9. AOP (Aspect-Oriented Programming)

AOP 개념과 필요성

다음은 Spring 프레임워크에서의 **AOP(Aspect-Oriented Programming, 관점 지향 프로그래밍)**의 **개념, 필요성, 핵심 구성 요소, 내부 동작 원리, 실무 적용 예제, DI와의 차이점, 적용 위치 및 전략**까지 포함한 **가장 깊이 있는 설명**이다.

AOP란? (Aspect-Oriented Programming)

1. AOP의 개념

AOP(관점 지향 프로그래밍)는 프로그램의 핵심 로직(Core Concern)과는 **관심사가 다르지만 반복적으로 필요한 공통 기능(Cross-Cutting Concern)**을
한 곳에 모아 분리하여 재사용성과 유지보수성을 향상시키는 프로그래밍 패러다임이다.

> 즉, "모든 클래스에 반복적으로 들어가는 공통 코드를 하나의 '관점(Aspect)'으로 묶어 처리"하는 것

---

2. AOP가 필요한 이유

✳ 반복되는 부가기능을 모듈화

- 트랜잭션 처리
- 로그 기록
- 보안 검사
- 예외 처리
- 인증/인가
- 성능 측정

> 이 기능들이 **여러 클래스에 중복될 경우 코드가 분산**되어 버그 위험 증가, 테스트 어려움

---

3. OOP vs AOP

항목	OOP (객체지향)	AOP (관점지향)
관심사 분리	**기능 중심(역할)**으로 분리	**관점 중심(공통 관심사)**로 분리
중복 제거	상속, 인터페이스 등으로 해결 시 한계 존재	여러 지점에 **투명하게 적용** 가능
단위	클래스, 객체	**Aspect, Join Point, Advice**
핵심 목표	응집도 ↑, 결합도 ↓	**횡단 관심사(Cross-Cutting Concern) 분리**

---

4. AOP의 핵심 용어

용어	설명
Aspect	공통 기능(트랜잭션, 로그 등)을 담은 모듈
Advice	공통 기능이 **실제로 실행되는 코드** (`@Before`, `@After`, `@Around` 등)
Join Point	AOP가 적용 가능한 지점 (메서드 호출, 예외 발생 등)
Pointcut	Join Point 중에서 실제 AOP를 적용할 **조건 필터링** (예: 특정 패키지 이하 모든 메서드)
Weaving	Advice를 실제 대상 객체의 Join Point에 **결합**하는 과정
Proxy	스프링이 생성하는 **위임 객체**, 실제 대상 객체 앞에서 AOP 처리 수행

---

5. Spring AOP 동작 방식 (프록시 기반)

1. IoC 컨테이너가 빈을 생성할 때 **프록시 객체(대리 객체)를 대신 생성**
2. 사용자가 메서드를 호출하면 실제 대상 객체가 아니라 **프록시 객체를 통해 요청 처리**
3. 프록시 객체는 **Advice → Target Method → 후처리** 순으로 실행

> 기본적으로 Spring AOP는 **JDK 동적 프록시** 또는 **CGLIB 프록시**를 사용

---

6. 코드 예제 (Spring AOP)

6.1 Aspect 클래스 정의

```
@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* com.example.service.*.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println("[LOG] Before: " + joinPoint.getSignature());
    }

    @AfterReturning("execution(* com.example.service.*.*(..))")
    public void logAfter(JoinPoint joinPoint) {
        System.out.println("[LOG] After: " + joinPoint.getSignature());
    }
}
```

6.2 설정 클래스

```
@Configuration
@EnableAspectJAutoProxy
@ComponentScan("com.example")
public class AppConfig { }
```

→ `@EnableAspectJAutoProxy`는 AOP 프록시 기능을 활성화시킴

---

7. Advice 종류

종류	설명
`@Before`	대상 메서드 실행 **전**에 실행
`@After`	대상 메서드 실행 **후** 무조건 실행 (성공/실패 모두)
`@AfterReturning`	대상 메서드 **성공 후에만** 실행
`@AfterThrowing`	대상 메서드에서 예외 발생 시 실행
`@Around`	메서드 실행 전/후 **모두 제어** 가능 (가장 강력함)

---

8. Pointcut 예시

표현식	설명
`execution(* com.example..*.*(..))`	com.example 이하 모든 메서드
`execution(* *..*Service.*(..))`	모든 `*Service` 클래스의 모든 메서드
`execution(public void delete*(..))`	모든 delete로 시작하는 public 메서드
`within(com.example.controller..*)`	특정 패키지 내부 클래스 전체

---

9. AOP의 실무 적용 예

기능	설명
트랜잭션 관리	`@Transactional`은 내부적으로 AOP를 사용함
로깅 처리	모든 서비스 메서드 호출 전후 로그 남기기
성능 측정	메서드 실행 시간 측정 후 로그 출력
보안 체크	인증/인가 로직 Aspect로 분리
공통 예외 처리	try-catch를 Aspect로 분리 처리

---

10. AOP 사용 시 주의점

항목	설명
private 메서드에는 적용 안 됨	Spring AOP는 프록시 기반 → public 메서드만 가능

항목	설명
내부 메서드 호출은 AOP 우회됨	동일 클래스 내 `this.메서드()`는 프록시를 거치지 않음
프록시 객체 확인 어려움	디버깅 시 실제 클래스 대신 프록시 객체가 반환됨
성능 고려 필요	Advice가 많아지면 오버헤드 발생 가능

---

11. AOP vs DI 차이

항목	DI (의존성 주입)	AOP (관점 지향 프로그래밍)
목적	객체 간 관계 구성	공통 로직의 횡단 관심 분리
단위	Bean, Component, Service 등	Advice, Aspect, Join Point
예시	Controller에 Service 주입	모든 Service에 로깅 적용
구현 방식	`@Autowired`, `@Inject`, 생성자 등	`@Aspect`, `@Before`, 프록시 등

