

## 9. 스프링 AOP 개념

#인강/6.핵심 원리 - 고급편/강의#

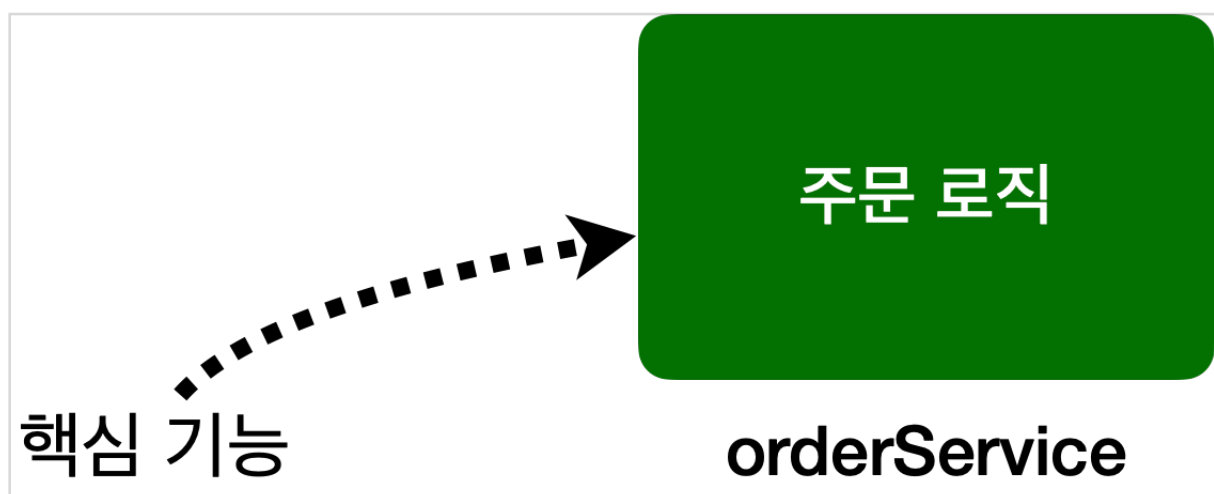
- 9. 스프링 AOP - AOP 소개 - 핵심 기능과 부가 기능
- 9. 스프링 AOP - AOP 소개 - 애스펙트
- 9. 스프링 AOP - AOP 적용 방식
- 9. 스프링 AOP - AOP 용어 정리
- 9. 스프링 AOP 개념 - 정리

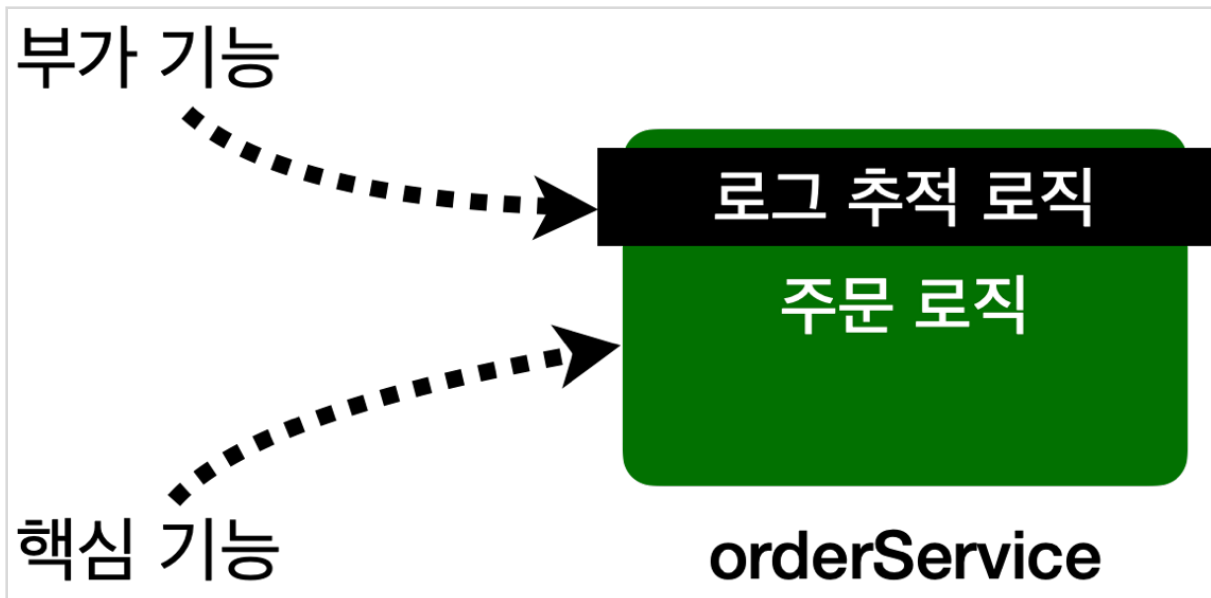
### AOP 소개 - 핵심 기능과 부가 기능

#### 핵심 기능과 부가 기능

애플리케이션 로직은 크게 **핵심 기능**과 **부가 기능**으로 나눌 수 있다.

