### Q What is AOP (Aspect-Oriented Programming)?

**AOP (Aspect-Oriented Programming)** is a programming paradigm that helps you **separate cross-cutting concerns** from your main business logic.

**🔧 In Simple Terms:**

AOP lets you write **modular code** for things like:

* Logging
* Security
* Transaction management
* Exception handling

Instead of repeating that code in multiple classes, you **write it once** and apply it where needed using AOP.

**📦 Example Use Case:**

Suppose you want to **log method execution time** in many service methods. Instead of putting System.out.println() in each method, you can write one logging aspect.

**🧱 Key Concepts in AOP:**

| **Term** | **Meaning** |
| --- | --- |
| **Aspect** | The cross-cutting logic (e.g., logging, security) |
| **Join Point** | A point in program execution (like a method call) where aspect can run |
| **Advice** | The action taken at a join point (before/after/around a method) |
| **Pointcut** | Expression to match join points (e.g., all methods in a package) |
| **Weaving** | The process of linking aspect with the target code |

**✅ Example in Spring AOP:**

@Aspect

@Component

public class LoggingAspect {

@Before("execution(\* com.example.service.\*.\*(..))")

public void logBeforeMethod(JoinPoint joinPoint) {

System.out.println("Method called: " + joinPoint.getSignature().getName());

}

}

This logs every method call in the com.example.service package **before** it runs.

**🔄 Types of Advice:**

* @Before – runs before method
* @After – runs after method
* @AfterReturning – runs after method successfully returns
* @AfterThrowing – runs after method throws an exception
* @Around – runs before and after the method (you control method execution)

**💡 Why Use AOP?**

* Cleaner code
* Reusable logic
* Centralized management of concerns like logging, security, etc.

Let me know if you want a full working Spring Boot example using AOP!

**✅ CompletableFuture in Java (Java 8+)**

CompletableFuture is part of java.util.concurrent and is used for **asynchronous programming**. It allows you to write **non-blocking**, **callback-based**, and **parallel** code in a clean and manageable way.

**Q Why Use CompletableFuture?**

* Run code asynchronously without blocking the main thread
* Chain multiple operations
* Handle errors gracefully
* Run tasks in parallel and combine results

**🧪 Basic Example**

import java.util.concurrent.CompletableFuture;

public class CompletableFutureExample {

public static void main(String[] args) {

CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {

System.out.println("Running async task in: " + Thread.currentThread().getName());

});

future.join(); // Wait for task to complete

System.out.println("Task Completed");

}

}

**🔗 Chaining with .thenApply()**

public class ThenApplyExample {

public static void main(String[] args) {

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {

return "Hello";

}).thenApply(result -> result + " World");

System.out.println(future.join()); // Output: Hello World

}

}

**🔁 Combining Two Futures**

public class CombineExample {

public static void main(String[] args) {

CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> "Hello");

CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() -> "World");

CompletableFuture<String> combined = future1.thenCombine(future2, (a, b) -> a + " " + b);

System.out.println(combined.join()); // Output: Hello World

}

}

**🚨 Exception Handling**

public class ExceptionHandlingExample {

public static void main(String[] args) {

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {

if (true) throw new RuntimeException("Something went wrong");

return "Success";

}).exceptionally(ex -> "Recovered from error: " + ex.getMessage());

System.out.println(future.join());

}

}

**🛠 Common Methods**

| **Method** | **Description** |
| --- | --- |
| runAsync(Runnable) | Run task with no return value |
| supplyAsync(Supplier) | Run task and return result |
| thenApply() | Transform result |
| thenAccept() | Consume result |
| thenRun() | Run another task without using result |
| thenCombine() | Combine results from two futures |
| exceptionally() | Handle exceptions |
| whenComplete() | Run after completion (success or failure) |

Let me know if you want a **real-world Spring Boot example** or a use case like **parallel API calls** or **chained DB processing**!

**🔍 What is Spring Boot Actuator?**

**Spring Boot Actuator** is a built-in module that provides **production-ready features** to help you monitor and manage your Spring Boot application — **without writing extra code**.