---

12. 요약

항목	설명
AOP란?	공통 기능을 핵심 비즈니스 로직에서 분리하여 모듈화
적용 위치	트랜잭션, 로깅, 보안, 예외 처리 등
핵심 어노테이션	`@Aspect`, `@Before`, `@Around`, `@EnableAspectJAutoProxy`
구성 요소	Aspect, Pointcut, Advice, JoinPoint, Weaving
실무 권장 방식	`@Around` + 포인트컷 + 별도 모듈 구성으로 재사용성 확보

핵심 용어: JoinPoint, Pointcut, Advice, Aspect

Spring AOP에서 자주 등장하는 핵심 용어인 **JoinPoint**, **Pointcut**, **Advice**, **Aspect**는 AOP 프로그래밍의 중심 개념입니다.
다음은 이 네 가지를 개념, 예시, 내부 구조, 상호작용, 실전 코드와 함께 **가장 깊이 있게 정리한 자료**입니다.

🔹 핵심 용어 요약

용어	의미 요약
JoinPoint	AOP가 적용 가능한 지점 (메서드 호출 등)
Pointcut	어떤 JoinPoint에 AOP를 적용할지 결정하는 규칙
Advice	JoinPoint에 끼워넣을 실제 실행 코드

용어	의미 요약
Aspect	Pointcut + Advice를 묶은 단위 (AOP 로직 모듈)

---

1. JoinPoint

✅ 개념

- 프로그램 실행 중에 **AOP가 끼어들 수 있는 지점**
- Spring AOP에서는 오직 **메서드 실행(메서드 호출)**만 JoinPoint로 취급함

✅ 예시

```
public void processOrder(String id)
```

→ 이 메서드를 호출하는 모든 시점이 JoinPoint
(Spring AOP는 필드 접근, 생성자 호출 등의 JoinPoint는 지원하지 않음)

✅ 주요 API

```
@Before("execution(...)")
public void log(JoinPoint joinPoint) {
    joinPoint.getSignature();       // 메서드 이름, 시그니처
    joinPoint.getArgs();            // 전달인자 목록
    joinPoint.getTarget();          // 실제 대상 객체
    joinPoint.getThis();            // 프록시 객체
}
```

---

2. Pointcut

✅ 개념

- AOP를 적용할 JoinPoint를 **필터링하는 표현식 또는 조건**
- 실제로 Advice가 **어느 지점에 적용될지 결정하는 룰**

✅ 표현 방식

```
execution(* com.example.service.*.*(..))
```

→ `com.example.service` 하위 클래스의 모든 메서드

주요 표현식 예시

표현식	설명
`execution(* *(..))`	모든 메서드
`execution(* com.app..*Service.*(..))`	Service 하위 모든 메서드

표현식	설명
`within(com.example..*)`	해당 패키지 내부 클래스 전체
`@annotation(com.example.Loggable)`	특정 어노테이션이 붙은 메서드

---

3. Advice

✅ 개념

- Pointcut으로 선정된 JoinPoint에 **실제로 실행될 코드 조각**
- 로그 출력, 트랜잭션 처리, 예외 처리 등을 수행

✅ 종류와 실행 시점

종류	어노테이션	실행 시점
`Before`	`@Before`	메서드 **호출 전**
`After`	`@After`	메서드 **종료 후 항상**
`AfterReturning`	`@AfterReturning`	**정상 종료 후**
`AfterThrowing`	`@AfterThrowing`	**예외 발생 후**
`Around`	`@Around`	메서드 **전후 모두 제어**, 결과 반환 포함

✅ 예시

```
@Around("execution(* com.example..*(..))")
public Object aroundLog(ProceedingJoinPoint joinPoint) throws Throwable {
    long start = System.currentTimeMillis();
    Object result = joinPoint.proceed(); // 대상 메서드 실행
    long end = System.currentTimeMillis();
    System.out.println("실행 시간: " + (end - start) + "ms");
    return result;
}
```

---

4. Aspect

✅ 개념

- 여러 Advice와 Pointcut을 **하나의 모듈 단위로 묶은 클래스**
- `@Aspect` 어노테이션을 붙인 클래스는 스프링이 **프록시 객체로 변환**

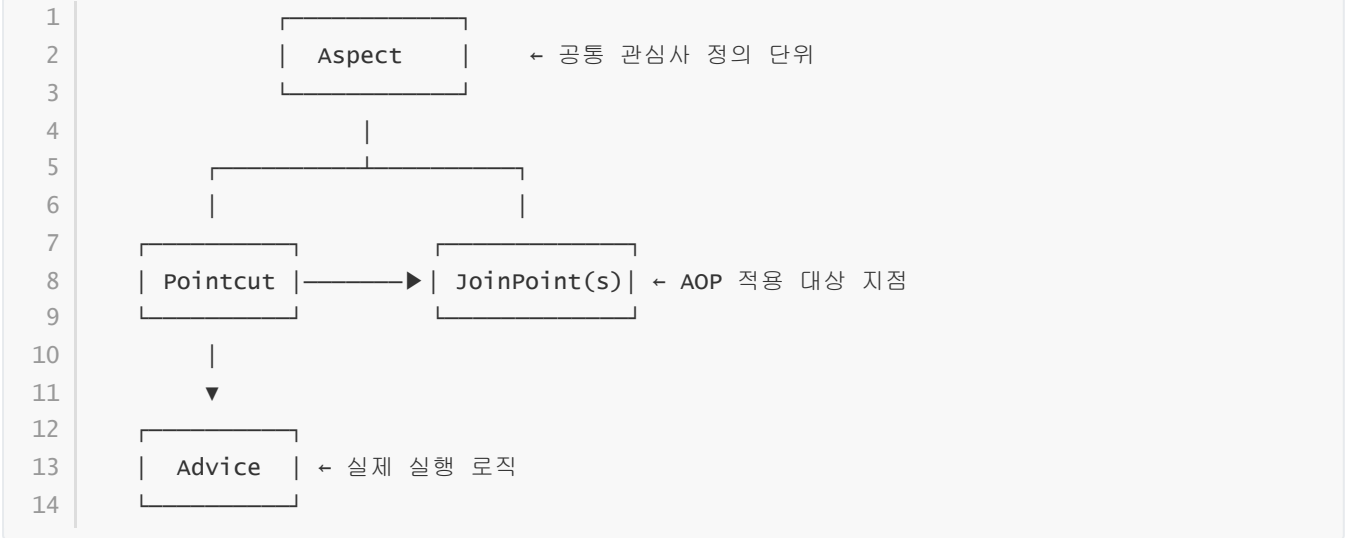
✅ 예시

```java
@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* com.example.service.*.*(..))")
    public void logBefore(JoinPoint jp) {
        System.out.println("메서드 시작: " + jp.getSignature());
    }

    @AfterReturning(pointcut = "...", returning = "result")
    public void logResult(Object result) {
        System.out.println("결과: " + result);
    }
}
```

