5 seconds before checking recovery  
✔ **slowCallDurationThreshold: 2000ms** → If a call takes longer than 2 seconds, it's considered slow

**📌 Step 3: Implement Circuit Breaker in Service Layer**

java

CopyEdit

import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;

import org.springframework.stereotype.Service;

import org.springframework.web.client.RestTemplate;

@Service

public class MyService {

private final RestTemplate restTemplate = new RestTemplate();

@CircuitBreaker(name = "myServiceCircuitBreaker", fallbackMethod = "fallbackResponse")

public String callExternalService() {

System.out.println("Calling external service...");

return restTemplate.getForObject("http://example.com/api", String.class);

}

// Fallback method when circuit is open or service fails

public String fallbackResponse(Exception ex) {

return "Fallback Response: Service is currently unavailable!";

}

}

✔ **@CircuitBreaker(name = "myServiceCircuitBreaker", fallbackMethod = "fallbackResponse")**

* If the **external service fails**, it calls fallbackResponse().

✔ **fallbackResponse()**

* This method **prevents failure propagation** by returning a default response.

**📌 Step 4: Call the Service in a Controller**

java

CopyEdit

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/api")

public class MyController {

private final MyService myService;

public MyController(MyService myService) {

this.myService = myService;

}

@GetMapping("/fetch")

public String fetchData() {

return myService.callExternalService();

}

}

**🔹 Real-World Use Case: Payment Service**

**🛒 Scenario:**

* A **Payment Service** calls an **external banking API**.
* If the **banking API fails**, the service should **not hang indefinitely**.
* Instead, it should **fail fast** and **return a fallback response**.

**📌 Payment Service with Circuit Breaker**

java

CopyEdit

@Service

public class PaymentService {

private final RestTemplate restTemplate = new RestTemplate();

@CircuitBreaker(name = "paymentCircuitBreaker", fallbackMethod = "fallbackPayment")

public String processPayment() {

System.out.println("Processing Payment...");

return restTemplate.getForObject("http://banking-api.com/pay", String.class);

}

public String fallbackPayment(Exception ex) {

return "Payment service is currently unavailable. Please try again later.";

}

}

✔ If the **banking API is down**, fallbackPayment() will return a **graceful error message** instead of hanging indefinitely.

**🔹 Testing Circuit Breaker**

1️⃣ **Run the application**  
2️⃣ **Hit the endpoint multiple times:**

bash

CopyEdit

http://localhost:8080/api/fetch

3️⃣ **Simulate failure by stopping the external API**  
4️⃣ **Circuit Breaker should open** and return:

csharp

CopyEdit

Fallback Response: Service is currently unavailable!

**🔹 Monitoring Circuit Breaker Status**

You can expose Circuit Breaker metrics using **Spring Boot Actuator**:

**📌 Add Actuator Dependency**

xml

CopyEdit

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

**📌 Enable Endpoints in application.yml**

yaml

CopyEdit

management:

endpoints:

web:

exposure:

include: "health,metrics"

**📌 Check Circuit Breaker Status**

bash

CopyEdit

http://localhost:8080/actuator/health

**🔹 Circuit Breaker vs Retry**

| **Feature** | **Circuit Breaker** | **Retry** |
| --- | --- | --- |
| **Goal** | Prevent cascading failures | Retry failed requests |
| **Action** | Stops requests temporarily | Keeps retrying for a time |
| **When Used?** | Service is **completely down** | Temporary **network issues** |
| **State** | **Open, Closed, Half-Open** | No state tracking |

✔ **Use Circuit Breaker** when a **service is down** for an extended period.  
✔ **Use Retry** when **temporary failures** (e.g., slow response, timeout) occur.

**📌 Conclusion**

The **Circuit Breaker Pattern** helps **increase microservices resilience** by preventing cascading failures. **Resilience4j** makes it easy to implement in Java Spring Boot applications.

✅ **Use Circuit Breaker for:**

* Preventing system overload when a service is failing
* Ensuring **high availability** in microservices
* Failing fast instead of hanging indefinitely

===========================================================================

1. **Marker interface and it's use, write the own marker interface code.**

Marker Interface in Java

A Marker Interface is an interface that does not contain any methods or fields. It is used to signal or mark a class for a specific behavior.

🔹 Why Use a Marker Interface?

* Provides metadata to JVM or frameworks about a class.
* Used by built-in Java APIs (e.g., Serializable, Cloneable).
* Helps in conditional processing (e.g., checking if an object implements a marker interface).

🔹 Built-in Marker Interfaces in Java

1️⃣ Serializable → Marks an object for serialization.  
2️⃣ Cloneable → Allows cloning of objects.  
3️⃣ Remote → Marks objects for remote method calls.

🔹 Creating a Custom Marker Interface

We can create our own marker interface to mark classes for specific functionality.

📌 Example: Custom Marker Interface for Logging

java

CopyEdit

// Marker Interface

public interface Loggable {

// No methods or fields

}

📌 Implementing the Marker Interface

java

CopyEdit

public class User implements Loggable {

private String name;

private int age;

public User(String name, int age) {

this.name = name;

this.age = age;

}

}

📌 Checking if an Object is "Marked"

We can use instanceof to check if a class implements our marker interface.

java

CopyEdit

public class MarkerInterfaceExample {

public static void main(String[] args) {

User user = new User("Alice", 25);

logObject(user);

}

public static void logObject(Object obj) {

if (obj instanceof Loggable) {

System.out.println("Logging object: " + obj.getClass().getSimpleName());

} else {

System.out.println("This object cannot be logged.");

}

}

}

🔹 Output

pgsql

CopyEdit

Logging object: User

If we pass an object that does not implement Loggable, it will print:

pgsql

CopyEdit

This object cannot be logged.

**===========================================================================**

1. **Producer consumer program**

Here’s the **easiest Producer-Consumer program** using BlockingQueue from **Java’s concurrent package**, which automatically handles synchronization.

**🔹 Simplified Producer-Consumer Using BlockingQueue**

java

CopyEdit

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.LinkedBlockingQueue;

// Producer Thread

class Producer implements Runnable {

private final BlockingQueue<Integer> queue;

public Producer(BlockingQueue<Integer> queue) {

this.queue = queue;

}

@Override

public void run() {

int value = 1;

while (true) {

try {

System.out.println("Produced: " + value);

queue.put(value++); // Automatically waits if queue is full

Thread.sleep(1000); // Simulate delay

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

// Consumer Thread

class Consumer implements Runnable {

private final BlockingQueue<Integer> queue;

public Consumer(BlockingQueue<Integer> queue) {

this.queue = queue;

}

@Override

public void run() {

while (true) {

try {

int value = queue.take(); // Automatically waits if queue is empty

System.out.println("Consumed: " + value);

Thread.sleep(1500); // Simulate delay

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

// Main class

public class SimpleProducerConsumer {

public static void main(String[] args) {

BlockingQueue<Integer> queue = new LinkedBlockingQueue<>(5);

Thread producerThread = new Thread(new Producer(queue));

Thread consumerThread = new Thread(new Consumer(queue));

producerThread.start();

consumerThread.start();

}

}

**🔹 Output Example**

makefile

CopyEdit

Produced: 1

Consumed: 1

Produced: 2

Produced: 3

Consumed: 2

Produced: 4

Consumed: 3

**🔹 Why is this Easier?**

✅ **No need for wait() and notify()** – handled internally.  
✅ **Thread-safe** – BlockingQueue manages synchronization.  
✅ **Cleaner code** – avoids manual condition checks.

=====================================================================

1. **Difference between synchronized block and concurrent Hash map.**

Difference Between synchronized Block and ConcurrentHashMap in Java

| Feature | synchronized Block | ConcurrentHashMap |
| --- | --- | --- |
| Definition | Ensures that only one thread can access a block of code at a time. | A thread-safe version of HashMap that allows multiple threads to read and write efficiently. |
| Locking Mechanism | Locks the entire block of code or object. | Uses segment-based locking (locks only small portions). |
| Concurrency | Only one thread can access at a time, reducing performance. | Multiple threads can access different segments simultaneously. |
| Performance | Slower due to full locking. | Faster due to fine-grained locking. |
| Use Case | Best for synchronizing a small block of code or a critical section. | Best for highly concurrent environments with frequent reads/writes. |
| Example Usage | Used for protecting small critical sections. | Used in multi-threaded applications with frequent map operations. |

🔹 Example of synchronized Block

Locks the whole block, allowing only one thread at a time.

java

CopyEdit

class SharedData {

private int count = 0;

public void increment() {

synchronized (this) { // Only one thread at a time

count++;

System.out.println("Count: " + count);

}

}

}

🔹 Example of ConcurrentHashMap

Allows multiple threads to access different segments without blocking each other.

java

CopyEdit

import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapExample {

public static void main(String[] args) {

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

map.put(1, "Alice");

map.put(2, "Bob");

System.out.println(map.get(1)); // Output: Alice

}

}

🔹 Key Takeaways

✔ Use synchronized block for small critical sections.  
✔ Use ConcurrentHashMap when you need a thread-safe map with high performance.

==========================================================

1. **What is collision in HashMap and how to resolved**

**What is Collision in HashMap?**

A **collision** in a HashMap occurs when **two different keys** generate the **same hash code** and get assigned to the same bucket (i.e., same index in the internal array).

Since a **bucket** can store only one value, we need a way to handle multiple values in the same bucket.

**🔹 How Collisions Happen**

**1️⃣ Hashing Mechanism in HashMap**

* HashMap uses **hash codes** (generated by hashCode()) to determine the index where the key-value pair should be stored.
* If **two keys have the same hash index**, a **collision** occurs.

**2️⃣ Example of Collision**

java

CopyEdit

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

map.put(1, "Alice");

map.put(17, "Bob"); // Suppose both 1 and 17 generate the same hash index

System.out.println(map.get(1)); // Alice

System.out.println(map.get(17)); // Bob

Here, if **1 and 17 map to the same index**, a **collision** occurs.

**🔹 Collision Resolution Strategies in HashMap**

**1️⃣ Separate Chaining (Linked List) - Used in Java 7**

* Each bucket stores a **linked list** of key-value pairs.
* If a collision happens, the new entry is **appended** to the linked list.
* When retrieving a value, Java iterates through the list.

🔹 **Example (Simplified Representation)**

less

CopyEdit

Bucket[5] → (1, "Alice") → (17, "Bob") → (33, "Charlie")

🔹 **Drawback**

* Performance degrades when too many collisions occur (**O(n) lookup time** in the worst case).

**2️⃣ Separate Chaining with Balanced Tree (Java 8 and later)**

* If the linked list **length exceeds 8**, it is converted into a **Red-Black Tree** for faster lookups (**O(log n) time** instead of **O(n)**).

🔹 **Example (Tree Structure After 8+ Elements)**

rust

CopyEdit

Bucket[5]

├── (1, "Alice")

├── (17, "Bob")

└── (33, "Charlie")

(Stored as a balanced tree for efficiency)

**3️⃣ Open Addressing (Not Used in HashMap, Used in HashTable)**

* Instead of using a **linked list/tree**, it **finds the next available slot** in the array.
* Methods include:
  + **Linear Probing** (check next slot)
  + **Quadratic Probing** (check next, then next²)
  + **Double Hashing** (use another hash function to find a slot)

🔹 **Example of Linear Probing**

scss

CopyEdit

Bucket[5] → Occupied

Bucket[6] → Empty (Insert here)

🚀 **HashMap does NOT use this technique, but Hashtable and some other data structures do.**

**🔹 How to Avoid Collisions?**

✔ **Use a good hash function** (hashCode() should distribute keys evenly).  
✔ **Increase HashMap capacity** (more buckets reduce collisions).  
✔ **Use immutable and unique keys** (like String or UUID).