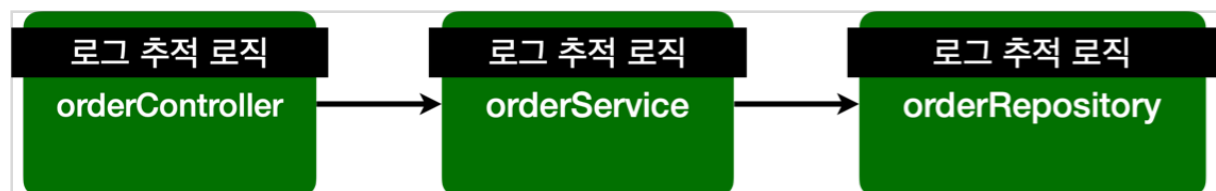
- **핵심 기능**은 해당 객체가 제공하는 고유의 기능이다. 예를 들어서 `OrderService`의 핵심 기능은 주문 로직이다.
- **부가 기능**은 핵심 기능을 보조하기 위해 제공되는 기능이다. 예를 들어서 로그 추적 로직, 트랜잭션 기능이 있다. 이러한 부가 기능은 단독으로 사용되지 않고, 핵심 기능과 함께 사용된다. 예를 들어서 로그 추적 기능은 어떤 핵심 기능이 호출되었는지 로그를 남기기 위해 사용한다. 그러니까 부가 기능은 이름 그대로 핵심 기능을 보조하기 위해 존재한다.



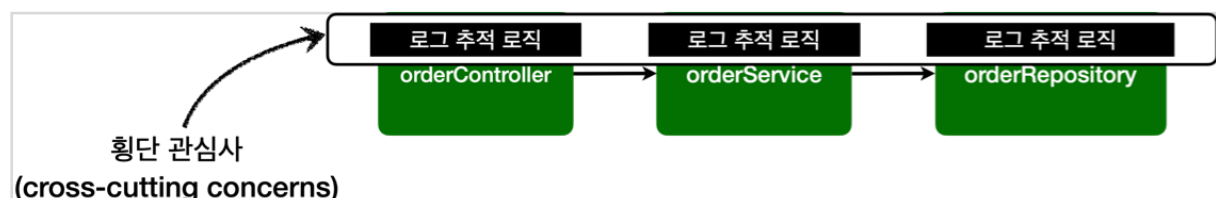


주문 로직을 실행하기 직전에 로그 추적 기능을 사용해야 하면, 핵심 기능인 주문 로직과 부가 기능인 로그 추적 로직이 하나의 객체 안에 섞여 들어가게 된다. 부가 기능이 필요한 경우 이렇게 둘을 합해서 하나의 로직을 완성한다. 이제 주문 서비스를 실행하면 핵심 기능인 주문 로직과 부가 기능인 로그 추적 로직이 함께 실행된다.

여러 곳에서 공통으로 사용하는 부가 기능



보통 부가 기능은 여러 클래스에 걸쳐서 함께 사용된다. 예를 들어서 모든 애플리케이션 호출을 로깅 해야 하는 요구사항을 생각해보자. 이러한 부가 기능은 횡단 관심사(cross-cutting concerns)가 된다. 쉽게 이야기해서 하나의 부가 기능이 여러 곳에 동일하게 사용된다는 뜻이다.



부가 기능 적용 문제

그런데 이런 부가 기능을 여러 곳에 적용하려면 너무 번거롭다. 예를 들어서 부가 기능을 적용해야 하는 클래스가 100개면 100개 모두에 동일한 코드를 추가해야 한다.