> 하나의 Aspect 클래스에 여러 Advice를 선언할 수 있음

---

5. 용어 간 관계 요약 그림



---

6. 실전 요약

용어	개념 요약	코드 예
Aspect	AOP 묶음 클래스	`@Aspect` 클래스
Pointcut	적용 조건 지정	`execution(* com..*Service.*(..))`
JoinPoint	AOP 삽입 가능한 지점	메서드 호출
Advice	실제 실행할 부가기능	`@Around`, `@Before`, `@After` 등

---

7. 결론 요약

항목	설명
JoinPoint	AOP가 삽입될 수 있는 실제 코드 지점
Pointcut	어떤 JoinPoint에 Advice를 적용할지 지정
Advice	실제 실행될 코드 (로깅, 트랜잭션 등)
Aspect	Pointcut + Advice를 묶은 AOP 구성 클래스

`@Aspect` 설정 방식

- `@Before`, `@After`, `@Around`, `@AfterReturning`, `@AfterThrowing`

다음은 Spring AOP에서 사용되는 핵심 Advice 어노테이션인
`@Before`, `@After`, `@Around`, `@AfterReturning`, `@AfterThrowing`
각각의 **정의, 실행 시점, 용도, 코드 예시, 파라미터 처리 방식** 등을 포함한 **가장 깊이 있고 실전 중심적인 설명**입니다.

Spring AOP Advice 어노테이션

어노테이션	실행 시점	예외 시 실행 여부	반환값 접근	예외 접근	대표 용도
`@Before`	메서드 실행 **전**	❌	❌	❌	입력 값 검증, 로그 시작
`@After`	메서드 실행 **후** 항상	✅	❌	❌	정리 작업 (무조건 실행)
`@AfterReturning`	메서드 **정상 종료 후**	❌	✅	❌	결과 로그, 캐시
`@AfterThrowing`	메서드 **예외 발생 시**	✅	❌	✅	예외 로깅, 트랜잭션 롤백
`@Around`	**전/후/예외 모두**	✅	✅	✅	실행 시간 측정, 커스텀 제어

---

1. `@Before`

✅ 설명

- 대상 메서드가 실행되기 **직전에 실행**
- 전달되는 인자 확인 또는 사전 로그 처리에 적합

✅ 예시

```
@Before("execution(* com.example.service.*.*(..))")
public void beforeLog(JoinPoint jp) {
    System.out.println("Before: " + jp.getSignature());
}
```

---

2. `@After`

✅ 설명

- 대상 메서드가 실행된 후, **정상/예외 관계없이 무조건 실행**
- 리소스 해제, 로그 마무리 등에 사용

✅ 예시

```
@After("execution(* com.example.service.*.*(..))")
public void afterLog(JoinPoint jp) {
    System.out.println("After: " + jp.getSignature());
}
```

---

3. `@AfterReturning`

✅ 설명

- 대상 메서드가 **정상적으로 반환되었을 때만** 실행
- 리턴값을 Advice 메서드에 전달받을 수 있음

✅ 예시

```
@AfterReturning(
    pointcut = "execution(* com.example.service.*.*(..))",
    returning = "result")
public void afterReturn(Object result) {
    System.out.println("Return value: " + result);
}
```

- `returning` 속성의 값은 **Advice 메서드의 파라미터 이름과 동일해야 함**

---

4. `@AfterThrowing`

✅ 설명

- 대상 메서드에서 **예외가 던져졌을 때만** 실행
- 예외 로그, 모니터링, 알림 등에 사용

✅ 예시

```
@AfterThrowing(
    pointcut = "execution(* com.example.service.*.*(..))",
    throwing = "ex")
public void afterThrowing(Throwable ex) {
    System.out.println("예외 발생: " + ex.getMessage());
}
```

---

5. `@Around`

✅ 설명

- 대상 메서드의 실행을 **완전히 감싸서 전후/예외 모두 처리**
- `proceed()` 메서드를 호출해야 대상 메서드가 실행됨
- **가장 강력하고 범용적인 Advice**

✅ 예시

```
@Around("execution(* com.example.service.*.*(..))")
public Object aroundAdvice(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("Before 실행");

    try {
        Object result = pjp.proceed(); // 대상 메서드 호출
        System.out.println("AfterReturning 실행");
        return result;
    } catch (Throwable ex) {
        System.out.println("AfterThrowing 실행");
        throw ex;
    } finally {
        System.out.println("After 실행");
    }
}
```

✅ 주요 API

메서드	설명
`pjp.getArgs()`	메서드 인자 배열
`pjp.getSignature()`	메서드 정보

메서드	설명
`pjp.proceed()`	실제 대상 메서드 실행

실무 패턴별 추천 Advice

상황	추천 Advice
메서드 호출 로그	`@Before`, `@After`
메서드 실행 시간 측정	`@Around`
결과값 캐시 처리	`@AfterReturning`
예외 발생 시 슬랙 알림	`@AfterThrowing`
공통 트랜잭션 처리	`@Around` + `@Transactional`