🚀 **Would you like a custom hashCode() implementation to minimize collisions?**

===========================================================================

1. **What is intermediate operation on java 8?(filter,map,flatmap,sorted,limit,skip,peek etc)**

**Intermediate Operations in Java 8 Streams**

In **Java 8 Streams**, **Intermediate Operations** are **lazy operations** that **process** the stream elements but **do not execute immediately**. They return a new **stream**, allowing further processing.

**🔹 List of Intermediate Operations**

| **Operation** | **Description** | **Example** |
| --- | --- | --- |
| filter(Predicate<T>) | Selects elements based on a condition | stream.filter(x -> x > 10) |
| map(Function<T, R>) | Transforms each element | stream.map(x -> x \* 2) |
| flatMap(Function<T, Stream<R>>) | Flattens multiple streams into one | stream.flatMap(List::stream) |
| sorted(Comparator<T>) | Sorts elements | stream.sorted() |
| limit(n) | Limits stream size to n | stream.limit(5) |
| skip(n) | Skips first n elements | stream.skip(3) |
| peek(Consumer<T>) | Performs an action without modifying elements | stream.peek(System.out::println) |
| distinct() | Removes duplicates | stream.distinct() |

**🔹 Intermediate Operations with Examples**

**1️⃣ filter() – Filter Elements**

java

CopyEdit

import java.util.List;

import java.util.stream.Collectors;

public class FilterExample {

public static void main(String[] args) {

List<Integer> numbers = List.of(10, 20, 30, 5, 15);

List<Integer> filtered = numbers.stream()

.filter(n -> n > 10) // Keep numbers > 10

.collect(Collectors.toList());

System.out.println(filtered); // Output: [20, 30, 15]

}

}

**2️⃣ map() – Transform Elements**

java

CopyEdit

import java.util.List;

import java.util.stream.Collectors;

public class MapExample {

public static void main(String[] args) {

List<String> names = List.of("Alice", "Bob", "Charlie");

List<Integer> nameLengths = names.stream()

.map(String::length) // Convert names to their lengths

.collect(Collectors.toList());

System.out.println(nameLengths); // Output: [5, 3, 7]

}

}

**3️⃣ flatMap() – Flatten Nested Collections**

java

CopyEdit

import java.util.List;

import java.util.stream.Collectors;

public class FlatMapExample {

public static void main(String[] args) {

List<List<String>> nestedList = List.of(

List.of("Apple", "Banana"),

List.of("Orange", "Grapes")

);

List<String> flattened = nestedList.stream()

.flatMap(List::stream) // Flatten nested lists

.collect(Collectors.toList());

System.out.println(flattened); // Output: [Apple, Banana, Orange, Grapes]

}

}

**4️⃣ sorted() – Sort Elements**

java

CopyEdit

import java.util.List;

import java.util.stream.Collectors;

public class SortedExample {

public static void main(String[] args) {

List<Integer> numbers = List.of(5, 2, 8, 1);

List<Integer> sortedNumbers = numbers.stream()

.sorted() // Sort in ascending order

.collect(Collectors.toList());

System.out.println(sortedNumbers); // Output: [1, 2, 5, 8]

}

}

**5️⃣ limit() – Get First n Elements**

java

CopyEdit

import java.util.List;

import java.util.stream.Collectors;

public class LimitExample {

public static void main(String[] args) {

List<Integer> numbers = List.of(10, 20, 30, 40, 50);

List<Integer> limited = numbers.stream()

.limit(3) // Take first 3 elements

.collect(Collectors.toList());

System.out.println(limited); // Output: [10, 20, 30]

}

}

**6️⃣ skip() – Skip First n Elements**

java

CopyEdit

import java.util.List;

import java.util.stream.Collectors;

public class SkipExample {

public static void main(String[] args) {

List<Integer> numbers = List.of(10, 20, 30, 40, 50);

List<Integer> skipped = numbers.stream()

.skip(2) // Skip first 2 elements

.collect(Collectors.toList());

System.out.println(skipped); // Output: [30, 40, 50]

}

}

**7️⃣ peek() – Debug Stream Elements**

java

CopyEdit

import java.util.List;

import java.util.stream.Collectors;

public class PeekExample {

public static void main(String[] args) {

List<Integer> numbers = List.of(1, 2, 3, 4);

List<Integer> result = numbers.stream()

.peek(n -> System.out.println("Processing: " + n))

.map(n -> n \* 2)

.collect(Collectors.toList());

System.out.println(result); // Output: Processing: 1, Processing: 2, Processing: 3, Processing: 4

}

}

**🔹 Key Takeaways**

✔ **Intermediate operations are "lazy"** – they execute only when a terminal operation (collect(), forEach(), reduce()) is called.  
✔ **Used for filtering, transforming, sorting, limiting, skipping, and debugging streams**.  
✔ **Chained together** to form a powerful data processing pipeline.

===========================================================================

1. **Flow of micro service application.**

**What are the diffrent component of microservices?**

**Service discovery**

**load balancers.**

**APi Gateways.**

**Service Registries.**

**Circuit breakers.**

**Service monitoring.**

**Service orchestrosation.**

**External Configurations.**

**Flow of a Microservices Application**

A **microservices-based application** follows a well-structured flow for handling requests, processing data, and communicating between services. Below is a step-by-step breakdown of the flow in a **real-world microservices application**.

**🔹 Microservices Flow Diagram**

plaintext

CopyEdit

Client (Web/Mobile) → API Gateway → Service Discovery → Load Balancer → Microservices

↘ Authentication & Authorization (Security Layer)

↘ Circuit Breaker (Resilience Layer)

↘ Communication (REST, gRPC, Messaging)

↘ Database & External APIs

↘ Monitoring & Logging

**🔹 Step-by-Step Flow of a Microservices Application**

**1️⃣ Client Request Initiation**

* A **client (Web App, Mobile App, or External System)** makes a request to the **API Gateway**.
* Example: A user wants to fetch their order details from an **E-commerce System**.

📌 **Example Request**

nginx

CopyEdit

GET https://api.myshop.com/orders/123

**2️⃣ API Gateway**

* The **API Gateway** acts as a **single entry point** for all client requests.
* It **routes, authenticates, and secures** requests.
* Performs **rate limiting, caching, and request transformation**.

📌 **Example API Gateway Routes (Spring Cloud Gateway)**

yaml

CopyEdit

spring:

cloud:

gateway:

routes:

- id: order-service

uri: lb://ORDER-SERVICE

predicates:

- Path=/orders/\*\*

📌 **How it Works?**

* The **Gateway** identifies the request **/orders/123**.
* It **forwards the request** to the **Order Service**.

**3️⃣ Authentication & Authorization (Security Layer)**

* **OAuth2 / JWT (JSON Web Token)** is used to verify user identity.
* The **API Gateway checks the token** before forwarding requests.

📌 **JWT Authentication Example in Spring Boot**

java

CopyEdit

public boolean validateToken(String token) {

return Jwts.parser().setSigningKey(secretKey)

.parseClaimsJws(token).getBody().getExpiration().after(new Date());

}

✅ *Ensures only authorized users can access services.*

**4️⃣ Service Discovery (Eureka)**

* The **Gateway** queries the **Service Registry (Eureka, Consul)** to find the **Order Service**.
* If multiple instances exist, **Load Balancer** distributes the request.

📌 **Example Eureka Service Registration**

yaml

CopyEdit

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

✅ *Dynamically locates available services instead of using hardcoded URLs.*

**5️⃣ Load Balancing**

* If multiple instances of ORDER-SERVICE are running, the **Load Balancer** distributes the requests.
* **Client-side (Ribbon) or Server-side (NGINX, Kubernetes Ingress) Load Balancing** is used.

📌 **Load Balancing with RestTemplate**

java

CopyEdit

@LoadBalanced

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

✅ *Ensures efficient resource utilization and high availability.*

**6️⃣ Order Service (Business Logic Layer)**

* The **Order Service** fetches data from its **own database**.
* Uses **JPA/Hibernate** for data access.

📌 **Example Order Service Code**

java

CopyEdit

@RestController

@RequestMapping("/orders")

public class OrderController {

@Autowired

private OrderService orderService;

@GetMapping("/{id}")

public Order getOrder(@PathVariable Long id) {

return orderService.getOrderById(id);

}

}

📌 **Example Data Access Code**

java

CopyEdit

@Repository

public interface OrderRepository extends JpaRepository<Order, Long> {}

✅ *Encapsulates business logic and interacts with the database.*

**7️⃣ Inter-Service Communication**

🔹 **Two Methods**: 1️⃣ **Synchronous (REST, gRPC)** → Direct API calls between services.  
2️⃣ **Asynchronous (Kafka, RabbitMQ, AWS SQS)** → Event-driven architecture.

📌 **Example of REST Communication (Calling Payment Service)**

java

CopyEdit

public PaymentResponse getPayment(Long orderId) {

return restTemplate.getForObject("http://PAYMENT-SERVICE/payments/" + orderId, PaymentResponse.class);

}

📌 **Example of Event Communication (Using Kafka)**

java

CopyEdit

@KafkaListener(topics = "order-events", groupId = "order-group")

public void processOrderEvent(String orderEvent) {

System.out.println("Received order event: " + orderEvent);

}

✅ *Reduces tight coupling and improves scalability.*

**8️⃣ Circuit Breaker (Resilience Layer)**

* If PAYMENT-SERVICE is **down**, the Circuit Breaker prevents failures from spreading.

📌 **Using Resilience4j Circuit Breaker**

java

CopyEdit

@CircuitBreaker(name = "paymentService", fallbackMethod = "fallbackResponse")

public String processPayment(Long orderId) {

return restTemplate.getForObject("http://PAYMENT-SERVICE/pay/" + orderId, String.class);

}

public String fallbackResponse(Long orderId, Exception e) {

return "Payment Service is Down! Try again later.";

}

✅ *Enhances fault tolerance and system resilience.*

**9️⃣ Database Access**

* Each microservice has **its own database** (**Database Per Service Pattern**).
* **Order Service** connects to **MySQL**, **Payment Service** connects to **MongoDB**.

📌 **Example MySQL Configuration**

yaml

CopyEdit

spring:

datasource:

url: jdbc:mysql://localhost:3306/orders\_db

username: root

password: password

✅ *Ensures data isolation and prevents service failures from affecting others.*

**🔟 External API Calls**

* If required, the service **calls an external API** (e.g., Fetch delivery status from a third-party courier).
* Uses **FeignClient** for simplified communication.

📌 **Feign Client Example**

java

CopyEdit

@FeignClient(name = "delivery-service", url = "https://api.courier.com")

public interface DeliveryClient {

@GetMapping("/track/{orderId}")

String trackOrder(@PathVariable Long orderId);

}

✅ *Provides easy-to-use REST API calls between services.*

**1️⃣1️⃣ Monitoring & Logging**

* **Prometheus + Grafana** → **Performance Monitoring**
* **Spring Boot Actuator** → **Health Checks**
* **ELK Stack (Elasticsearch, Logstash, Kibana)** → **Centralized Logging**
* **Jaeger / Zipkin** → **Distributed Tracing**

📌 **Enable Actuator for Health Monitoring**

yaml

CopyEdit

management:

endpoints:

web:

exposure:

include: "health,metrics"

📌 **Access Health Check**

bash

CopyEdit

http://localhost:8080/actuator/health

✅ *Ensures system observability and troubleshooting.*

**🔹 End-to-End Flow Recap**

1️⃣ **Client** sends request to **API Gateway**.  
2️⃣ **API Gateway** forwards request to respective microservice via **Service Discovery**.  
3️⃣ **Authentication & Authorization** ensure security.  
4️⃣ **Load Balancer** distributes traffic across multiple instances.  
5️⃣ **Microservice processes request** and fetches data from its **database**.  
6️⃣ If another service is required, **inter-service communication** happens (REST/Kafka).  
7️⃣ If a service fails, **Circuit Breaker** prevents system-wide failure.  
8️⃣ **External API Calls** (if needed).  
9️⃣ Logs and metrics are collected by **Monitoring & Logging tools**.  
🔟 Response is sent back to the client.