**✅ What It Does:**

It exposes **REST endpoints** that give you insights into:

* Application health
* Metrics (CPU, memory, HTTP requests)
* Environment variables
* Beans
* Thread dumps
* Custom application info
* and much more

**🚀 How to Enable Actuator**

1. **Add dependency in pom.xml:**

<dependency>

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

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

</dependency>

1. **Add config in application.properties or application.yml:**

management.endpoints.web.exposure.include=\*

**📡 Common Actuator Endpoints**

| **Endpoint** | **Description** | **Default URL** |
| --- | --- | --- |
| /actuator | Shows available actuator endpoints | <http://localhost:8080/actuator> |
| /actuator/health | Shows app health status (UP/DOWN) | <http://localhost:8080/actuator/health> |
| /actuator/info | Displays app info (you configure this) | <http://localhost:8080/actuator/info> |
| /actuator/metrics | Shows various metrics (memory, threads, etc) | <http://localhost:8080/actuator/metrics> |
| /actuator/beans | Lists all Spring Beans | <http://localhost:8080/actuator/beans> |
| /actuator/env | Shows environment properties | <http://localhost:8080/actuator/env> |
| /actuator/loggers | Manage logging levels | <http://localhost:8080/actuator/loggers> |

**🛠 Custom Info Example**

To add custom info to /actuator/info, in application.properties:

info.app.name=MyApp

info.app.version=1.0.0

Access it at: http://localhost:8080/actuator/info

**🔐 Security Note**

By default, most endpoints are **not exposed** over the web for safety. You must explicitly expose the ones you need and **secure them** if your app is public.

**📦 Use Cases:**

* Integrating with Prometheus/Grafana
* Health checks in Kubernetes
* Viewing app metrics on dashboards
* Debugging with bean and thread info

Let me know if you want to see an example project or integrate it with tools like **Prometheus**, **Zipkin**, or **Sleuth**!

**🔍 What is ConcurrentHashMap in Java?**

ConcurrentHashMap is a thread-safe, high-performance implementation of a hash-based map in Java. It's part of the java.util.concurrent package and is designed to allow **safe concurrent access by multiple threads** without the need for external synchronization.

**✅ Why Use ConcurrentHashMap?**

* Unlike HashMap, it **does not throw ConcurrentModificationException** during concurrent access.
* Better performance than Collections.synchronizedMap() because it uses **fine-grained locking**.
* Supports high concurrency with **multiple threads reading and writing** simultaneously.

**🔐 Thread-Safety Mechanism:**

ConcurrentHashMap uses **segment locking (Java 7)** or **bucket-level locking (Java 8 and above)**:

* Only the **bucket or portion** being updated is locked, not the entire map.
* Multiple threads can read and write different buckets **without blocking each other**.

**📦 Example Usage**

import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapExample {

public static void main(String[] args) {

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

map.put("A", 1);

map.put("B", 2);

// Fetching a value

System.out.println("Value for A: " + map.get("A"));

// Replacing value

map.replace("A", 10);

// Checking if key exists

System.out.println("Contains B? " + map.containsKey("B"));

}

}

**🆚 ConcurrentHashMap vs HashMap vs SynchronizedMap**

| **Feature** | **HashMap** | **SynchronizedMap** | **ConcurrentHashMap** |
| --- | --- | --- | --- |
| Thread-safe | ❌ No | ✅ Yes (locks whole map) | ✅ Yes (locks only part of map) |
| Performance (multi-threaded) | ❌ Poor | 🚫 Slow (global lock) | ✅ Fast (fine-grained locking) |
| Null keys/values | ✅ 1 null key, many null values | ✅ Same as HashMap | ❌ Not allowed (no null key/value) |

**🔧 Important Notes:**

* null keys or values are **not allowed**.
* Common methods like putIfAbsent(), compute(), and computeIfAbsent() are **atomic** and safe for concurrent use.

**⚙️ Real Use Case Example:**

map.computeIfAbsent("user123", key -> fetchDataFromDB(key));

This ensures only **one thread** will fetch and compute value for "user123" even if multiple threads try at once.

Let me know if you want examples with multiple threads or integration in a real project like a Spring Boot service.