주의 사항

- `@Before`, `@AfterReturning`은 **리턴값 조작 불가**
- `@Around`만이 유일하게 **리턴값 변경 및 실행 중단 가능**
- Advice는 반드시 Spring이 관리하는 **프록시 객체를 통해 실행됨**

전체 흐름 요약 (정상 종료 기준)

```
클라이언트 요청
    ↓
@Before Advice
    ↓
@Around - before
    ↓
(Target Method 실행)
    ↓
@Around - after
    ↓
@AfterReturning
    ↓
@After
    ↓
응답 반환
```

AOP 프록시 구현 방식

• JDK Dynamic Proxy

JDK Dynamic Proxy는 Java에서 제공하는 **런타임 프록시 객체 생성 기술**로,
인터페이스 기반의 프록시 객체를 생성하여 **관심사를 분리**하거나 **공통 기능을 메서드 호출 전후에 삽입**할 수 있게 해주는 강력한 메커니즘입이다.
Spring AOP의 핵심 기반 중 하나이며, AspectJ 없이도 동작할 수 있게 해준다.

✅ 1. 개념 요약

- **JDK Dynamic Proxy**는 **인터페이스 기반 객체**에 대해 **프록시 클래스를 런타임에 생성**
- 실제 대상 객체(Method Target)의 **메서드 호출을 가로채서 가공하거나 확장**할 수 있음
- Java SE 표준 API만으로 동작

✅ 2. 동작 구조

```
클라이언트 코드 → 프록시 객체 → InvocationHandler → 실제 구현체
```

- 프록시는 지정된 인터페이스를 구현하고, 호출된 메서드는 모두 `InvocationHandler`로 위임됨
- `InvocationHandler`는 중간에서 **공통 로직 처리 + 실제 메서드 호출**을 담당

✅ 3. 핵심 클래스 및 인터페이스

클래스/인터페이스	설명
`java.lang.reflect.Proxy`	프록시 클래스 생성 도구
`java.lang.reflect.InvocationHandler`	프록시의 메서드 호출을 가로채는 핸들러

✅ 4. 기본 예제

1. 대상 인터페이스

```
public interface HelloService {
    String sayHello(String name);
}
```

2. 대상 구현체

```
public class HelloServiceImpl implements HelloService {
    public String sayHello(String name) {
        return "Hello, " + name;
    }
}
```

3. 프록시 생성

```
HelloService target = new HelloServiceImpl();

HelloService proxy = (HelloService) Proxy.newProxyInstance(
    HelloService.class.getClassLoader(),
    new Class[]{HelloService.class},
    new InvocationHandler() {
        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
            System.out.println("[Before] " + method.getName());
            Object result = method.invoke(target, args);
            System.out.println("[After] " + method.getName());
            return result;
        }
    }
);
```

4. 실행 결과

```
proxy.sayHello("Spring");

// 출력
[Before] sayHello
[After] sayHello
```

✅ 5. 특징

항목	설명
인터페이스 필요	반드시 인터페이스를 구현한 클래스만 프록시 가능
런타임 생성	컴파일 시점이 아닌 **실행 시점에 클래스 생성**
리플렉션 사용	`Method.invoke()` 로 메서드 호출
성능	기본적으로 빠르나, 대량 처리 시 리플렉션 비용 주의

✅ 6. Spring AOP에서의 활용

Spring AOP는 프록시 기반으로 동작하며 다음과 같이 처리된다:

상황	사용 프록시 방식
대상이 **인터페이스 기반 Bean**일 경우	✅ **JDK Dynamic Proxy** 사용
대상이 **구현체 클래스만 있을 경우**	❌ → **CGLIB Proxy** 사용

Spring 설정:

```
@EnableAspectJAutoProxy(proxyTargetClass = false) // 기본: JDK Proxy
```

✅ 7. 장점과 단점

✅ 장점

- Java 기본 API만으로 동작 (AspectJ 불필요)
- 인터페이스 기반으로 설계된 서비스에 적합
- 객체 내부 변경 없이 외부 로직 삽입 가능

⚠️ 단점

- **인터페이스가 반드시 필요**
- 내부 메서드 호출(this)은 프록시를 거치지 않음
- 리플렉션 성능 저하 이슈 (크리티컬한 경우만)

✅ 8. 동작 흐름 요약

```
Proxy.newProxyInstance()
    └> 클래스 로더, 인터페이스 목록, InvocationHandler 전달
        └> 프록시 클래스 생성 및 바이트코드 동적 생성
            └> method 호출 → invoke() 내부에서 가로채 처리
```

✅ 9. 실무에서의 활용 예

활용 예	설명
Spring AOP 기본 구현	JDK 동적 프록시 기반으로 로깅, 트랜잭션 구현
Java에서 자체 AOP 구성	DI 없이 인터페이스 기반 프록시 구현 가능
다중 인터페이스 구현 클래스 감싸기	인터페이스별 로직 분리 가능
테스트 시 Mock 프록시 작성	JDK Proxy로 간단한 테스트용 프록시 작성 가능

✅ 10. 대체 기술: CGLIB과 비교

항목	JDK Dynamic Proxy	CGLIB Proxy
대상 필요 요소	인터페이스 반드시 필요	클래스만 있어도 가능
프록시 방식	인터페이스 구현 프록시	클래스 상속 기반 프록시
리플렉션 사용 여부	사용	바이트코드 생성
성능	약간 낮음	빠르지만 무겁고 제약 있음
Spring 기본 사용 우선순위	JDK → CGLIB 순	

✅ 요약

항목	설명
목적	런타임 시 인터페이스 기반 객체의 호출을 가로채기 위한 프록시 생성
핵심 API	`Proxy`, `InvocationHandler`
스프링 AOP 활용	인터페이스 기반일 경우 JDK 프록시 사용
실무 적용	공통 로직 삽입, 트랜잭션, 로깅, 인증, 커스텀 프레임워크 등
대안 기술	클래스 기반이라면 CGLIB 사용 필요

- # CGLIB Proxy

다음은 **CGLIB Proxy (Code Generation Library)**에 대한 가장 깊이 있고 체계적인 설명이다.
Spring AOP에서 JDK 동적 프록시를 대체하는 중요한 기술이며, **클래스를 기반으로 런타임 프록시 객체를 생성**하여 다양한 AOP 기능을 가능하게 해준다.