**🔹 Conclusion**

✔ **Microservices follow a structured flow from API Gateway to database interactions.**  
✔ **Each component (Service Discovery, Load Balancing, Circuit Breaker, Monitoring) plays a vital role.**  
✔ **Using tools like Spring Boot, Eureka, Kafka, and Prometheus, we can build robust microservices.**

===================================================================

1. **WAP to sort and remove the duplicate char from below list using java8 stream api.**

Here’s a **Java 8 Stream API** program to **sort and remove duplicate characters** from a given list:

**Program**

java

CopyEdit

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class UniqueSortedChars {

public static void main(String[] args) {

List<Character> charList = Arrays.asList('b', 'a', 'd', 'c', 'a', 'b', 'e', 'd');

// Using Java 8 Stream API to remove duplicates and sort the characters

List<Character> sortedUniqueChars = charList.stream()

.distinct() // Remove duplicates

.sorted() // Sort in natural order

.collect(Collectors.toList());

System.out.println("Sorted Unique Characters: " + sortedUniqueChars);

}

}

**Output**

mathematica

CopyEdit

Sorted Unique Characters: [a, b, c, d, e]

**Explanation**

1. **.stream()** → Converts the list into a stream.
2. **.distinct()** → Removes duplicate elements.
3. **.sorted()** → Sorts elements in **ascending order**.
4. **.collect(Collectors.toList())** → Collects the results into a list.

=========================================================================

1. **Use of stream api how count each number count and put into HashMap**

**List<Integer> list={2,3,4,2,1,2,3,4,5};**

Here’s how you can use **Java 8 Stream API** to count the occurrences of each number in the list and store them in a HashMap:

**Program**

java

CopyEdit

import java.util.\*;

import java.util.function.Function;

import java.util.stream.Collectors;

public class NumberFrequency {

public static void main(String[] args) {

List<Integer> list = Arrays.asList(2, 3, 4, 2, 1, 2, 3, 4, 5);

// Using Stream API to count occurrences and store in a HashMap

Map<Integer, Long> frequencyMap = list.stream()

.collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));

// Print the result

System.out.println("Number Frequency Map: " + frequencyMap);

}

}

**Output**

javascript

CopyEdit

Number Frequency Map: {1=1, 2=3, 3=2, 4=2, 5=1}

**Explanation**

1. **.stream()** → Converts the list into a stream.
2. **.collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))**
   * groupingBy(Function.identity()) → Groups elements based on their values.
   * Collectors.counting() → Counts occurrences of each element.
3. Stores the result in a **HashMap**, where:
   * **Key** = Unique number from the list.
   * **Value** = Count of occurrences.

========================================================================

1. **How to handle exception in Spring using @ExceptionHandler and @ControllerAdvice.**

[**https://www.youtube.com/watch?v=IdHHwZg3v58**](https://www.youtube.com/watch?v=IdHHwZg3v58)

[**https://github.com/codesnippetjava/ecom/**](https://github.com/codesnippetjava/ecom/)

**Handling Exceptions in Spring Boot Using @ExceptionHandler & @ControllerAdvice**

Spring Boot provides **@ExceptionHandler** and **@ControllerAdvice** annotations to handle exceptions **globally and locally** in a clean and structured way.

**🔹 1. Using @ExceptionHandler (Local Exception Handling)**

@ExceptionHandler is used inside a specific **controller** to handle exceptions **locally** for that controller.

**Example: Handling Exception in a Controller**

java

CopyEdit

@RestController

@RequestMapping("/products")

public class ProductController {

@GetMapping("/{id}")

public String getProduct(@PathVariable int id) {

if (id <= 0) {

throw new ProductNotFoundException("Invalid product ID: " + id);

}

return "Product " + id;

}

// Local Exception Handling inside the same Controller

@ExceptionHandler(ProductNotFoundException.class)

public ResponseEntity<String> handleProductNotFound(ProductNotFoundException ex) {

return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT\_FOUND);

}

}

**Output (When ID is invalid)**

vbnet

CopyEdit

HTTP Status: 404 Not Found

Body: "Invalid product ID: -1"

**🔹 2. Using @ControllerAdvice (Global Exception Handling)**

* @ControllerAdvice is a **global** exception handler.
* It can handle exceptions across **multiple controllers**.

**Step 1: Create a Custom Exception Class**

java

CopyEdit

public class ProductNotFoundException extends RuntimeException {

public ProductNotFoundException(String message) {

super(message);

}

}

**Step 2: Create a Global Exception Handler Using @ControllerAdvice**

java

CopyEdit

@RestControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ProductNotFoundException.class)

public ResponseEntity<String> handleProductNotFound(ProductNotFoundException ex) {

return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT\_FOUND);

}

@ExceptionHandler(Exception.class) // Generic Exception Handler

public ResponseEntity<String> handleGeneralException(Exception ex) {

return new ResponseEntity<>("An error occurred: " + ex.getMessage(), HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

**🔹 3. Returning a Custom Error Response Object**

Instead of returning just a **String message**, we can return a **custom error response**.

**Step 1: Define an ErrorResponse Class**

java

CopyEdit

public class ErrorResponse {

private String message;

private int status;

public ErrorResponse(String message, int status) {

this.message = message;

this.status = status;

}

// Getters & Setters

public String getMessage() { return message; }

public void setMessage(String message) { this.message = message; }

public int getStatus() { return status; }

public void setStatus(int status) { this.status = status; }

}

**Step 2: Modify GlobalExceptionHandler to Use ErrorResponse**

java

CopyEdit

@RestControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ProductNotFoundException.class)

public ResponseEntity<ErrorResponse> handleProductNotFound(ProductNotFoundException ex) {

ErrorResponse errorResponse = new ErrorResponse(ex.getMessage(), HttpStatus.NOT\_FOUND.value());

return new ResponseEntity<>(errorResponse, HttpStatus.NOT\_FOUND);

}

@ExceptionHandler(Exception.class)

public ResponseEntity<ErrorResponse> handleGeneralException(Exception ex) {

ErrorResponse errorResponse = new ErrorResponse("Something went wrong", HttpStatus.INTERNAL\_SERVER\_ERROR.value());

return new ResponseEntity<>(errorResponse, HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

**Output (When ID is Invalid)**

css

CopyEdit

HTTP Status: 404 Not Found

{

"message": "Invalid product ID: -1",

"status": 404

}

**🔹 4. Summary**

| **Annotation** | **Scope** | **Use Case** |
| --- | --- | --- |
| @ExceptionHandler | Local (Specific Controller) | Handle exceptions for a single controller. |
| @ControllerAdvice | Global (All Controllers) | Handle exceptions across multiple controllers. |
| @RestControllerAdvice | Global (REST API) | Same as @ControllerAdvice but for REST APIs. |

✅ **Best Practices** ✔ Always use @ControllerAdvice for **global exception handling**.  
✔ Use @ExceptionHandler inside a controller **only when necessary**.  
✔ Return a **custom error response** instead of plain text.  
✔ Log exceptions using **SLF4J/Logger** for debugging.

===========================================================================

1. **Suppose we have one Interface ABC and it's implemented by three class, then how to handle if three bean id is same?**

Handling Multiple Beans with the Same ID in Spring Boot

If you have an interface (ABC) implemented by three different classes and they all have the same bean ID, Spring will throw a BeanDefinitionOverrideException or a NoUniqueBeanDefinitionException when trying to inject them.

To handle this situation, you can use the following approaches:

Approach 1: Use @Qualifier to Specify the Bean Name

You can explicitly specify which bean to inject using @Qualifier.

Example

Step 1: Define an Interface

java

CopyEdit

public interface ABC {

void display();

}

Step 2: Implement the Interface in Three Different Classes

java

CopyEdit

import org.springframework.stereotype.Component;

@Component("beanA")

public class ClassA implements ABC {

public void display() {

System.out.println("ClassA Implementation");

}

}

@Component("beanB")

public class ClassB implements ABC {

public void display() {

System.out.println("ClassB Implementation");

}

}

@Component("beanC")

public class ClassC implements ABC {

public void display() {

System.out.println("ClassC Implementation");

}

}

* Here, we manually specify the bean name using @Component("beanName").

Step 3: Inject Using @Qualifier

java

CopyEdit

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.stereotype.Service;

@Service

public class ABCService {

@Autowired

@Qualifier("beanA") // Specify which bean to use

private ABC abc;

public void execute() {

abc.display();

}

}

✅ This will ensure that ClassA is injected instead of ClassB or ClassC.

Approach 2: Use @Primary for a Default Bean

If one implementation is the default, mark it with @Primary to avoid conflicts.

Example

java

CopyEdit

@Component

@Primary // This will be the default bean

public class ClassA implements ABC {

public void display() {

System.out.println("ClassA Implementation");

}

}

* Now, if multiple beans exist and no @Qualifier is provided, ClassA will be injected by default.

===========================================================================

1. **JVM architecture and it's explanation with example**

**JVM Architecture & Explanation with Example**

The **Java Virtual Machine (JVM)** is an engine that provides a runtime environment to execute Java programs. It is responsible for converting Java bytecode into machine code and managing memory.

**1️⃣ Components of JVM Architecture**

JVM consists of the following main components:

**1.1 ClassLoader Subsystem**

* **Loads Java classes into memory.**
* Follows the **delegation model**:
  1. **Bootstrap ClassLoader** → Loads Java core classes (rt.jar).
  2. **Extension ClassLoader** → Loads classes from ext directory.
  3. **Application ClassLoader** → Loads classes from the classpath.

**Example: How ClassLoader Works**

java

CopyEdit

public class ClassLoaderExample {

public static void main(String[] args) {

System.out.println(ClassLoaderExample.class.getClassLoader());

}

}

🛠 **Output**

bash

CopyEdit

sun.misc.Launcher$AppClassLoader@18b4aac2

**1.2 Runtime Data Areas**

The memory inside JVM is divided into different sections:

**1.2.1 Heap Memory (Used for Object Storage)**

* **Divided into:**
  1. **Young Generation** → Stores newly created objects.
     + **Eden Space** → All new objects are allocated here.
     + **Survivor Spaces (S0 & S1)** → Survive GC cycles before moving to Old Gen.
  2. **Old Generation (Tenured)** → Stores long-lived objects.

**1.2.2 Stack Memory (Used for Method Execution)**

* Each **thread** gets a separate **stack**.
* Stores **local variables, method calls, and references**.

**1.2.3 Metaspace (JDK 8+)**

* Stores **class metadata**.
* Dynamically **resizable** (unlike **PermGen in JDK 7**).

**1.2.4 PC Register (Program Counter)**

* Stores the **current instruction address**.

**1.2.5 Native Method Stack**

* Stores **native method calls** (e.g., JNI calls to C/C++).

**1.3 Execution Engine**

* Converts **bytecode into machine code**.
* Components:
  1. **Interpreter** → Executes bytecode line by line (slow).
  2. **JIT Compiler (Just-In-Time)** → Converts frequently used bytecode into native code (fast).
  3. **Garbage Collector (GC)** → Frees up unused memory.