부가 기능을 별도의 유틸리티 클래스로 만든다고 해도, 해당 유틸리티 클래스를 호출하는 코드가 결국 필요하다. 그리고 부가 기능이 구조적으로 단순 호출이 아니라 try~catch~finally 같은 구조가

필요하다면 더욱 복잡해진다. (예: 실행 시간 측정)

더 큰 문제는 수정이다. 만약 부가 기능에 수정이 발생하면, 100개의 클래스 모두를 하나씩 찾아가면서 수정해야 한다. 여기에 추가로 부가 기능이 적용되는 위치를 변경한다면 어떻게 될까? 예를 들어서 부가 기능을 모든 컨트롤러, 서비스, 리포지토리에 적용했다가, 로그가 너무 많이 남아서 서비스 계층에만 적용한다고 수정해야하면 어떻게 될까? 또 수 많은 코드를 고쳐야 할 것이다.

**부가 기능 적용 문제를 정리하면 다음과 같다.**

- 부가 기능을 적용할 때 아주 많은 반복이 필요하다.
- 부가 기능이 여러 곳에 퍼져서 중복 코드를 만들어낸다.
- 부가 기능을 변경할 때 중복 때문에 많은 수정이 필요하다.
- 부가 기능의 적용 대상을 변경할 때 많은 수정이 필요하다.

소프트웨어 개발에서 변경 지점은 하나가 될 수 있도록 잘 모듈화 되어야 한다. 그런데 부가 기능처럼 특정 로직을 애플리케이션 전반에 적용하는 문제는 일반적인 OOP 방식으로 해결이 어렵다.

## AOP 소개 - 애스펙트

### 핵심 기능과 부가 기능을 분리

누군가는 이러한 부가 기능 도입의 문제점들을 해결하기 위해 오랜기간 고민해왔다.

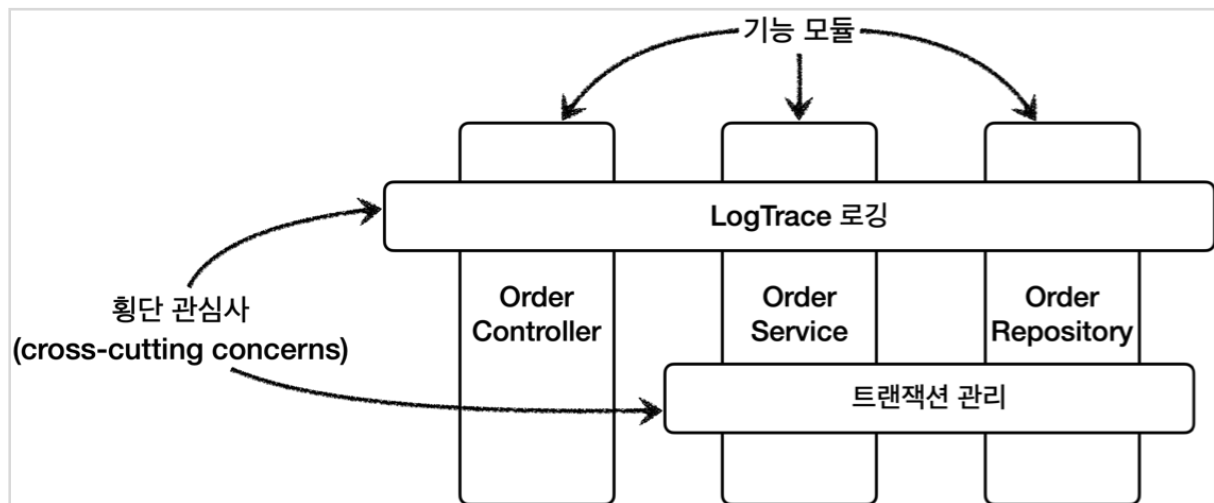
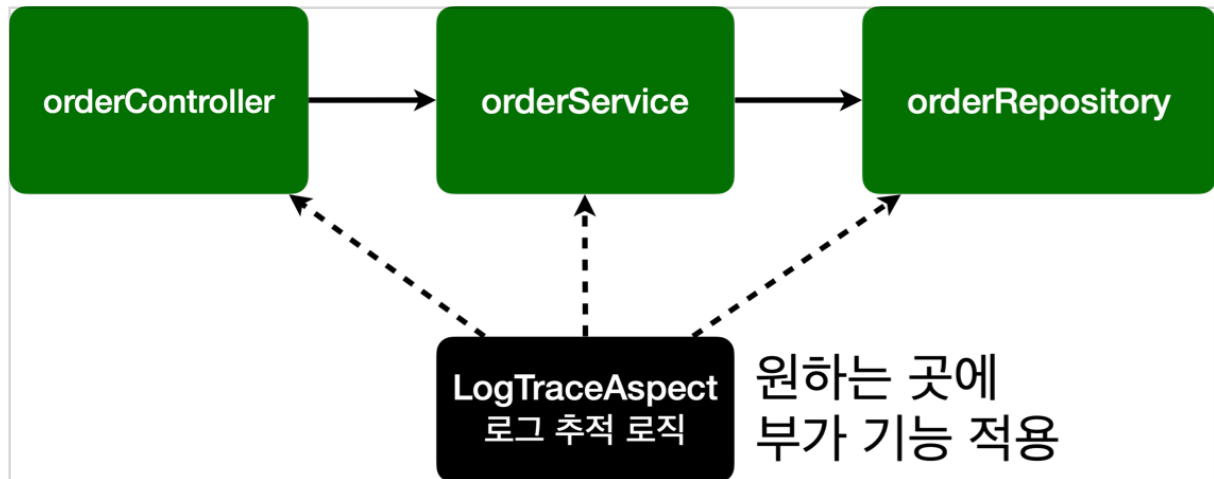
그 결과 부가 기능을 핵심 기능에서 분리하고 한 곳에서 관리하도록 했다. 그리고 해당 부가 기능을 어디에 적용할지 선택하는 기능도 만들었다. 이렇게 부가 기능과 부가 기능을 어디에 적용할지 선택하는 기능을 합해서 하나의 모듈로 만들었는데 이것이 바로 애스펙트(aspect)이다. 애스펙트는 쉽게 이야기해서 부가 기능과, 해당 부가 기능을 어디에 적용할지 정의한 것이다. 예를 들어서 로그 출력 기능을 모든 컨트롤러에 적용해라 라는 것이 정의되어 있다.

