**📘 What is Spring AOP?**

**AOP (Aspect-Oriented Programming)** is a programming paradigm that aims to increase modularity by allowing the separation of **cross-cutting concerns** (like logging, security, transactions, etc.) from the main business logic.

**✅ Spring AOP Focus**

* **Spring AOP** is **proxy-based** and works only for **Spring Beans**.
* It supports **method-level weaving** at runtime (not compile-time or load-time).
* Uses **JDK dynamic proxies** (for interfaces) or **CGLIB proxies** (for classes without interfaces).

**🔑 Key Terminology**

| **Term** | **Description** |
| --- | --- |
| **Aspect** | A module of cross-cutting concern (e.g., logging, security) |
| **Advice** | Action taken by an aspect at a specific join point (e.g., before, after) |
| **Join Point** | A point during execution (e.g., method call) where an advice can be applied |
| **Pointcut** | Expression to select join points (method signatures, annotations, etc.) |
| **Weaving** | Linking aspects with other application types or objects |
| **Proxy** | A wrapper around a bean that applies the advice(s) |

**🔧 Types of Advice**

| **Advice Type** | **Annotation** | **When It Runs** |
| --- | --- | --- |
| **Before Advice** | @Before | Before the method executes |
| **After Returning** | @AfterReturning | After the method executes successfully |
| **After Throwing** | @AfterThrowing | If method throws an exception |
| **After (finally)** | @After | Runs after method regardless of outcome |
| **Around** | @Around | Controls method execution before and after (most powerful) |

**✅ Enabling AOP in Spring**

**1. Add AOP dependency (Maven)**

<dependency>

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

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

</dependency>

**2. Enable AspectJ support**

@Configuration

@EnableAspectJAutoProxy

public class AppConfig {

}

**🧠 Real-Time Example – Logging**

**Step 1: Create Aspect Class**

@Aspect

@Component

public class LoggingAspect {

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

public void logBefore(JoinPoint joinPoint) {

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

}

@AfterReturning(pointcut = "execution(\* com.example.service.\*.\*(..))", returning = "result")

public void logAfterReturning(JoinPoint joinPoint, Object result) {

System.out.println("AfterReturning: " + joinPoint.getSignature().getName() + " returned " + result);

}

@AfterThrowing(pointcut = "execution(\* com.example.service.\*.\*(..))", throwing = "ex")

public void logAfterThrowing(JoinPoint joinPoint, Throwable ex) {

System.out.println("Exception in " + joinPoint.getSignature().getName() + " - " + ex);

}

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

public void logAfter(JoinPoint joinPoint) {

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

}

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

public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {

System.out.println("Around - Before method: " + joinPoint.getSignature().getName());

Object result = joinPoint.proceed(); // executes the method

System.out.println("Around - After method: " + joinPoint.getSignature().getName());

return result;

}

}

**Step 2: Business Logic Class**

java

@Service

public class CustomerService {

public String getCustomerById(String id) {

if (id.equals("fail")) {

throw new RuntimeException("Customer not found");

}

return "Customer\_" + id;

}

}

**🎯 Pointcut Expressions**

| **Syntax** | **Meaning** |
| --- | --- |
| execution(\* package.Class.method(..)) | Matches method executions |
| within(com.example.service..\*) | All methods in package |
| this(beanType) | Proxies implementing a specific interface |
| @annotation(MyAnnotation) | Methods annotated with @MyAnnotation |

**📍 Order of Execution (Advice precedence)**

If multiple advices apply to the same method:

1. **Around**
2. **Before**
3. **Method Executes**
4. **AfterReturning** / **AfterThrowing**
5. **After**

**🧪 Testing AOP**

You can test by calling methods from CustomerService:

java

@SpringBootApplication

public class AopApp implements CommandLineRunner {

@Autowired

private CustomerService customerService;

public static void main(String[] args) {

SpringApplication.run(AopApp.class, args);

}

@Override

public void run(String... args) throws Exception {

customerService.getCustomerById("123");

try {

customerService.getCustomerById("fail");

} catch (Exception e) {

System.out.println("Caught Exception: " + e.getMessage());

}

}

}

**📦 Summary**

| **Concept** | **Description** |
| --- | --- |
| Cross-cutting concern | Code that affects multiple parts (e.g. logging) |
| AOP Benefit | Cleaner separation, DRY principles |
| Spring AOP Limitation | Works only on Spring-managed beans |
| Preferred Advice | @Around for flexibility, others for specific needs |