**2️⃣ JVM Memory Diagram**

mathematica

CopyEdit

JVM Memory

├── Heap Memory

│ ├── Young Gen

│ │ ├── Eden Space

│ │ ├── Survivor Space (S0, S1)

│ ├── Old Gen (Tenured)

│

├── Stack Memory (Thread-local)

├── Metaspace (Class Metadata)

├── PC Register

└── Native Method Stack

**3️⃣ Example Program to Demonstrate JVM Components**

Let's write a **Java program** and analyze how JVM executes it.

java

CopyEdit

public class JVMExample {

static String staticVar = "Stored in Metaspace";

public static void main(String[] args) {

JVMExample obj1 = new JVMExample(); // Stored in Heap

obj1.display();

}

public void display() {

int localVar = 10; // Stored in Stack

System.out.println("Hello from JVM! Local Variable: " + localVar);

}

}

**4️⃣ How JVM Executes the Above Program**

1. **ClassLoader loads** JVMExample.class into memory.
2. **Memory allocation**:
   * **Heap** → Stores obj1 instance.
   * **Stack** → Stores localVar inside display().
   * **Metaspace** → Stores staticVar.
3. **Execution Engine (JIT)** compiles & executes bytecode.

🛠 **Output**

sql

CopyEdit

Hello from JVM! Local Variable: 10

**5️⃣ JVM Garbage Collection (GC)**

* **Minor GC** → Cleans Young Gen (Eden & Survivor Spaces).
* **Major GC (Full GC)** → Cleans Old Gen (Expensive!).
* **Types of GC in JVM**:
  1. **Serial GC** → Best for small applications (-XX:+UseSerialGC).
  2. **Parallel GC** → Uses multiple threads (-XX:+UseParallelGC).
  3. **G1 GC** → Optimized for large applications (-XX:+UseG1GC).

**6️⃣ JVM Tuning Parameters**

* Set **Initial Heap Size**

diff

CopyEdit

-Xms512m

* Set **Maximum Heap Size**

diff

CopyEdit

-Xmx2g

* Enable **Garbage Collector (G1GC)**

ruby

CopyEdit

-XX:+UseG1GC

**7️⃣ Summary**

| **JVM Component** | **Description** |
| --- | --- |
| **ClassLoader** | Loads Java classes into memory. |
| **Heap (Young + Old Gen)** | Stores objects. |
| **Stack** | Stores method calls & local variables. |
| **Metaspace** | Stores class metadata (JDK 8+). |
| **Execution Engine** | Runs bytecode (Interpreter + JIT). |
| **Garbage Collector (GC)** | Removes unused objects. |

===========================================================================

1. **what is @SpringBootApplication and tell me which two annotation combination is?**

**What is @SpringBootApplication?**

The @SpringBootApplication annotation is a **convenience annotation** in **Spring Boot** that enables **auto-configuration, component scanning, and configuration properties** in a single declaration.

It is used on the **main class** of a **Spring Boot application** and marks it as the primary Spring Boot configuration.

**Which Two Annotations Make Up @SpringBootApplication?**

Internally, @SpringBootApplication is a combination of **three annotations**, but the two most important ones are:

1. **@Configuration** → Marks the class as a **Spring configuration class**.
2. **@ComponentScan** → Enables **component scanning** to detect @Component, @Service, @Repository, and @Controller beans.
3. **@EnableAutoConfiguration** → Automatically configures Spring Boot components based on dependencies.

📌 **Key Point:**

* @SpringBootApplication is equivalent to using all three annotations manually.

**Example of @SpringBootApplication in Action**

java

CopyEdit

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class MyApp {

public static void main(String[] args) {

SpringApplication.run(MyApp.class, args);

}

}

🔹 **What Happens Here?**

* SpringApplication.run(MyApp.class, args); → Bootstraps the Spring Boot application.
* **Auto-configuration** loads default configurations (like an embedded Tomcat).
* **Component scanning** finds beans inside the package.

**What If We Replace @SpringBootApplication with Separate Annotations?**

Instead of:

java

CopyEdit

@SpringBootApplication

You can write:

java

CopyEdit

@Configuration

@EnableAutoConfiguration

@ComponentScan

public class MyApp {

public static void main(String[] args) {

SpringApplication.run(MyApp.class, args);

}

}

✅ **Both implementations work the same!**

**Summary**

| **Annotation** | **Purpose** |
| --- | --- |
| **@Configuration** | Marks the class as a Spring configuration class. |
| **@ComponentScan** | Automatically scans for Spring-managed components. |
| **@EnableAutoConfiguration** | Enables Spring Boot’s auto-configuration. |
| **@SpringBootApplication** | Combines all three annotations into one. |

=====================================================================

1. **How to implement caching in Spring Data**

**Implementing Caching in Spring Data**

**📌 Why Use Caching?**

**Caching improves performance by reducing redundant database queries and storing frequently accessed data in memory.**

**✅ Step-by-Step Implementation in Spring Boot**

**We’ll use Spring Boot, Spring Data JPA, and Spring Cache.**

**🔹 Step 1: Add Dependencies**

**In pom.xml, add the Spring Boot Cache and Cache Provider.**

**For In-Memory Caching (Simple Cache/ConcurrentMapCache)**

**xml**

**CopyEdit**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-cache</artifactId>**

**</dependency>**

**For Redis Cache**

**xml**

**CopyEdit**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-data-redis</artifactId>**

**</dependency>**

**🔹 Step 2: Enable Caching in Spring Boot**

**In the main application class, enable caching with @EnableCaching:**

**java**

**CopyEdit**

**import org.springframework.boot.SpringApplication;**

**import org.springframework.boot.autoconfigure.SpringBootApplication;**

**import org.springframework.cache.annotation.EnableCaching;**

**@SpringBootApplication**

**@EnableCaching // Enable Spring Caching**

**public class SpringCacheApplication {**

**public static void main(String[] args) {**

**SpringApplication.run(SpringCacheApplication.class, args);**

**}**

**}**

**🔹 Step 3: Create Entity and Repository**

**We'll use Spring Data JPA for database interaction.**

**📌 Create Product Entity**

**java**

**CopyEdit**

**import jakarta.persistence.Entity;**

**import jakarta.persistence.GeneratedValue;**

**import jakarta.persistence.GenerationType;**

**import jakarta.persistence.Id;**

**import java.io.Serializable;**

**@Entity**

**public class Product implements Serializable { // Serializable for Redis cache**

**@Id**

**@GeneratedValue(strategy = GenerationType.IDENTITY)**

**private Long id;**

**private String name;**

**private double price;**

**// Getters and Setters**

**}**

**📌 Create Repository**

**java**

**CopyEdit**

**import org.springframework.data.jpa.repository.JpaRepository;**

**import org.springframework.stereotype.Repository;**

**@Repository**

**public interface ProductRepository extends JpaRepository<Product, Long> {**

**}**

**🔹 Step 4: Implement Caching in Service Layer**

**📌 Using @Cacheable, @CachePut, @CacheEvict**

**java**

**CopyEdit**

**import org.springframework.beans.factory.annotation.Autowired;**

**import org.springframework.cache.annotation.CacheEvict;**

**import org.springframework.cache.annotation.CachePut;**

**import org.springframework.cache.annotation.Cacheable;**

**import org.springframework.stereotype.Service;**

**import java.util.Optional;**

**@Service**

**public class ProductService {**

**@Autowired**

**private ProductRepository productRepository;**

**// Fetch product from cache or database**

**@Cacheable(value = "products", key = "#id") // Cache the result**

**public Product getProductById(Long id) {**

**System.out.println("Fetching from DB...");**

**return productRepository.findById(id).orElse(null);**

**}**

**// Update product and refresh cache**

**@CachePut(value = "products", key = "#product.id") // Update cache**

**public Product updateProduct(Product product) {**

**return productRepository.save(product);**

**}**

**// Delete product and remove from cache**

**@CacheEvict(value = "products", key = "#id") // Remove from cache**

**public void deleteProduct(Long id) {**

**productRepository.deleteById(id);**

**}**

**}**

**🔹 Step 5: Configure Cache in application.properties**

**📌 For In-Memory Cache (Default)**

**properties**

**CopyEdit**

**spring.cache.type=simple**

**📌 For Redis Cache**

**properties**

**CopyEdit**

**spring.cache.type=redis**

**spring.redis.host=localhost**

**spring.redis.port=6379**

**🔹 Step 6: Testing the Cache Implementation**

**📌 Create a REST Controller**

**java**

**CopyEdit**

**import org.springframework.beans.factory.annotation.Autowired;**

**import org.springframework.web.bind.annotation.\*;**

**@RestController**

**@RequestMapping("/products")**

**public class ProductController {**

**@Autowired**

**private ProductService productService;**

**@GetMapping("/{id}")**

**public Product getProduct(@PathVariable Long id) {**

**return productService.getProductById(id);**

**}**

**@PostMapping("/")**

**public Product saveProduct(@RequestBody Product product) {**

**return productService.updateProduct(product);**

**}**

**@DeleteMapping("/{id}")**

**public void deleteProduct(@PathVariable Long id) {**

**productService.deleteProduct(id);**

**}**

**}**

**🔹 Step 7: Testing the Cache Behavior**

**Scenario 1: Fetch Data from DB and Cache**

**1️⃣ First Call → Fetches from DB and caches the result.  
2️⃣ Second Call → Fetches from cache (faster response).**

**bash**

**CopyEdit**

**GET http://localhost:8080/products/1**

**Console Output (First Call)**

**csharp**

**CopyEdit**

**Fetching from DB...**

**Console Output (Second Call)**

**(No DB log, fetched from cache)**

**Scenario 2: Updating Product (Cache Refresh)**

**1️⃣ Update a product → Cache is updated.  
2️⃣ Fetching again → Returns new value from cache.**

**bash**

**CopyEdit**

**PUT http://localhost:8080/products/**

**Body:**

**{**

**"id": 1,**

**"name": "Updated Product",**

**"price": 100.0**

**}**

**Scenario 3: Deleting Product (Cache Eviction)**

**1️⃣ Delete a product → Removes it from cache.  
2️⃣ Fetching again → Queries database again.**

**bash**

**CopyEdit**

**DELETE http://localhost:8080/products/1**

**🔹 Summary of Annotations**

| **Annotation** | **Description** |
| --- | --- |
| **@Cacheable(value = "cacheName", key = "#id")** | **Fetch from cache, if not found then fetch from DB and store in cache.** |
| **@CachePut(value = "cacheName", key = "#object.id")** | **Updates both database and cache.** |
| **@CacheEvict(value = "cacheName", key = "#id")** | **Deletes an entry from cache.** |
| **@EnableCaching** | **Enables caching in Spring Boot.** |

**🚀 Conclusion**

* **Spring Boot makes caching easy with @Cacheable, @CachePut, and @CacheEvict.**
* **Default cache (SimpleCache) is in-memory, but Redis is recommended for production.**
* **Caching reduces database load and improves performance.**

**==========================================================================**

1. **Java microservices:** [**https://www.youtube.com/watch?v=mPPhcU7oWDU&t=10862s**](https://www.youtube.com/watch?v=mPPhcU7oWDU&t=10862s)

**Source code:** [**https://github.com/SaiUpadhyayula/spring-boot-microservices**](https://github.com/SaiUpadhyayula/spring-boot-microservices)

**Master Event-Driven Microservices with Spring Boot, Cloud Functions, RabbitMQ & Kafka|4 Hours| Hindi**

[**https://www.youtube.com/watch?v=fd\_aS3ZSkxk**](https://www.youtube.com/watch?v=fd_aS3ZSkxk)

**SAGA Design pattern**[**: https://www.youtube.com/watch?v=WGI\_ciUa3FE**](:%20https:/www.youtube.com/watch?v=WGI_ciUa3FE)

[**https://github.com/JavaaTechSolutions/distributed-transaction/tree/main/delivery-ms**](https://github.com/JavaaTechSolutions/distributed-transaction/tree/main/delivery-ms)

**===============================================================**

1. **Spring Security.**

**Expalin spring security? How Spring security works?**

**Concept:**

**Authorization:**

**Authentication.**

**CSRF attack protection.**

**Session Management:**

**Password Encodding:**

**Integration with Oauth2/JWT:**

[**https://www.youtube.com/watch?v=RRH2o5WCBr4**](https://www.youtube.com/watch?v=RRH2o5WCBr4)

**Spring Security Explained: How It Works? 🚀**

Spring Security is a **powerful authentication and authorization framework** that helps secure Spring Boot applications by handling user authentication, access control, session management, and protection against security threats like **CSRF, XSS, SQL Injection, etc.**

**🔹 How Does Spring Security Work?**

Spring Security works as a **filter-based security framework**. When a request is made to a secured API, it passes through several security filters that handle **authentication and authorization** before reaching the application logic.