그렇다 바로 우리가 이전에 알아본 `@Aspect` 바로 그것이다. 그리고 스프링이 제공하는 어드바이저도 어드바이스(부가 기능)과 포인트컷(적용 대상)을 가지고 있어서 개념상 하나의 애스펙트이다.

애스펙트는 우리말로 해석하면 관점이라는 뜻인데, 이름 그대로 애플리케이션을 바라보는 관점을 하나하나의 기능에서 횡단 관심사(cross-cutting concerns) 관점으로 달리 보는 것이다.

이렇게 애스펙트를 사용한 프로그래밍 방식을 관점 지향 프로그래밍 **AOP(Aspect-Oriented Programming)**이라 한다.

참고로 **AOP는 OOP를 대체하기 위한 것이 아니라 횡단 관심사를 깔끔하게 처리하기 어려운 OOP의 부족한 부분을 보조하는 목적으로 개발되었다.**



### AspectJ 프레임워크

AOP의 대표적인 구현으로 AspectJ 프레임워크(<https://www.eclipse.org/aspectj/>)가 있다. 물론 스프링도 AOP를 지원하지만 대부분 AspectJ의 문법을 차용하고, AspectJ가 제공하는 기능의 일부만 제공한다.

AspectJ 프레임워크는 스스로를 다음과 같이 설명한다.

- 자바 프로그래밍 언어에 대한 완벽한 관점 지향 확장
- 횡단 관심사의 깔끔한 모듈화
  - 오류 검사 및 처리
  - 동기화
  - 성능 최적화(캐싱)
  - 모니터링 및 로깅

## AOP 적용 방식

AOP를 사용하면 핵심 기능과 부가 기능이 코드상 완전히 분리되어서 관리된다.