✅ 1. CGLIB Proxy란?

CGLIB (Code Generation Library)는 **클래스를 상속하여 프록시 객체를 생성하는 바이트코드 생성 라이브러리**다.

- 대상 객체가 **인터페이스를 구현하지 않아도** 프록시 가능
- **런타임에 새로운 클래스를 생성**하고, 그 클래스가 원본 클래스를 **상속**하여 동작

> Spring AOP에서는 대상 클래스가 인터페이스를 구현하지 않았을 경우, 자동으로 CGLIB을 사용함.

✅ 2. CGLIB Proxy vs JDK Dynamic Proxy

항목	CGLIB Proxy	JDK Dynamic Proxy
프록시 방식	**클래스 상속** 기반	**인터페이스 구현** 기반

항목	CGLIB Proxy	JDK Dynamic Proxy
인터페이스 필요 여부	❌ 필요 없음	✅ 인터페이스 필수
프록시 대상 제한	`final` 클래스/메서드는 프록시 불가	없음 (인터페이스 기반이므로)
Spring 기본 동작	대상이 클래스면 자동 적용	대상이 인터페이스면 우선 적용
성능	JDK보다 약간 빠름 (캐시 이후)	약간 느림 (리플렉션 기반)
바이트코드 수정 여부	✅ 있음 (ASM 또는 ByteBuddy 기반)	❌ 없음

✅ 3. Spring AOP에서 CGLIB이 사용되는 조건

Spring에서 AOP 프록시 방식은 기본적으로 다음 기준에 따라 결정된다:

▶️ 기본 설정

```
@EnableAspectJAutoProxy(proxyTargetClass = false) // 기본값: false → JDK Proxy 우선
```

▶️ 강제로 CGLIB 사용하려면:

```
@EnableAspectJAutoProxy(proxyTargetClass = true)
```

✅ 4. 동작 구조 요약

1. 대상 클래스의 바이트코드를 분석
2. 해당 클래스를 **상속**하는 새로운 클래스 동적으로 생성
3. 재정의된 메서드 내부에서 **Advice** → **대상 메서드** 실행
4. 프록시 객체를 빈으로 등록

✅ 5. 실습 예제

```
public class HelloService {
    public String hello(String name) {
        return "Hello, " + name;
    }
}
```

```
Enhancer enhancer = new Enhancer();
enhancer.setSuperclass(HelloService.class);
enhancer.setCallback(new MethodInterceptor() {
    @Override
    public Object intercept(Object obj, Method method, Object[] args, MethodProxy
proxy) throws Throwable {
        System.out.println("[Before] " + method.getName());
```

```
        Object result = proxy.invokeSuper(obj, args); // 원본 메서드 호출
        System.out.println("[After] " + method.getName());
        return result;
    }
});

HelloService proxy = (HelloService) enhancer.create();
proxy.hello("Spring");
```

결과

```
[Before] hello
[After] hello
```

✅ 6. 주요 API

클래스/인터페이스	설명
`Enhancer`	CGLIB 프록시 객체 생성기
`MethodInterceptor`	JDK의 `InvocationHandler`와 유사한 인터페이스
`MethodProxy`	`invokeSuper()`를 통해 원본 메서드 호출

✅ 7. 프록시 객체의 생성 방식

CGLIB은 **ASM / ByteBuddy**와 같은 바이트코드 조작 라이브러리를 사용하여 다음을 수행함:

- 런타임에 대상 클래스를 **extends**
- 메서드를 override하여 중간에 **Advice 코드 삽입**
- 생성된 프록시 클래스는 **캐시되어 재사용 가능**

✅ 8. 주의 사항

항목	설명
`final` 클래스/메서드 금지	상속 기반이므로 override 불가한 경우 프록시 생성 실패
생성자 호출 문제	생성자는 프록시화 불가능 → 생성자 내부 로직은 가로챌 수 없음
private 메서드	override 불가 → AOP 적용 불가
내부 호출 우회 문제	this.메서드() 호출은 프록시를 거치지 않음 → AOP 적용 안 됨

✅ 9. 실무 사용 예

상황	설명
인터페이스가 없는 레거시 클래스 프록시화	JDK Proxy로 불가 → CGLIB 사용
공통 로직 삽입 (로깅, 트랜잭션 등)	프록시 기반 AOP 구성
@Transactional이 클래스에 적용된 경우	CGLIB이 트랜잭션 프록시 생성

✅ 10. AOP 프록시 적용 여부 판단 기준

대상 Bean 특성	Spring이 선택하는 프록시 방식
인터페이스만 존재	JDK Dynamic Proxy
클래스만 존재	CGLIB Proxy
`proxyTargetClass=true` 설정	항상 CGLIB 사용
`final` 클래스 or 메서드	❌ CGLIB 사용 불가, 예외 발생 가능

✅ 11. Spring Boot에서 AOP 설정 방식

```
spring:
  aop:
    proxy-target-class: true # CGLIB 강제 사용
```

✅ 12. 요약

항목	설명
정의	대상 클래스를 상속하여 프록시 클래스 생성
기반 기술	ASM, ByteBuddy
프록시 조건	클래스 기반, 인터페이스 없어도 가능
주의점	final, private, 생성자 호출 불가
Spring 적용	인터페이스 없는 Bean에 자동 적용됨
실무 활용	트랜잭션, 로깅, 보안, 레거시 클래스 프록시 처리

실전 활용 예

• 로깅

다음은 **Spring 기반 애플리케이션에서의 로깅(logging)**에 대한 **가장 깊이 있고 실무 중심적인 설명**이다.
Java의 전통적 로깅 개념부터, Spring Boot에서의 로깅 설정, 주요 로깅 프레임워크(SLF4J, Logback, Log4j2), AOP 기반 로깅, MDC 활용법, 그리고 로그 전략까지 전부 포함한다.