**🔹 Steps:** 1️⃣ **Client sends a request** (e.g., login or API access).  
2️⃣ **Spring Security intercepts the request** through filters.  
3️⃣ **Authentication:** Verifies user credentials (username/password, JWT, OAuth2).  
4️⃣ **Authorization:** Grants or denies access based on roles/permissions.  
5️⃣ **Session Management:** Maintains or invalidates user sessions.  
6️⃣ **Response sent back to the client** (success or failure).

**🌟 1. Authentication (Who Are You?)**

Authentication is the process of verifying a user’s identity before allowing access to an application.

**✔️ Types of Authentication in Spring Security**

* **Basic Authentication:** Uses a username & password.
* **Form-Based Authentication:** Uses login forms.
* **JWT Authentication:** Uses a secure token instead of sessions.
* **OAuth2 Authentication:** Uses third-party authentication providers (Google, Facebook, etc.).

**🔹 Example: Custom Authentication in Spring Boot**

java

CopyEdit

@Bean

public UserDetailsService userDetailsService() {

UserDetails user = User.withUsername("user")

.password(passwordEncoder().encode("password"))

.roles("USER")

.build();

UserDetails admin = User.withUsername("admin")

.password(passwordEncoder().encode("admin123"))

.roles("ADMIN")

.build();

return new InMemoryUserDetailsManager(user, admin);

}

**🌟 2. Authorization (What Can You Do?)**

Once a user is authenticated, **authorization** ensures they can only access permitted resources.

**✔️ Role-Based Access Control (RBAC)**

* **Admin Role**: Can access all APIs.
* **User Role**: Can only access specific APIs.

🔹 **Example: Securing Endpoints with Roles**

java

CopyEdit

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(auth -> auth

.requestMatchers("/admin/\*\*").hasRole("ADMIN") // Only Admins can access

.requestMatchers("/user/\*\*").hasRole("USER") // Only Users can access

.anyRequest().authenticated()

)

.formLogin()

.httpBasic();

return http.build();

}

**🌟 3. CSRF Attack Protection (Cross-Site Request Forgery)**

CSRF (Cross-Site Request Forgery) is an attack where a **malicious website tricks users** into executing unwanted actions on a different website where they are authenticated.

**🔹 Spring Security’s CSRF Protection**

By default, Spring Security enables **CSRF protection**, which requires a CSRF token for state-changing requests (POST, PUT, DELETE).

**Example: Disabling CSRF (For APIs, Not Recommended)**

java

CopyEdit

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.csrf(csrf -> csrf.disable()) // Not recommended for web applications

.authorizeHttpRequests(auth -> auth.anyRequest().authenticated())

.formLogin();

return http.build();

}

**✅ Best Practice:** Keep CSRF enabled for web applications but disable it for REST APIs.

**🌟 4. Session Management**

Spring Security provides various ways to manage user sessions.

**🔹 Types of Session Management**

✔️ **Stateless Session (JWT/OAuth2)** – No session stored in the backend.  
✔️ **Session Fixation Protection** – Prevents session hijacking.  
✔️ **Concurrent Session Control** – Limits the number of active sessions per user.

**Example: Stateless Session (JWT-Based)**

java

CopyEdit

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.sessionManagement(session -> session

.sessionCreationPolicy(SessionCreationPolicy.STATELESS)) // No server-side session

.authorizeHttpRequests(auth -> auth.anyRequest().authenticated())

.httpBasic();

return http.build();

}

**🌟 5. Password Encoding**

Storing plain-text passwords is a **huge security risk**! 🔥  
Spring Security provides **BCryptPasswordEncoder** to hash passwords before storing them in the database.

**Example: Hashing a Password in Spring Boot**

java

CopyEdit

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

🔹 **BCrypt** automatically generates a **random salt** every time, making it very secure.

**🌟 6. Integration with OAuth2 & JWT**

Spring Security **easily integrates** with modern authentication protocols like **OAuth2 and JWT**.

**🔹 OAuth2 (Third-Party Authentication)**

OAuth2 allows users to log in using external providers like **Google, Facebook, GitHub, etc.**

**Example: Adding OAuth2 Login with Google**

xml

CopyEdit

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-oauth2-client</artifactId>

</dependency>

java

CopyEdit

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(auth -> auth.anyRequest().authenticated())

.oauth2Login(); // Enables Google, Facebook, GitHub Login

return http.build();

}

**🔹 JWT (JSON Web Token) Authentication**

Instead of maintaining sessions, JWT is a **stateless** authentication mechanism where the server issues a token, and the client sends it in **every request**.

**Steps for JWT Authentication:** 1️⃣ User logs in and gets a **JWT token**.  
2️⃣ User sends the token in **Authorization Header** in every request.  
3️⃣ Server validates the token before granting access.

**Example: Generating a JWT Token**

java

CopyEdit

public String generateToken(String username) {

return Jwts.builder()

.setSubject(username)

.setIssuedAt(new Date())

.setExpiration(new Date(System.currentTimeMillis() + 1000 \* 60 \* 60)) // 1 hour expiry

.signWith(SignatureAlgorithm.HS256, "mySecretKey")

.compact();

}

**Example: Sending Token in API Request**

bash

CopyEdit

curl -H "Authorization: Bearer eyJhbGciOi..." http://localhost:8080/api/secure

**🔹 Summary of Key Concepts**

| **Concept** | **Description** |
| --- | --- |
| **Authentication** | Verifies who the user is (Login) |
| **Authorization** | Controls what the user can access (Roles & Permissions) |
| **CSRF Protection** | Prevents malicious requests from being executed |
| **Session Management** | Controls user sessions (Stateful or Stateless JWT) |
| **Password Encoding** | Uses **BCrypt** to hash passwords securely |
| **OAuth2 Integration** | Supports third-party login providers (Google, Facebook) |
| **JWT Integration** | Uses **stateless authentication** with signed tokens |

==========================================================================

1. **When new feature to be added by client in system, how we can estimated and come up with efforts-develeopement, testing and delivering to client.**

**Effort Estimation for a New Feature in a System**

Estimating the effort required for **development, testing, and delivery** of a new feature involves multiple steps. The process ensures that the feature is delivered **on time, within scope, and with high quality**.

**🔹 1. Understand the Feature Requirements**

**✔️ Steps:**

1️⃣ **Client Discussion:**

* Understand the feature's purpose, business logic, and dependencies.
* Ask **clarifying questions**:
  + What problem does this feature solve?
  + Who will use it?
  + Any specific performance/security expectations?

2️⃣ **Break Down the Feature into Subtasks**

* Identify different modules affected.
* Check for UI/Backend/API changes.
* Identify any **database changes, security considerations, or third-party integrations**.

3️⃣ **Define Acceptance Criteria**

* List clear criteria that determine **when the feature is “done”**.
* Example: “User should be able to reset the password via email OTP.”

**🔹 2. Estimate Efforts Using WBS (Work Breakdown Structure)**

**✔️ Steps:**

Break the work into smaller **manageable tasks** and estimate time for each.

| **Phase** | **Task** | **Effort (in days)** |
| --- | --- | --- |
| **Requirement Analysis** | Finalize feature scope, acceptance criteria | 1 |
| **Design & Planning** | API contracts, database schema, UI wireframes | 2 |
| **Development** | Backend implementation | 4 |
|  | Frontend development | 3 |
|  | Integration with existing modules | 2 |
| **Testing** | Unit Testing | 2 |
|  | API Testing (Postman, RestAssured) | 2 |
|  | UI Testing | 2 |
|  | Performance/Security Testing | 2 |
| **Code Review & Fixes** | Peer review and fixes | 1 |
| **Deployment & UAT** | Deploy on staging & collect client feedback | 2 |
| **Production Release** | Deploy to production & monitor | 1 |
| **Total Estimated Time** | **Sum of all efforts** | **~22 days** |

✔️ Add a **buffer (10-15%)** for unforeseen challenges, dependency issues, or rework.

**🔹 3. Consider Development & Testing Approach**

* **Agile Development (Sprint Planning)** – Break tasks into **user stories** and plan in **2-week sprints**.
* **Parallel Testing & Dev** – Start **API testing while UI is being developed** to save time.
* **Automated Testing** – Use JUnit, Selenium, RestAssured to **speed up testing**.
* **CI/CD Integration** – Automate deployments for faster feedback.

**🔹 4. Provide a Final Estimate to Client**

**Format for Client:**

* **Effort Estimate**: 22–25 days
* **Delivery Timeline**: ~4 weeks
* **Phases**: Requirement analysis → Development → Testing → UAT → Release

**🚀 Final Thoughts**

✅ **Break down the feature** into small, estimable tasks.  
✅ **Use Agile methodology** to track progress and adjust estimates.  
✅ **Factor in risks** like rework, integration issues, or client feedback.  
✅ **Automate testing and CI/CD** to speed up delivery.

=========================================================================

1. **How we can move from Monolithic to MicroServices?Design pattern to be used? DDD and Bounded Context.**

**Migrating from Monolithic to Microservices: A Complete Guide 🚀**

Moving from a **monolithic architecture** to **microservices** is a complex but rewarding process. It requires careful planning, refactoring, and adopting key design patterns such as **Domain-Driven Design (DDD)** and **Bounded Contexts**.

**🔹 1. Challenges with Monolithic Architecture**

Before moving, it's important to understand **why** monoliths can become problematic:  
❌ **Scalability Issues** – Scaling a single codebase is inefficient.  
❌ **Slow Deployment** – A small change requires the entire app to be rebuilt & deployed.  
❌ **Tightly Coupled Code** – Changes in one module can break others.  
❌ **Harder Maintenance** – Large codebase increases complexity.

**🔹 Solution? → Microservices Architecture!**

* Services are loosely coupled, independent, and scalable.
* Deployment and updates happen per service.
* Teams work on different services independently.

**🔹 2. Steps to Migrate from Monolithic to Microservices**

**✔️ Step 1: Identify Bounded Contexts (Using DDD - Domain-Driven Design)**

* **Bounded Contexts** define clear boundaries for different domains in the system.
* Each **domain** becomes a separate **microservice**.
* Example: In an **eCommerce** system, we might have:
  + **User Service** (manages authentication & users)
  + **Order Service** (handles purchases & invoices)
  + **Payment Service** (manages transactions)
  + **Inventory Service** (manages stock levels)

🔹 **How to Identify Bounded Contexts?**  
✔️ Analyze **business functionalities** and **data dependencies**.  
✔️ Use **Event Storming** to map system behaviors.  
✔️ Define **clear ownership** for each service.

**✔️ Step 2: Strangle the Monolith (Strangler Fig Pattern)**

Instead of rewriting everything at once:  
✔️ **Extract microservices gradually** (e.g., move authentication first, then orders).  
✔️ **Route requests** through an API Gateway (e.g., Zuul, Spring Cloud Gateway).  
✔️ **Refactor database** to be service-specific.