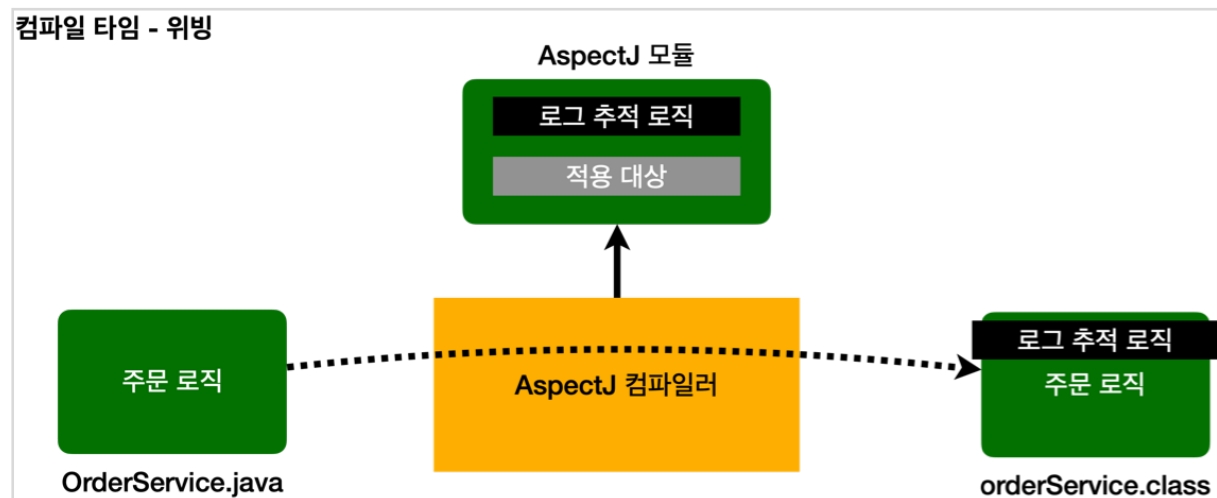
그렇다면 AOP를 사용할 때 부가 기능 로직은 어떤 방식으로 실제 로직에 추가될 수 있을까?

크게 3가지 방법이 있다.

- 컴파일 시점
- 클래스 로딩 시점
- 런타임 시점(프록시)

### 컴파일 시점

컴파일 타임 - 위빙



.java 소스 코드를 컴파일러를 사용해서 .class 를 만드는 시점에 부가 기능 로직을 추가할 수 있다.

이때는 AspectJ가 제공하는 특별한 컴파일러를 사용해야 한다. 컴파일 된 .class 를 디컴파일 해보면 애스펙트 관련 호출 코드가 들어간다. 이해하기 쉽게 풀어서 이야기하면 부가 기능 코드가 핵심 기능이 있는 컴파일된 코드 주변에 실제로 붙어 버린다고 생각하면 된다. AspectJ 컴파일러는 Aspect를 확인해서 해당 클래스가 적용 대상인지 먼저 확인하고, 적용 대상인 경우에 부가 기능 로직을 적용한다.

참고로 이렇게 원본 로직에 부가 기능 로직이 추가되는 것을 위빙(Weaving)이라 한다.

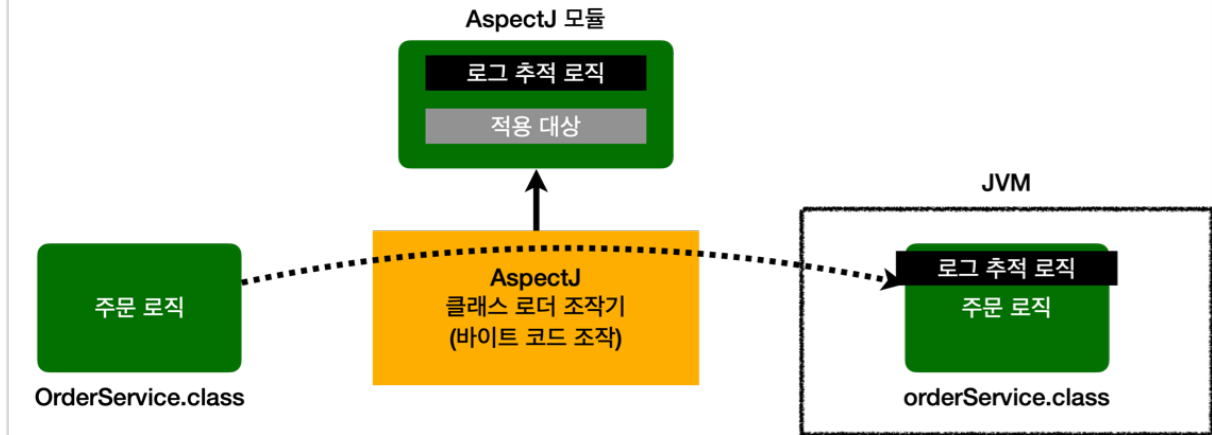
- 위빙(Weaving): 옷감을 짜다. 직조하다. 애스펙트와 실제 코드를 연결해서 붙이는 것

### 컴파일 시점 - 단점

컴파일 시점에 부가 기능을 적용하려면 특별한 컴파일러도 필요하고 복잡하다.