✅ 1. 로깅이란?

로깅(logging)은 **프로그램의 실행 흐름, 상태, 오류, 디버깅 정보 등을 외부로 출력**하여
시스템의 **상태 추적 및 문제 해결, 보안 감시, 통계 분석**에 활용되는 핵심 도구이다.

> `System.out.println()`은 간단하지만 성능, 확장성, 관리 측면에서 부적절함
> → **전문 로깅 프레임워크**를 사용해야 함

✅ 2. Java 로깅 시스템 개요

계층	예시
Logging API 표준화 계층	**SLF4J**, **JUL**, Commons Logging
실제 구현체	**Logback**, Log4j2, JUL, tinylog 등
Spring Boot 기본 설정	**SLF4J + Logback** 조합

✅ 3. SLF4J (Simple Logging Facade for Java)

역할

- **로깅 추상화 계층** (Facade)
- 다양한 구현체(Logback, Log4j2 등)에 **종속되지 않고 코딩 가능**

사용 예

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class MyService {
    private static final Logger log = LoggerFactory.getLogger(MyService.class);

    public void run() {
        log.info("서비스 실행 시작");
        log.debug("디버그 정보: {}", someValue);
        log.warn("경고 발생");
        log.error("오류 발생", exception);
    }
}
```

✅ 4. Spring Boot의 기본 로깅 구조

계층	기본 설정
추상화	SLF4J
구현체	Logback
설정 파일	`application.yml`, `logback-spring.xml`

자동 설정

- `spring-boot-starter-logging`이 자동으로 SLF4J + Logback 구성
- `System.out.println()` 없이도 `@Slf4j`, `log.info()`로 출력 가능

✅ 5. 로그 레벨 (우선순위 높음 → 낮음)

레벨	설명
`ERROR`	심각한 오류 발생
`WARN`	경고성 메시지, 주의 필요
`INFO`	정상 흐름 정보, 운영 로그
`DEBUG`	상세한 내부 동작 정보
`TRACE`	가장 상세한 로직 추적 (잘 안 씀)

> 설정된 레벨보다 **상위 레벨만 출력됨**

✅ 6. 설정 방법

`application.yml`

```yaml
logging:
  level:
    root: INFO
    com.example.service: DEBUG
  file:
    name: logs/app.log
  pattern:
    console: "%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger - %msg%n"
```

`logback-spring.xml`

```
<configuration>
  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>logs/app.log</file>
    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger - %msg%n</pattern>
    </encoder>
  </appender>

  <root level="INFO">
    <appender-ref ref="FILE"/>
  </root>
</configuration>
```

✅ 7. Lombok `@Slf4j` 활용

```
import lombok.extern.slf4j.Slf4j;

@Slf4j
@Service
public class MyService {
    public void hello() {
        log.info("헬로우");
    }
}
```

> Lombok이 `Logger log = LoggerFactory.getLogger(...)` 자동 생성

✅ 8. AOP 기반 공통 로깅

Aspect를 이용한 호출/응답 로깅

```
@Aspect
@Component
public class LoggingAspect {

    @Around("execution(* com.example.service..*(..))")
    public Object logMethod(ProceedingJoinPoint pjp) throws Throwable {
        String method = pjp.getSignature().toShortString();
        Object[] args = pjp.getArgs();

        log.info("▶▶ 호출: {} args={}", method, Arrays.toString(args));

        try {
            Object result = pjp.proceed();
            log.info("◀◀ 리턴: {} result={}", method, result);
            return result;
        } catch (Throwable ex) {
```

```
            log.error("XX 예외: {} ex={}", method, ex.getMessage());
            throw ex;
        }
    }
}
```

✅ 9. MDC (Mapped Diagnostic Context) 활용

- **로그마다 사용자 정보, 트랜잭션 ID 등 Context를 삽입**할 수 있음

```
MDC.put("userId", "admin123");

log.info("데이터 조회 요청");

MDC.clear();
```

로그 패턴 설정 (logback.xml)

```
<pattern>%d{HH:mm:ss} [%thread] %X{userId} %-5level %logger - %msg%n</pattern>
```

→ 로그 출력:

```
12:30:55 [main] admin123 INFO com.example.Service - 데이터 조회 요청
```

✅ 10. 실무 로깅 전략

전략	설명
레벨 구분 명확히	DEBUG와 INFO 혼동 금지
운영 환경에서는 `DEBUG` 제거	`INFO` 이상만 출력
파일 + 콘솔 로그 분리	운영/디버깅 분리
예외 로그는 항상 `log.error(…, ex)`	스택트레이스 포함 필요
AOP + MDC 조합으로 유저 추적	API 기반 시스템에서 유용

✅ 11. 요약

항목	설명
기본 구조	SLF4J (추상화) + Logback (구현)
어노테이션 사용	`@Slf4j` (Lombok) 추천
설정 방법	`application.yml` 또는 `logback-spring.xml`

항목	설명
공통 로깅 처리	AOP 기반 `@Around` Advice
고급 기능	MDC, 비동기 로그, 파일 분리 등

- # 보안

Spring에서의 **보안(Security)**은 대부분 **Spring Security**를 통해 처리되며,
인증(Authentication), 인가(Authorization), 암호화(Encoding), CSRF 방어, 세션 관리, JWT 처리, OAuth 연동, 비밀번호 보호 등 **애플리케이션 전반의 보안 요구사항을 통합적으로 해결**하는 강력한 프레임워크이다.

아래는 **Spring 보안의 개념, 목적, 구성요소, 핵심 기능, 내부 동작 원리, 실무 전략까지 포함한 가장 깊이 있는 설명**이다.

🔐 Spring 보안(Security) 개요

1. 정의

Spring Security는 Spring 애플리케이션에 **인증/인가, 세션 보안, 암호화, 공격 방어 기능**을 제공하는 **보안 프레임워크**이다.

- 웹 보안 (HTTP 요청 필터링, 세션 보안)
- 메서드 보안 (`@Secured`, `@PreAuthorize`)
- OAuth2, JWT, LDAP, SSO, CSRF, CORS 등 지원
- 확장성과 커스터마이징이 매우 뛰어남

---

2. 보안이 필요한 이유

위협 종류	방지 수단
인증 우회 공격	로그인 필터링, 세션 검사
권한 상승	역할 기반 인가 검증
CSRF	토큰 기반 검증
XSS/Clickjacking	헤더 설정
데이터 탈취	비밀번호 암호화, HTTPS
무차별 대입 공격	로그인 제한, CAPTCHA

---

3. 핵심 구성 요소

구성 요소	설명
`Authentication`	사용자가 **누구인지 확인**
`Authorization`	사용자가 **무엇을 할 수 있는지 검사**

구성 요소	설명
`SecurityFilterChain`	HTTP 요청 필터 체인 구성
`UserDetailsService`	사용자 정보 로딩
`PasswordEncoder`	비밀번호 암호화 및 검증
`AuthenticationManager`	인증 처리 총괄
`GrantedAuthority`	권한(ROLE_USER, ROLE_ADMIN 등)
`SecurityContext`	현재 인증 정보 저장소

---

4. 기본 동작 흐름

```
[요청] → SecurityFilterChain → AuthenticationFilter →
AuthenticationManager → UserDetailsService → UserDetails 반환 → 인증 성공 →
SecurityContextHolder 저장 → Controller 진입
```

---

5. 핵심 기능

기능	설명
폼 로그인	기본 로그인 페이지 제공 (`/login`)
세션 관리	세션 고정 공격 방지, 세션 만료 처리
패스워드 암호화	`BCryptPasswordEncoder` 등 사용
역할 기반 인가	URL, 메서드 단위 `@Secured`, `@PreAuthorize`
CSRF 방어	POST 요청에 대해 토큰 요구
CORS 설정	다른 도메인 요청 허용/차단
OAuth2/OpenID	외부 인증 서버 연동
JWT 토큰 인증	Stateless 방식 지원

---

6. 설정 예제 (Spring Boot 3.x 이상, Security 6 기준)

```
@Configuration
@EnableMethodSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
```

```
        return http
            .csrf(AbstractHttpConfigurer::disable)
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .requestMatchers("/user/**").hasAnyRole("USER", "ADMIN")
                .anyRequest().permitAll()
            )
            .formLogin(Customizer.withDefaults())
            .build();
    }

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails user = User.withUsername("user")
            .password(passwordEncoder().encode("1234"))
            .roles("USER")
            .build();

        return new InMemoryUserDetailsManager(user);
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

7. 실무 보안 전략

항목	전략
비밀번호 저장	✅ **BCrypt, Argon2 사용**, 절대 평문 금지
인증 방식	✅ JWT 또는 세션 기반 선택
CSRF 공격	✅ 상태 저장 방식에서는 반드시 CSRF 보호 활성화
인증 우회	✅ 필터 체인에서 `permitAll()` 조절 주의
인가 검증	✅ URI + 메서드 두 군데 다 확인
관리자 권한	✅ 반드시 별도 Role로 분리하고 절대 노출 금지
OAuth 연동	✅ `spring-security-oauth2-client` 사용
민감한 데이터 로그	❌ 사용자 비밀번호, 토큰 등 로깅 금지
보안 헤더 적용	✅ `frameOptions()`, `xssProtection()` 등 적용

8. 보안 관련 어노테이션

어노테이션	설명
`@Secured("ROLE_ADMIN")`	메서드 접근 제어
`@PreAuthorize("hasRole('ADMIN')")`	SpEL 기반 인가
`@PostAuthorize(...)`	메서드 실행 후 인가 검사
`@WithMockUser(...)`	테스트용 인증 사용자 주입
`@AuthenticationPrincipal`	현재 인증된 사용자 정보 추출

---

9. 토큰 기반 인증 (JWT)

```
Authorization: Bearer <token>
```

Spring Security는 기본적으로 세션 기반이지만, **JWT 필터를 커스터마이징하여 무상태(stateless) 인증 시스템 구축 가능**

---

10. 보안 관련 클래스 요약

클래스/인터페이스	역할
`UserDetailsService`	사용자 인증 정보 제공
`UserDetails`	사용자 객체
`Authentication`	인증 토큰
`SecurityContextHolder`	인증 정보 저장소
`AuthenticationManager`	인증 책임자
`GrantedAuthority`	권한 정보

---

11. 요약

항목	설명
인증(Authentication)	로그인, 신원 확인
인가(Authorization)	권한 확인, 접근 통제
기본 방식	필터 체인 + 세션/토큰 인증
주요 기술	BCrypt, JWT, OAuth2, CSRF 보호

항목	설명
실무 권장	역할 기반 접근 통제 + AOP 기반 메서드 보안 + 로그 제한

- # 트랜잭션

다음은 **Spring의 트랜잭션(Transaction) 처리**에 대한 가장 깊이 있는 설명이다.
트랜잭션의 개념, Spring 트랜잭션의 구조, `@Transactional` 어노테이션의 동작 방식, 내부 AOP 메커니즘, 트랜잭션 전파, 격리 수준, 롤백 전략, 선언적 vs 명령적 처리, 실무에서의 실수 방지 전략까지 완벽하게 다룬다.