**✔️ Step 3: Implement Microservices Communication**

There are **two ways** microservices can communicate:  
🔹 **Synchronous (REST, gRPC)** → For simple request-response operations.  
🔹 **Asynchronous (Event-Driven, Kafka, RabbitMQ)** → For decoupling and high scalability.

**✔️ Step 4: Refactor the Database**

* A monolithic system often uses **a single database** for all functionalities.
* In microservices, each service should have its **own database** (**Database per Service Pattern**).
* If data needs to be shared, use **Event Sourcing** or **CQRS** to maintain consistency.

**✔️ Step 5: Implement API Gateway & Service Discovery**

✔️ **API Gateway** (Spring Cloud Gateway, Netflix Zuul) – Handles routing, security, load balancing.  
✔️ **Service Discovery** (Eureka, Consul) – Helps microservices dynamically locate each other.

**✔️ Step 6: Implement Transaction Management**

Microservices don't support traditional ACID transactions. Use **Saga Pattern** for distributed transactions:  
1️⃣ **Choreography-Based Saga** – Each service triggers the next step.  
2️⃣ **Orchestration-Based Saga** – A central controller manages transactions.

**✔️ Step 7: CI/CD & Deployment Strategy**

* Use **Docker & Kubernetes** for containerized deployment.
* Implement **CI/CD pipelines** for automated builds & deployments.

**🔹 3. Microservices Design Patterns**

| **Pattern** | **Use Case** |
| --- | --- |
| **Strangler Fig** | Migrate monolith gradually |
| **Database per Service** | Each service has its own database |
| **Saga Pattern** | Distributed transactions |
| **Event-Driven Architecture** | Decouple services using events |
| **CQRS (Command Query Responsibility Segregation)** | Separate read & write databases |
| **Service Registry & Discovery** | Auto-discover services (Eureka, Consul) |
| **API Gateway** | Centralized routing & security |

**🔹 4. Domain-Driven Design (DDD) & Bounded Context**

DDD helps in designing microservices by **grouping related functionalities** into **Bounded Contexts**.

🔹 **Example: An E-commerce System**

| **Bounded Context** | **Microservice** |
| --- | --- |
| User Management | User Service |
| Orders | Order Service |
| Payment Processing | Payment Service |
| Inventory | Inventory Service |

* Each Bounded Context is **loosely coupled** but **functionally cohesive**.
* Communication happens via **REST APIs, gRPC, or Events**.

**🔹 5. Example: Migrating a Monolithic Order Module to Microservices**

**Monolithic Order Management:**

java

CopyEdit

@RestController

public class OrderController {

@Autowired OrderService orderService;

@PostMapping("/order")

public ResponseEntity<String> createOrder(@RequestBody Order order) {

return ResponseEntity.ok(orderService.processOrder(order));

}

}

**Step 1: Extract OrderService as a Microservice**

java

CopyEdit

@RestController

@RequestMapping("/orders")

public class OrderMicroservice {

@PostMapping("/")

public ResponseEntity<String> createOrder(@RequestBody Order order) {

// Publish Event to Kafka

orderEventPublisher.publish(order);

return ResponseEntity.ok("Order placed successfully!");

}

}

**Step 2: Implement Asynchronous Communication (Event-Driven)**

🔹 **Using Kafka to communicate between Order and Payment Services**  
**Order Service → Kafka → Payment Service**

java

CopyEdit

@KafkaListener(topics = "order-events")

public void processPayment(String orderDetails) {

// Process payment asynchronously

}

**🚀 Final Thoughts**

✅ **Break the Monolith using Bounded Contexts** (DDD).  
✅ **Migrate Gradually** using **Strangler Fig Pattern**.  
✅ **Use Microservices Design Patterns** (Saga, CQRS, Event Sourcing).  
✅ **Ensure Microservices Communication** via API Gateway, Kafka, or gRPC.  
✅ **Implement CI/CD, Docker, Kubernetes** for seamless deployment.

=====================================================================

1. **What to do when you recognize performance degradation in system?**

**Handling Performance Degradation in a System**

When you notice **performance degradation**, you need a **structured approach** to **identify, analyze, and optimize** the bottlenecks. Here’s a step-by-step **troubleshooting guide** for handling system slowdowns efficiently.

**🔹 Step 1: Identify the Symptoms**

✔️ **High Response Time** – API calls take longer than expected.  
✔️ **Increased Latency** – Delay in database queries or service communication.  
✔️ **CPU/Memory Spikes** – Unusual resource consumption.  
✔️ **High Thread Contention** – Too many threads waiting for execution.  
✔️ **Increased Load & Slow Throughput** – More users, but slower request handling.

**📌 Tools to Monitor Performance:**

* **Application Performance Monitoring (APM)** – New Relic, Dynatrace, AppDynamics
* **Logging & Tracing** – ELK Stack (Elasticsearch, Logstash, Kibana), Grafana, Splunk
* **Profiling & Debugging** – JProfiler, YourKit, VisualVM
* **Distributed Tracing** – Jaeger, Zipkin (for microservices)

**🔹 Step 2: Analyze the Root Cause**

Once the issue is identified, analyze different system components:

**1️⃣ Check Application Code Efficiency**

✔️ Identify slow methods using **profiling tools**.  
✔️ Look for **unoptimized loops, nested iterations, or excessive object creation**.  
✔️ Check for **unnecessary synchronization or blocking calls**.  
✔️ Optimize Java collections – use **ConcurrentHashMap** instead of **synchronized HashMap**.

**🔹 Example: Optimizing a Loop** ❌ **Inefficient Code:**

java

CopyEdit

for (int i = 0; i < list.size(); i++) {

if (!set.contains(list.get(i))) {

set.add(list.get(i));

}

}

✅ **Optimized Code:**

java

CopyEdit

set.addAll(list);

**2️⃣ Check Database Performance**

✔️ Identify slow SQL queries using **EXPLAIN ANALYZE** (MySQL/PostgreSQL) or **SQL Profiler**.  
✔️ Optimize **indexes** – Ensure the correct use of primary keys, foreign keys, and composite indexes.  
✔️ Avoid **N+1 query problem** in Hibernate by using **JOIN FETCH** or **Batch Fetching**.  
✔️ Implement **connection pooling** (HikariCP is recommended).  
✔️ Use **caching (Redis, Ehcache)** to reduce frequent DB hits.

**🔹 Example: SQL Query Optimization** ❌ **Bad Query (Full Table Scan):**

sql

CopyEdit

SELECT \* FROM orders WHERE customer\_name = 'John Doe';

✅ **Optimized Query (Indexing):**

sql

CopyEdit

CREATE INDEX idx\_customer\_name ON orders(customer\_name);

SELECT \* FROM orders WHERE customer\_name = 'John Doe';

**3️⃣ Optimize JVM & Garbage Collection**

✔️ Monitor **GC logs** (-XX:+PrintGCDetails -Xloggc:gc.log).  
✔️ Tune JVM heap settings (-Xms -Xmx).  
✔️ Use **G1GC** or **ZGC** for large heap sizes (>8GB).  
✔️ Avoid **memory leaks** caused by unclosed resources.

**🔹 Example: Optimizing Garbage Collection**

* **Enable G1GC for large applications**:

sh

CopyEdit

-XX:+UseG1GC -XX:MaxGCPauseMillis=200 -XX:InitiatingHeapOccupancyPercent=45

* **Use Memory Analyzer (MAT) to find memory leaks**.

**4️⃣ Check Microservices Communication**

✔️ **Reduce synchronous calls (REST) and use asynchronous messaging (Kafka, RabbitMQ).**  
✔️ Implement **circuit breakers** (Resilience4j, Hystrix) to prevent cascading failures.  
✔️ Enable **connection pooling & HTTP Keep-Alive** to optimize REST API calls.  
✔️ Optimize JSON serialization (Use **Jackson FasterXML** or **Kryo**).

**🔹 Example: Using Circuit Breaker (Resilience4j)**

java

CopyEdit

@CircuitBreaker(name = "orderService", fallbackMethod = "orderFallback")

public String getOrderDetails() {

return restTemplate.getForObject("http://order-service/orders", String.class);

}

public String orderFallback(Exception e) {

return "Order Service is currently unavailable!";

}

**5️⃣ Check Network & Load Balancing**

✔️ **Use CDNs (Content Delivery Networks)** for static assets.  
✔️ **Enable GZIP Compression** for reducing response payload size.  
✔️ **Use Load Balancers (NGINX, HAProxy, AWS ALB)** to distribute traffic.  
✔️ Reduce latency by deploying services **closer to users** using **edge computing**.

**6️⃣ Implement Caching**

✔️ **Use in-memory caches (Redis, Memcached) to reduce DB calls**.  
✔️ Cache frequently used queries using **Spring Cache**.  
✔️ **Enable HTTP Caching** (ETag, Cache-Control headers).

**🔹 Example: Caching with Redis in Spring Boot**

java

CopyEdit

@EnableCaching

@Configuration

public class CacheConfig {

@Bean

public RedisCacheManager cacheManager(RedisConnectionFactory redisConnectionFactory) {

return RedisCacheManager.builder(redisConnectionFactory).build();

}

}

**🔹 Step 3: Optimize & Test the Fixes**

Once you apply optimizations, test the changes:  
✔️ **Load Testing** – Use JMeter, Gatling, k6 to simulate traffic.  
✔️ **Stress Testing** – Push system limits using Apache Bench (ab).  
✔️ **Profiling & Monitoring** – Continuously observe application performance.

**🚀 Final Thoughts**

✅ **Monitor & analyze system bottlenecks using APM tools.**  
✅ **Optimize code, database queries, and JVM tuning.**  
✅ **Implement caching, load balancing, and connection pooling.**  
✅ **Improve microservices resilience with circuit breakers and async messaging.**  
✅ **Continuously test using performance monitoring tools.**

===========================================================================

1. **What is SOLID principle**

[**https://www.youtube.com/watch?v=BM\_lSZPMClo**](https://www.youtube.com/watch?v=BM_lSZPMClo)

[**https://github.com/Java-Techie-jt/solid-principles-example**](https://github.com/Java-Techie-jt/solid-principles-example)

The **SOLID** principles are five core design principles in object-oriented programming that help create code that is **easy to maintain, scale, and test**. These principles are especially useful in Java and other OOP languages.