### 클래스 로딩 시점

## 로드 타임 - 위빙

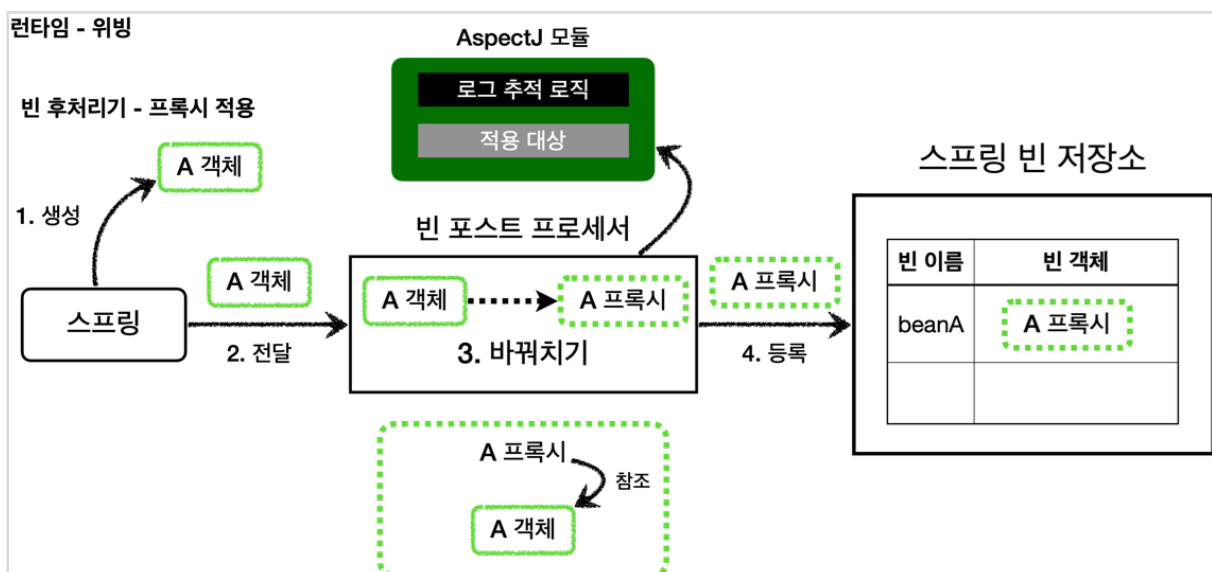


자바를 실행하면 자바 언어는 `.class` 파일을 JVM 내부의 클래스 로더에 보관한다. 이때 중간에서 `.class` 파일을 조작한 다음 JVM에 올릴 수 있다. 자바 언어는 `.class` 를 JVM에 저장하기 전에 조작할 수 있는 기능을 제공한다. 궁금한 분은 java Instrumentation를 검색해보자. 참고로 수 많은 모니터링 툴들이 이 방식을 사용한다. 이 시점에 애스펙트를 적용하는 것을 로드 타임 위빙이라 한다.

## 클래스 로딩 시점 - 단점

로드 타임 위빙은 자바를 실행할 때 특별한 옵션( `java -javaagent` )을 통해 클래스 로더 조작기를 지정해야 하는데, 이 부분이 번거롭고 운영하기 어렵다.

## 런타임 시점



런타임 시점은 컴파일도 다 끝나고, 클래스 로더에 클래스도 다 올라가서 이미 자바가 실행되고 난 다음을 말한다. 자바의 메인( `main` ) 메서드가 이미 실행된 다음이다. 따라서 자바 언어가 제공하는 범위 안에서 부가 기능을 적용해야 한다. 스프링과 같은 컨테이너의 도움을 받고 프록시와 DI, 빈 포스트 프로세서 같은 개념들을 총 동원해야 한다. 이렇게 하면 최종적으로 프록시를 통해 스프링 빈에 부가 기능을 적용할 수 있다. 그렇다. 지금까지 우리가 학습한 것이 바로 프록시 방식의 AOP이다.

프록시를 사용하기 때문에 AOP 기능에 일부 제약이 있다. 하지만 특별한 컴파일러나, 자바를 실행할 때 복잡한 옵션과 클래스 로더 조작기를 설정하지 않아도 된다. 스프링만 있으면 얼마든지 AOP를 적용할 수

있다.

**부가 기능이 적용되는 차이를 정리하면 다음과 같다.**

- 컴파일 시점: 실제 대상 코드에 애스팩트를 통한 부가 기능 호출 코드가 포함된다. AspectJ를 직접 사용해야 한다.
- 클래스 로딩 시점: 실제 대상 코드에 애스팩트를 통한 부가 기능 호출 코드가 포함된다. AspectJ를 직접 사용해야 한다.
- 런타임 시점: 실제 대상 코드는 그대로 유지된다. 대신에 프록시를 통해 부가 기능이 적용된다. 따라서 항상 프록시를 통해야 부가 기능을 사용할 수 있다. 스프링 AOP는 이 방식을 사용한다.