### ⚡ What is a Circuit Breaker in Microservices?

A **Circuit Breaker** is a **fault-tolerance pattern** used in microservices architecture to prevent cascading failures when one service is **failing or responding slowly**.

### 🧠 Analogy:

Think of it like an **electrical circuit breaker**:

* If a service fails too many times, the breaker "trips" and stops calling that service for a while.
* After a timeout, it **checks if the service is healthy** before resuming normal operations.

### 💥 Why It’s Needed in Microservices

In microservices:

* Services often depend on other services (e.g., user service → order service → payment service).
* If one downstream service fails or slows down, **other services can be overloaded** trying to call it.
* This can cause **system-wide failures**.

**Circuit Breaker prevents this.**

### 🔁 Circuit Breaker States

| **State** | **Description** |
| --- | --- |
| **Closed** | Everything is working. Calls go through normally. |
| **Open** | Too many failures. All requests are **blocked** for a period of time. |
| **Half-Open** | Test calls are allowed to check if the service has recovered. |

### ✅ Popular Circuit Breaker Libraries

| **Tool** | **Integration** |
| --- | --- |
| **Resilience4j** | Lightweight, preferred for Spring Boot |
| **Hystrix** | Deprecated, used to be common |
| **Sentinel** | Used by Alibaba |

### 🚀 Example Using Resilience4j in Spring Boot

#### 1. **Add Dependency (Maven):**

<dependency>

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

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

</dependency>

#### 2. **Annotate Your Method:**

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

@RestController

public class OrderController {

@GetMapping("/order")

@CircuitBreaker(name = "orderService", fallbackMethod = "fallbackOrder")

public String getOrder() {

// Simulate failure

throw new RuntimeException("Order Service Down");

}

public String fallbackOrder(Throwable t) {

return "Fallback: Order service is unavailable.";

}

}

#### 3. **application.yml Configuration:**

resilience4j.circuitbreaker:

instances:

orderService:

registerHealthIndicator: true

slidingWindowSize: 5

minimumNumberOfCalls: 5

failureRateThreshold: 50

waitDurationInOpenState: 10s

### 🔐 Benefits of Using Circuit Breaker

* Prevents cascading failures
* Improves system resilience
* Enables graceful degradation (fallback)
* Allows quick recovery after temporary issues

Let me know if you’d like:

* A full Spring Boot project using Resilience4j
* Circuit breaker + retry + rate limiter integration example

### 🔄 What is a Circular Dependency?

A **circular dependency** occurs when **two or more components depend on each other directly or indirectly**, creating a **loop** that prevents proper instantiation or compilation.

### 🔧 In the context of ****Java/Spring Boot****, it usually happens like this:

#### ❌ Example of Circular Dependency in Spring:

@Component

public class A {

@Autowired

private B b;

}

@Component

public class B {

@Autowired

private A a;

}

* A depends on B
* B depends on A
* Spring doesn't know which one to create first → **throws BeanCurrentlyInCreationException**

### ⚠️ Why Is It a Problem?

* Causes **infinite loops** in object construction
* Breaks dependency injection
* Difficult to test, maintain, or scale

### 🔍 How to Detect?

Spring Boot will throw an error like:

org.springframework.beans.factory.BeanCurrentlyInCreationException:

Requested bean is currently in creation: Is there an unresolvable circular reference?

### ✅ How to Resolve It

#### 1. **Use** @Lazy

Delays initialization until actually needed:

@Component

public class A {

@Autowired

@Lazy

private B b;

}

#### 2. **Refactor your Design**

Move common logic to a third component (e.g., ServiceCommon) to break the cycle.

#### 3. **Use Setter Injection**

Helps Spring complete one object before wiring the other:

@Component

public class B {

private A a;

@Autowired

public void setA(A a) {

this.a = a;

}

}

#### 4. **Use Interfaces**

Depend on interfaces instead of concrete classes to decouple components.

### 🧠 Real-World Example in Microservices

* **Service A** calls **Service B**
* **Service B** tries to call back **Service A** (e.g., for status)
* If not handled properly via REST or event queues, this causes a runtime circular call loop or **mutual dependency** issue

Let me know if you want a **code example**, or help **refactoring a real circular dependency** in your project.