**🧱 SOLID Acronym**

| **Principle** | **Name** | **Description** |
| --- | --- | --- |
| S | Single Responsibility Principle (SRP) | A class should have only one reason to change. |
| O | Open/Closed Principle (OCP) | Software entities should be open for extension, but closed for modification. |
| L | Liskov Substitution Principle (LSP) | Subclasses should be substitutable for their base classes. |
| I | Interface Segregation Principle (ISP) | No client should be forced to depend on methods it does not use. |
| D | Dependency Inversion Principle (DIP) | High-level modules should not depend on low-level modules; both should depend on abstractions. |

**✅ Java Examples for Each Principle**

**1️⃣ Single Responsibility Principle (SRP)**

java

CopyEdit

// BAD: This class handles both user data and email functionality

class User {

public void saveToDatabase() { /\* ... \*/ }

public void sendEmail(String message) { /\* ... \*/ }

}

// GOOD: Split responsibilities into different classes

class User {

public void saveToDatabase() { /\* ... \*/ }

}

class EmailService {

public void sendEmail(String message) { /\* ... \*/ }

}

**2️⃣ Open/Closed Principle (OCP)**

java

CopyEdit

// BAD: We modify the class to add new types of discounts

class DiscountCalculator {

public double calculate(String type) {

if ("Silver".equals(type)) return 10;

else if ("Gold".equals(type)) return 20;

return 0;

}

}

// GOOD: Use inheritance or interface to extend behavior

interface Discount {

double calculate();

}

class SilverDiscount implements Discount {

public double calculate() { return 10; }

}

class GoldDiscount implements Discount {

public double calculate() { return 20; }

}

**Objects of a superclass should be replaceable with objects of a subclass without breaking the app.**

**✅ Good:  
Subclasses should behave like the parent class.**

**💡 Example:**

**java**

**CopyEdit**

**class Bird {**

**void fly() {}**

**}**

**class Crow extends Bird { } // ✅ OK**

**class Ostrich extends Bird { } // ❌ Problem – ostrich can’t fly**

**Better to redesign if the subclass can't behave like the parent.**

**🔸 I — Interface Segregation Principle (ISP)**

**Don’t force a class to implement methods it doesn't use.**

**✅ Good:  
Use smaller, more specific interfaces.**

**💡 Example:**

**java**

**CopyEdit**

**interface Printer {**

**void print();**

**}**

**interface Scanner {**

**void scan();**

**}**

**class SimplePrinter implements Printer { }**

**class AllInOneMachine implements Printer, Scanner { }**

**Don’t make SimplePrinter implement scan() if it can't scan.**

**🔸 D — Dependency Inversion Principle (DIP)**

**High-level classes should not depend on low-level classes, but on abstractions.**

**✅ Good:  
Depend on interfaces, not concrete classes.**

**💡 Example:**

**java**

**CopyEdit**

**interface Keyboard { }**

**class USBKeyboard implements Keyboard { }**

**class Computer {**

**private Keyboard keyboard;**

**public Computer(Keyboard keyboard) {**

**this.keyboard = keyboard;**

**}**

**}**

=====================================================================

1. **How object is garbage collected?(Expalin about Eden Space, Old Generation,,memory level, and WeakHashMap and others.)**

**Java Garbage Collection (GC) & Memory Management**

**📌 Overview**

Garbage Collection (GC) in Java **automatically removes unused objects** to free memory. Java Memory is divided into different sections:

* **Eden Space (Young Generation)**
* **Survivor Spaces (S0, S1)**
* **Old Generation (Tenured)**
* **Permanent/Metaspace (For class metadata)**

Java uses **different GC algorithms** (like G1, CMS, ZGC) to optimize memory management.

**🔹 Java Heap Memory Structure**

Java heap is divided into **three major parts**:

**1️⃣ Young Generation (Eden + Survivor)**

* New objects are created in **Eden Space**.
* Objects that survive a few garbage collections move to **Survivor Spaces (S0, S1)**.
* If an object **lives long enough**, it moves to **Old Generation**.

**2️⃣ Old Generation (Tenured)**

* Stores **long-lived objects** (e.g., large collections, cached data).
* Uses **Major GC (Old GC) → Slower than Young GC**.

**3️⃣ Metaspace (Before Java 8: PermGen)**

* Stores **class metadata, static variables**.
* Introduced in **Java 8** (Replaces PermGen).

**🔹 Java Garbage Collection Process**

**1️⃣ Minor GC (Young GC)**

* **Triggered when Eden Space is full.**
* Moves surviving objects from **Eden → Survivor**.
* Objects that live long **eventually move to Old Generation**.

**2️⃣ Major GC (Old GC)**

* **Triggered when Old Generation is full**.
* More expensive and can cause **application pause**.

**3️⃣ Full GC**

* Cleans both **Young and Old Generations**.
* Causes **Stop-The-World (STW)**, halting the application.

**🔹 Garbage Collection Algorithms in Java**

Java provides **different GC implementations**:

| **GC Algorithm** | **Use Case** | **Pause Time** |
| --- | --- | --- |
| **Serial GC** | Small applications (Single-threaded) | High |
| **Parallel GC** | Multi-core systems, better throughput | Medium |
| **CMS (Concurrent Mark Sweep)** | Low-latency applications | Low |
| **G1 (Garbage First) GC** | Large heap sizes (default in Java 9+) | Low |
| **ZGC (JDK 11+)** | Ultra-low latency | Very Low |

**🔹 How Garbage Collection Works? (Step-by-Step)**

1️⃣ **New object is created in Eden Space**.  
2️⃣ **Eden gets full** → **Minor GC runs** and moves surviving objects to Survivor Space.  
3️⃣ If an object **survives multiple GC cycles**, it moves to **Old Generation**.  
4️⃣ If Old Generation is **full**, **Major GC runs**, cleaning long-lived objects.  
5️⃣ **Full GC runs occasionally**, pausing the entire application.

**🔹 Reference Types & Garbage Collection**

| **Reference Type** | **When Object is Collected?** | **Example** |
| --- | --- | --- |
| **Strong Reference** | Never (Unless explicitly set to null) | String s = new String("Hello"); |
| **Soft Reference** | Only if **memory is low** | SoftReference<String> ref = new SoftReference<>(new String("Soft")); |
| **Weak Reference** | Collected in the **next GC cycle** | WeakReference<String> ref = new WeakReference<>(new String("Weak")); |
| **Phantom Reference** | Used for **post-mortem cleanup** | PhantomReference<String> ref = new PhantomReference<>(new String("Phantom"), queue); |

**🔹 WeakHashMap & Garbage Collection**

A **WeakHashMap** removes its keys automatically when there are **no strong references** to them.

**Example: WeakHashMap Automatically Deletes Entries**

java

CopyEdit

import java.util.WeakHashMap;

public class WeakHashMapExample {

public static void main(String[] args) throws InterruptedException {

WeakHashMap<Object, String> map = new WeakHashMap<>();

Object key1 = new Object();

Object key2 = new Object();

map.put(key1, "Value1");

map.put(key2, "Value2");

System.out.println("Before GC: " + map);

key1 = null; // Remove strong reference

System.gc(); // Request Garbage Collection

Thread.sleep(1000); // Wait for GC

System.out.println("After GC: " + map);

}

}

**Output:**

pgsql

CopyEdit

Before GC: {java.lang.Object@5e91993f=Value1, java.lang.Object@1e80bfe8=Value2}

After GC: {java.lang.Object@1e80bfe8=Value2}

✅ **key1 is garbage collected, and its entry is removed from the WeakHashMap.**

**🔹 How to Avoid Memory Leaks?**

1️⃣ **Close resources properly** (Database connections, file streams).  
2️⃣ **Use WeakReference or WeakHashMap** for cache management.  
3️⃣ **Avoid static collections** holding objects indefinitely.  
4️⃣ **Use try-with-resources** for handling AutoCloseable resources.  
5️⃣ **Enable JVM monitoring tools** (JVisualVM, JConsole) to track memory.

**🚀 Summary**

* **Garbage Collection (GC)** removes unused objects to prevent memory leaks.
* **Heap Memory**: **Young Generation (Eden + Survivor)** → **Old Generation** → **Metaspace**.
* **Types of GC**: **Minor GC, Major GC, Full GC**.
* **GC Algorithms**: **Serial, Parallel, CMS, G1, ZGC**.
* **Reference Types**: **Strong, Soft, Weak, Phantom**.
* **WeakHashMap automatically removes unused keys**.
* **Memory leaks can be prevented using proper resource handling**.

========================================================================

1. **Issue management tools(JIIRA).**

**Issue Management with JIRA**

**JIRA** is a popular **issue tracking and project management tool** developed by Atlassian. It is widely used for **bug tracking, issue management, and agile project management** in software development.

**🔹 Key Features of JIRA for Issue Management**

**1️⃣ Issue Tracking**

* Create, assign, track, and manage issues (bugs, tasks, stories, etc.).
* Each issue has a **unique ID** (e.g., PROJECT-101).
* Customizable fields like **priority, status, severity, due date**.

**2️⃣ Agile Support (Scrum & Kanban)**

* Supports **Scrum (Sprint planning)** and **Kanban (Visual workflow)**.
* **Sprint management**: Track backlog, in-progress tasks, and completed tasks.
* **Burn-down charts, velocity reports** for performance analysis.

**3️⃣ Workflow Customization**

* Define **custom workflows** for different issue types.
* Configure issue transitions like **"To Do → In Progress → Done"**.

**4️⃣ Integration with Development Tools**

* Works with **Bitbucket, GitHub, Jenkins, Confluence**.
* Automate issue updates via **JIRA API & Webhooks**.

**5️⃣ Reporting & Dashboards**

* Generate reports like **Bug trends, Sprint progress, Workload distribution**.
* Customizable dashboards for **real-time tracking**.

**🔹 JIRA Workflow for Issue Management**

**🔸 Typical Issue Lifecycle:**

1️⃣ **Create Issue** – Define title, description, priority, assignee.  
2️⃣ **Backlog Grooming** – Prioritize issues in the backlog.  
3️⃣ **In Progress** – Developer starts working on the issue.  
4️⃣ **Code Review** – Peer reviews code before merging.  
5️⃣ **Testing (QA)** – Issue is verified by the QA team.  
6️⃣ **Done** – Issue is resolved & closed.

🔹 **Example Workflow in JIRA:**

css

CopyEdit

[To Do] → [In Progress] → [Code Review] → [Testing] → [Done]

**🔹 Example: Creating an Issue in JIRA**

**Manually:**

1. Click **"Create"** in JIRA.
2. Select **Project** (e.g., E-Commerce System).
3. Choose **Issue Type** (Bug, Story, Task).
4. Add **Title & Description**.
5. Set **Priority** (Low, Medium, High, Critical).
6. Assign to a **developer**.
7. Click **Submit**.

=====================================================================

1. **Agile methodology.**

**📌 Agile Methodology – A Complete Guide**

**Agile** is a **software development methodology** that focuses on **iterative development, collaboration, flexibility, and customer feedback**. It allows teams to deliver software **incrementally** rather than in one large release.

**🔹 Key Principles of Agile (From Agile Manifesto)**

1️⃣ **Individuals and interactions** over processes and tools.  
2️⃣ **Working software** over comprehensive documentation.  
3️⃣ **Customer collaboration** over contract negotiation.  
4️⃣ **Responding to change** over following a fixed plan.

🚀 Agile is all about **adapting to change quickly** and delivering **value continuously**.

**🔹 Agile Methodology Frameworks**

There are several Agile frameworks, but the most popular ones include:

| **Framework** | **Best For** | **Key Features** |
| --- | --- | --- |
| **Scrum** | Team-based development | Sprints, Daily Standups, Backlogs |
| **Kanban** | Continuous workflow | Visual board, WIP limits |
| **Lean** | Reducing waste | Optimize flow, eliminate non-value activities |
| **XP (Extreme Programming)** | High-quality code | Test-driven development (TDD), Pair programming |
| **SAFe (Scaled Agile Framework)** | Large enterprises | Scaling Agile across multiple teams |

**🔹 Scrum – The Most Popular Agile Framework**

Scrum is **an iterative approach to software development** that divides work into **sprints** (2-4 week cycles).

**👨‍💻 Scrum Team Roles**

1️⃣ **Product Owner** – Defines requirements & prioritizes backlog.  
2️⃣ **Scrum Master** – Facilitates the Agile process, removes blockers.  
3️⃣ **Development Team** – Builds & delivers the software.

**🔸 Scrum Workflow (Sprint Cycle)**

markdown

CopyEdit

1. Product Backlog → 2. Sprint Planning → 3. Sprint Execution (Daily Standups) →

4. Sprint Review → 5. Sprint Retrospective → Repeat

**🔹 Scrum Events**

| **Event** | **Purpose** |
| --- | --- |
| **Sprint Planning** | Define sprint goals & backlog items. |
| **Daily Standup** | 15-minute meeting to track progress. |
| **Sprint Review** | Demonstrate completed work. |
| **Sprint Retrospective** | Identify improvements for the next sprint. |

=========================================================================

1. **Shallow and deep copy example**

**Shallow Copy vs. Deep Copy in Java**

Copying an object in Java can be done in two ways:

1. **Shallow Copy**: Copies field values **as-is**. If the object has references to other objects, only the references are copied, **not the actual objects**.
2. **Deep Copy**: Creates a completely new object with its own copies of **all referenced objects**.