### AOP 적용 위치

AOP는 지금까지 학습한 메서드 실행 위치 뿐만 아니라 다음과 같은 다양한 위치에 적용할 수 있다.

- 적용 가능 지점(조인 포인트): 생성자, 필드 값 접근, static 메서드 접근, 메서드 실행
  - 이렇게 AOP를 적용할 수 있는 지점을 조인 포인트(Join point)라 한다.
- AspectJ를 사용해서 컴파일 시점과 클래스 로딩 시점에 적용하는 AOP는 바이트코드를 실제 조작하기 때문에 해당 기능을 모든 지점에 다 적용할 수 있다.
- 프록시 방식을 사용하는 스프링 AOP는 메서드 실행 지점에만 AOP를 적용할 수 있다.
  - 프록시는 메서드 오버라이딩 개념으로 동작한다. 따라서 생성자나 static 메서드, 필드 값 접근에는 프록시 개념이 적용될 수 없다.
  - 프록시를 사용하는 스프링 **AOP의 조인 포인트는 메서드 실행으로 제한된다.**
- 프록시 방식을 사용하는 스프링 AOP는 스프링 컨테이너가 관리할 수 있는 스프링 빈에만 **AOP를 적용**할 수 있다.

### 참고

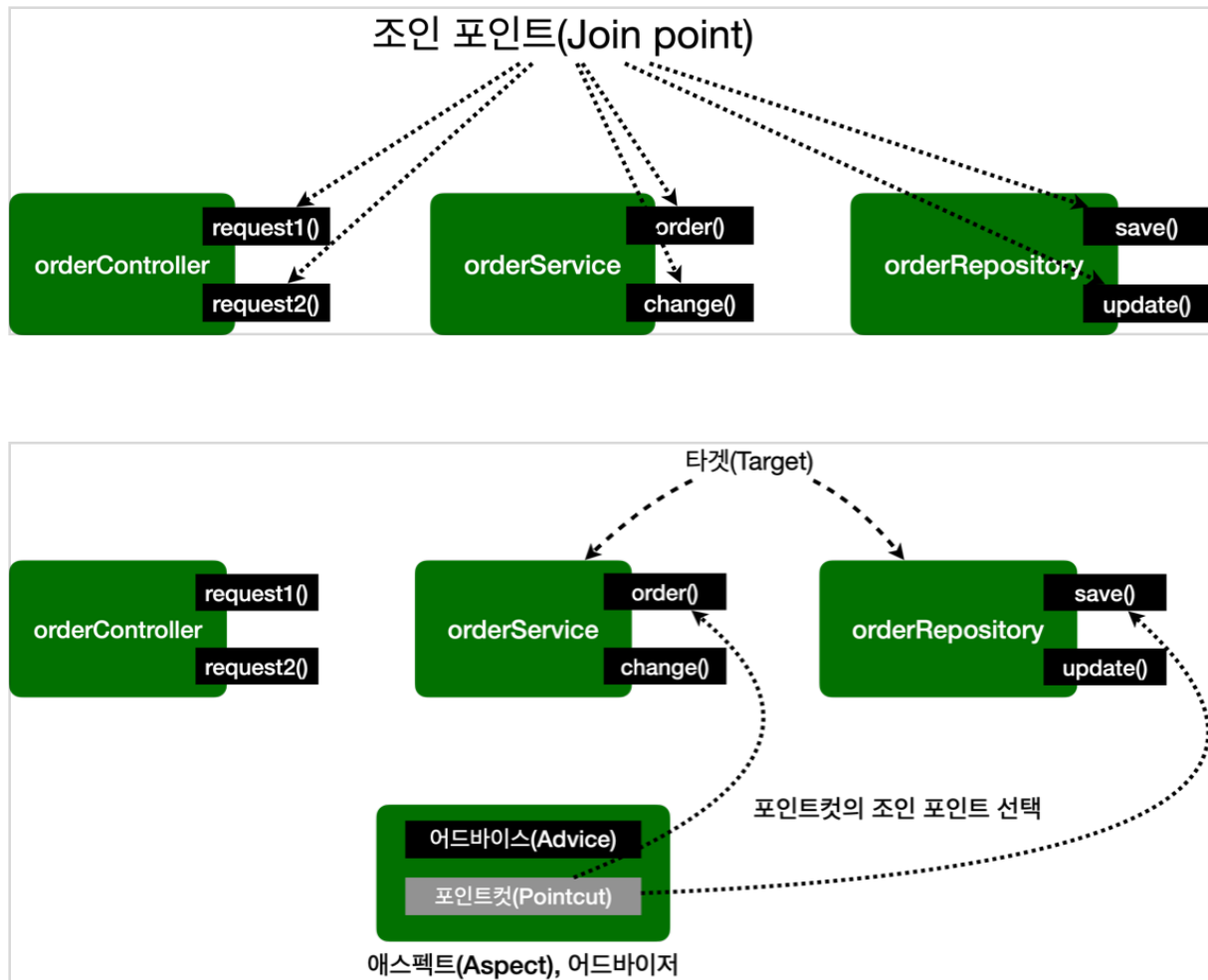
스프링은 AspectJ의 문법을 차용하고 프록시 방식의 AOP를 적용한다. AspectJ를 직접 사용하는 것이 아니다.