🔄 1. 트랜잭션이란?

✅ 정의

트랜잭션은 **하나의 작업 단위를 논리적으로 묶어 처리**하는 것으로, 해당 단위는 **모두 성공하거나 모두 실패해야 함**을 보장한다.

→ 데이터베이스에서의 **원자성(Atomicity)** 보장을 위한 핵심 개념이다.

✅ ACID

요소	설명
Atomicity	전부 성공 or 전부 실패
Consistency	일관된 상태 유지
Isolation	동시에 실행되어도 서로 간섭 없음
Durability	커밋된 결과는 영구 보존됨

⚙️ 2. Spring 트랜잭션 처리 구조

Spring은 **프로그래밍적 + 선언적** 트랜잭션 모두 지원한다.
선언적 방식(@Transactional)이 가장 많이 쓰인다.

주요 구성 요소

컴포넌트	역할
`PlatformTransactionManager`	트랜잭션 시작, 커밋, 롤백을 관리
`@Transactional`	선언적 트랜잭션 경계 지정
`TransactionInterceptor`	AOP로 트랜잭션을 걸어주는 Advice
`DataSourceTransactionManager`	JDBC 기반 트랜잭션 매니저
`JpaTransactionManager`	JPA 기반 트랜잭션 매니저

✅ 3. 기본 설정 예제

설정 (Spring Boot 기준)

```
@SpringBootApplication
@EnableTransactionManagement
public class MyApp { }
```

트랜잭션 적용 예

```
@Service
public class OrderService {

    @Transactional
    public void placeOrder(OrderRequest request) {
        orderRepository.save(...);
        paymentService.process(...); // 내부 호출 주의
    }
}
```

📌 4. `@Transactional` 핵심 속성

속성	기본값	설명
`propagation`	`REQUIRED`	트랜잭션 전파 방식
`isolation`	DB 기본값	트랜잭션 격리 수준
`rollbackFor`	`RuntimeException`	롤백 예외 지정
`readOnly`	`false`	읽기 전용 트랜잭션 최적화
`timeout`	-1 (무제한)	제한 시간 초과 시 rollback

🔁 5. 트랜잭션 전파 속성 (propagation)

속성	설명
`REQUIRED`	기존 트랜잭션 있으면 참여, 없으면 새로 시작
`REQUIRES_NEW`	항상 새 트랜잭션 시작 (기존은 일시 중단)
`NESTED`	중첩 트랜잭션 (Savepoint 사용)
`MANDATORY`	기존 트랜잭션 없으면 예외
`NEVER`	트랜잭션 있으면 예외
`SUPPORTS`	있으면 참여, 없으면 비트랜잭션

속성	설명
`NOT_SUPPORTED`	트랜잭션 없이 실행 (중단)

🔐 6. 격리 수준 (isolation)

수준	설명
`READ_UNCOMMITTED`	커밋되지 않은 데이터도 읽음 (더티 읽기 가능)
`READ_COMMITTED`	커밋된 데이터만 읽음 (SQL 기본)
`REPEATABLE_READ`	같은 쿼리 반복 시 항상 동일한 결과
`SERIALIZABLE`	가장 강력한 수준, 완전 직렬화
`DEFAULT`	DB의 기본 격리 수준 사용

❗ 7. 롤백 전략

Spring은 기본적으로 `RuntimeException`, `Error` 발생 시 rollback 한다.

예외 상황 명시적 처리 예

```
@Transactional(rollbackFor = Exception.class)
public void process() throws Exception {
    // checked 예외도 rollback
}
```

> `rollbackFor`, `noRollbackFor` 속성으로 제어 가능

⚠️ 8. 실무 실수 방지 전략

문제	원인	해결 방법
트랜잭션 적용 안 됨	`@Transactional`이 **인터페이스나 private 메서드**에 붙음	반드시 `public` 메서드에 적용
내부 메서드 호출 시 AOP 무시	`this.메서드()`로 호출됨 → 프록시 우회	별도 컴포넌트로 분리하거나 `ApplicationContext` 통해 호출
checked 예외에서 rollback 안 됨	기본은 `RuntimeException`만 롤백	`rollbackFor` 명시
readOnly 속성 무시됨	MyBatis, JPA 설정과 상충 가능성 있음	트랜잭션 매니저 설정 확인

문제	원인	해결 방법
병렬 업데이트 시 충돌 발생	격리 수준 낮음	`REPEATABLE_READ` 또는 `OPTIMISTIC_LOCK` 사용

⚙️ 9. 트랜잭션 커밋/롤백 동작 순서

```
1. 클라이언트 → 트랜잭션 시작
2. 비즈니스 로직 실행
3. 예외 발생 여부 확인
4. 예외 발생 시 → rollback
5. 성공 시 → commit
6. 후처리 / 이벤트 전파
```

🔁 10. 선언적 vs 프로그래밍적 트랜잭션

방식	예시
선언적 방식	`@Transactional` 사용 (가장 일반적)
프로그래밍 방식	`TransactionTemplate`, `PlatformTransactionManager` 직접 호출

```
transactionTemplate.execute(status -> {
    // 로직 실행
    return null;
});
```

→ **동적으로 트랜잭션 조건을 바꾸고 싶을 때** 유용

✅ 11. 요약

항목	설명
트랜잭션 기본 목적	원자성, 일관성, 격리성, 지속성 보장
사용 방법	`@Transactional` + 트랜잭션 매니저
전파 속성	트랜잭션 경계 간 관계 제어
롤백 조건	기본은 `RuntimeException` / 커스터마이징 가능
격리 수준	병행 처리 시 데이터 보호 수준 설정
실무 전략	public, 외부 호출, rollbackFor 등 주의