**1. Shallow Copy (Default Copy)**

* Only the primitive values and object references are copied.
* The referenced objects remain **shared** between the original and the copied object.
* Modifying the referenced object in one affects the other.

**Example of Shallow Copy**

java

CopyEdit

class Address {

String city;

Address(String city) {

this.city = city;

}

}

class Person implements Cloneable {

String name;

Address address; // Reference Type

Person(String name, Address address) {

this.name = name;

this.address = address;

}

// Shallow Copy

@Override

protected Object clone() throws CloneNotSupportedException {

return super.clone(); // Default shallow copy

}

void display() {

System.out.println(name + " lives in " + address.city);

}

public static void main(String[] args) throws CloneNotSupportedException {

Address addr1 = new Address("New York");

Person p1 = new Person("John", addr1);

// Shallow Copy

Person p2 = (Person) p1.clone();

p1.display(); // John lives in New York

p2.display(); // John lives in New York

// Modifying the shared reference object

p2.address.city = "Los Angeles";

p1.display(); // John lives in Los Angeles 😱 (Unexpected)

p2.display(); // John lives in Los Angeles

}

}

**🔴 Issue in Shallow Copy**

* Both p1 and p2 share the **same address object**.
* Changing p2.address.city affects p1.address.city.

**2. Deep Copy (Complete Copy)**

* Copies **both primitive values and referenced objects**.
* Each object gets its own separate copy.

**Example of Deep Copy**

java

CopyEdit

class Address {

String city;

Address(String city) {

this.city = city;

}

// Deep copy constructor

Address(Address other) {

this.city = other.city;

}

}

class Person implements Cloneable {

String name;

Address address;

Person(String name, Address address) {

this.name = name;

this.address = new Address(address); // Deep Copy

}

@Override

protected Object clone() throws CloneNotSupportedException {

return new Person(this.name, this.address); // Manually copying object

}

void display() {

System.out.println(name + " lives in " + address.city);

}

public static void main(String[] args) throws CloneNotSupportedException {

Address addr1 = new Address("New York");

Person p1 = new Person("John", addr1);

// Deep Copy

Person p2 = (Person) p1.clone();

p1.display(); // John lives in New York

p2.display(); // John lives in New York

// Modifying copied object's reference

p2.address.city = "Los Angeles";

p1.display(); // John lives in New York ✅ (No change)

p2.display(); // John lives in Los Angeles

}

}

**✅ Deep Copy Fixes the Issue**

* p1 and p2 **now have separate address objects**.
* Changing p2.address.city does **not** affect p1.address.city.

**Key Differences**

| **Feature** | **Shallow Copy** | **Deep Copy** |
| --- | --- | --- |
| **Copy Type** | Only references | Full duplication |
| **Primitive Fields** | Copied | Copied |
| **Reference Fields** | Shared (same memory) | New objects created |
| **Modification in Copied Object** | Affects original | Does not affect original |
| **Performance** | Faster | Slower (extra object creation) |

**Which One to Use?**

* **Shallow Copy**: If objects are immutable or don't need independent copies.
* **Deep Copy**: When you need a completely **independent copy**.

===========================================================================

1. **Java Design Pattern and it’s implementation.**

**Singleton Key Characteristics:**

* **Private constructor (prevents external instantiation).**
* **Static instance (holds the single object).**
* **Static method (provides access to that instance).**

**🔨 Basic Implementation (Not Thread-Safe)**

**java**

**CopyEdit**

**public class Singleton {**

**private static Singleton instance;**

**private Singleton() {**

**// private constructor to prevent instantiation**

**}**

**public static Singleton getInstance() {**

**if (instance == null) {**

**instance = new Singleton(); // Lazy initialization**

**}**

**return instance;**

**}**

**}**

**❌ Issue: Not safe in multi-threaded environments.**

**🧵 Thread-Safe Singleton (Synchronized)**

**java**

**CopyEdit**

**public class Singleton {**

**private static Singleton instance;**

**private Singleton() {}**

**public static synchronized Singleton getInstance() {**

**if (instance == null) {**

**instance = new Singleton();**

**}**

**return instance;**

**}**

**}**

**✔️ Safe but may suffer from performance issues due to method-level synchronization.**

**🚀 Double-Checked Locking (Best Practice)**

**java**

**CopyEdit**

**public class Singleton {**

**private static volatile Singleton instance;**

**private Singleton() {}**

**public static Singleton getInstance() {**

**if (instance == null) {**

**synchronized (Singleton.class) {**

**if (instance == null) {**

**instance = new Singleton();**

**}**

**}**

**}**

**return instance;**

**}**

**}**

**🔒 Thread-safe, lazy-loaded, and better performance.**

**Factory Method Pattern — Definition**

**The Factory Method Pattern is a creational design pattern that:**

* **Defines an interface for creating an object,**
* **But lets subclasses decide which class to instantiate.**

**This pattern allows the code that uses the object (client code) to remain decoupled from the code that creates the object (factory code).**

**✅ When to Use It**

* **You want to delegate the instantiation logic to subclasses or a factory.**
* **You want to avoid using new in your business logic.**
* **You have a superclass with multiple subclasses, and based on input or logic, you want to return one of the subclasses.**

**🎯 Example Scenario:**

**Let’s say we have a Shape interface, and we want to return a specific shape based on input.**

**✅ Step-by-Step Java Implementation**

**1. Product Interface**

**java**

**CopyEdit**

**public interface Shape {**

**void draw();**

**}**

**2. Concrete Products**

**java**

**CopyEdit**

**public class Circle implements Shape {**

**public void draw() {**

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

**}**

**}**

**public class Rectangle implements Shape {**

**public void draw() {**

**System.out.println("Drawing Rectangle");**

**}**

**}**

**3. Creator / Factory Class**

**java**

**CopyEdit**

**public class ShapeFactory {**

**public Shape getShape(String type) {**

**if (type == null) return null;**

**if (type.equalsIgnoreCase("CIRCLE")) {**

**return new Circle();**

**} else if (type.equalsIgnoreCase("RECTANGLE")) {**

**return new Rectangle();**

**}**

**return null;**

**}**

**}**

**4. Client Code (Usage)**

**java**

**CopyEdit**

**public class Main {**

**public static void main(String[] args) {**

**ShapeFactory factory = new ShapeFactory();**

**Shape shape1 = factory.getShape("circle");**

**shape1.draw(); // Output: Drawing Circle**

**Shape shape2 = factory.getShape("rectangle");**

**shape2.draw(); // Output: Drawing Rectangle**

**}**

**}**

**🔍 Real-World Examples**

* **DocumentBuilderFactory in Java XML APIs.**
* **Connection object via DriverManager.getConnection(...).**
* **Spring BeanFactory and ApplicationContext.**

**✅ Benefits**

* **Loose coupling between client code and concrete classes.**
* **Promotes code reuse and scalability.**
* **Makes unit testing easier (can mock interfaces).**

**Builder Design Pattern — Definition**

**The Builder Pattern is a creational design pattern that lets you construct complex objects step-by-step. It separates the construction of a complex object from its representation, allowing the same construction process to create different representations.**

**✅ When to Use It**

* **When you have a class with many constructor parameters (especially optional ones).**
* **When you want to make immutable objects.**
* **When constructors with too many parameters become hard to read/maintain (the *telescoping constructor* problem).**

**✅ Real-World Analogy**

**Think of building a Pizza:**

**java**

**CopyEdit**

**Pizza pizza = new Pizza.Builder()**

**.size("Large")**

**.cheese(true)**

**.pepperoni(false)**

**.build();**

**🧱 Java Implementation (Standard Builder)**

**1. The Product Class**

**java**

**CopyEdit**

**public class User {**

**// Required parameters**

**private final String firstName;**

**private final String lastName;**

**// Optional parameters**

**private final int age;**

**private final String email;**

**private final String phone;**

**private User(Builder builder) {**

**this.firstName = builder.firstName;**

**this.lastName = builder.lastName;**

**this.age = builder.age;**

**this.email = builder.email;**

**this.phone = builder.phone;**

**}**

**public static class Builder {**

**private final String firstName;**

**private final String lastName;**

**private int age;**

**private String email;**

**private String phone;**

**public Builder(String firstName, String lastName) {**

**this.firstName = firstName;**

**this.lastName = lastName;**

**}**

**public Builder age(int age) {**

**this.age = age;**

**return this;**

**}**

**public Builder email(String email) {**

**this.email = email;**

**return this;**

**}**

**public Builder phone(String phone) {**

**this.phone = phone;**

**return this;**

**}**

**public User build() {**

**return new User(this);**

**}**

**}**

**@Override**

**public String toString() {**

**return firstName + " " + lastName + ", Age: " + age + ", Email: " + email + ", Phone: " + phone;**

**}**

**}**

**2. Usage Example**

**java**

**CopyEdit**

**public class Main {**

**public static void main(String[] args) {**

**User user = new User.Builder("John", "Doe")**

**.age(30)**

**.email("john.doe@example.com")**

**.phone("123-456-7890")**

**.build();**

**System.out.println(user);**

**}**

**}**

**✅ Output**

**yaml**

**CopyEdit**

**John Doe, Age: 30, Email: john.doe@example.com, Phone: 123-456-7890**

**🔥 Advantages**

* **Makes the code more readable and flexible.**
* **Prevents the need for multiple constructors.**
* **Makes objects immutable.**
* **Easily handles optional fields.**

**🧵 Builder Pattern in Java APIs**

**You’ve probably already used it:**

**java**

**CopyEdit**

**StringBuilder sb = new StringBuilder()**

**.append("Hello")**

**.append(" World");**

**Or with Lombok:**

**java**

**CopyEdit**

**@Builder**

**public class Employee {**

**private String name;**

**private int age;**

**}**

**🧰 4. Decorator Pattern**

**java**

**CopyEdit**

**interface Coffee {**

**String getDescription();**

**double cost();**

**}**

**class BasicCoffee implements Coffee {**

**public String getDescription() { return "Basic Coffee"; }**

**public double cost() { return 5.0; }**

**}**

**class MilkDecorator implements Coffee {**

**private Coffee coffee;**

**public MilkDecorator(Coffee coffee) {**

**this.coffee = coffee;**

**}**

**public String getDescription() {**

**return coffee.getDescription() + ", Milk";**

**}**

**public double cost() {**

**return coffee.cost() + 1.5;**

**}**

**}**

**💡 Usage:**

**java**

**CopyEdit**

**Coffee coffee = new MilkDecorator(new BasicCoffee());**

**System.out.println(coffee.getDescription()); // Basic Coffee, Milk**

**System.out.println(coffee.cost()); // 6.5**

**👀 5. Observer Pattern**

**java**

**CopyEdit**

**import java.util.\*;**

**interface Observer {**

**void update(String message);**

**}**

**class ConcreteObserver implements Observer {**

**private String name;**

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

**public void update(String message) {**

**System.out.println(name + " received: " + message);**

**}**

**}**

**class Subject {**

**private List<Observer> observers = new ArrayList<>();**

**public void subscribe(Observer o) {**

**observers.add(o);**

**}**

**public void notifyAllObservers(String msg) {**

**for (Observer o : observers) {**

**o.update(msg);**

**}**

**}**

**}**

**📣 Usage:**

**java**

**CopyEdit**

**Subject subject = new Subject();**

**subject.subscribe(new ConcreteObserver("Observer 1"));**

**subject.subscribe(new ConcreteObserver("Observer 2"));**

**subject.notifyAllObservers("New event occurred!");**