### 중요

스프링이 제공하는 AOP는 프록시를 사용한다. 따라서 프록시를 통해 메서드를 실행하는 시점에만 AOP가 적용된다. AspectJ를 사용하면 앞서 설명한 것 처럼 더 복잡하고 더 다양한 기능을 사용할 수 있다. 그렇다면 스프링 AOP 보다는 더 기능이 많은 AspectJ를 직접 사용해서 AOP를 적용하는 것이 더 좋지 않을까?

AspectJ를 사용하려면 공부할 내용도 많고, 자바 관련 설정(특별한 컴파일러, AspectJ 전용 문법, 자바 실행 옵션)도 복잡하다. 반면에 스프링 AOP는 별도의 추가 자바 설정 없이 스프링만 있으면 편리하게 AOP를 사용할 수 있다. 실무에서는 스프링이 제공하는 AOP 기능만 사용해도 대부분의 문제를 해결할 수 있다. 따라서 스프링 AOP가 제공하는 기능을 학습하는 것에 집중하자.

## AOP 용어 정리



- **조인 포인트(Join point)**
  - 어드바이스가 적용될 수 있는 위치, 메소드 실행, 생성자 호출, 필드 값 접근, static 메서드 접근 같은 프로그램 실행 중 지점
  - 조인 포인트는 추상적인 개념이다. AOP를 적용할 수 있는 모든 지점이라 생각하면 된다.
  - 스프링 AOP는 프록시 방식을 사용하므로 조인 포인트는 항상 메소드 실행 지점으로 제한된다.
- **포인트컷(Pointcut)**
  - 조인 포인트 중에서 어드바이스가 적용될 위치를 선별하는 기능
  - 주로 AspectJ 표현식을 사용해서 지정
  - 프록시를 사용하는 스프링 AOP는 메서드 실행 지점만 포인트컷으로 선별 가능
- **타겟(Target)**
  - 어드바이스를 받는 객체, 포인트컷으로 결정
- **어드바이스(Advice)**
  - 부가 기능
  - 특정 조인 포인트에서 Aspect에 의해 취해지는 조치
  - Around(주변), Before(전), After(후)와 같은 다양한 종류의 어드바이스가 있음
- **어스펙트(Aspect)**



- 어드바이스 + 포인트컷을 모듈화 한 것
- `@Aspect` 를 생각하면 됨
- 여러 어드바이스와 포인트 컷이 함께 존재
- **어드바이저(Advisor)**
  - 하나의 어드바이스와 하나의 포인트 컷으로 구성
  - 스프링 AOP에서만 사용되는 특별한 용어
- **위빙(Weaving)**
  - 포인트컷으로 결정한 타겟의 조인 포인트에 어드바이스를 적용하는 것
  - 위빙을 통해 핵심 기능 코드에 영향을 주지 않고 부가 기능을 추가 할 수 있음
  - AOP 적용을 위해 애스펙트를 객체에 연결한 상태
    - 컴파일 타임(AspectJ compiler)
    - 로드 타임
    - 런타임, 스프링 AOP는 런타임, 프록시 방식
- **AOP 프록시**
  - AOP 기능을 구현하기 위해 만든 프록시 객체, 스프링에서 AOP 프록시는 JDK 동적 프록시 또는 CGLIB 프록시이다.

정리