# week 3

## AOP가 필요한 상황

앱의 모든 메서드의 실행 시간을 측정해야 함

```
public Long join(Member member) {
    long start = System.currentTimeMillis();
    try {
        validateDuplicateMember(member);
        memberRepository.save(member);
        return member.getId();
    } finally {
        long finish = System.currentTimeMillis();
        long timeMs = finish - start;
        System.out.println("join " + timeMs + "ms");
    }
}

public List<Member> findMembers() {
    long start = System.currentTimeMillis();
    try {
        return memberRepository.findAll();
    } finally {
        long finish = System.currentTimeMillis();
        long timeMs = finish - start;
        System.out.println("findMembers " + timeMs + "ms");
    }
}
```

위 방식처럼, 모든 함수에 시간을 측정하는 로직을 넣을 수 있겠지만, 함수의 개수가 많아진다면 작업하는 데 비용이 많이 듦.
또한 시간을 측정하는 방식에 변경이 생긴다면 모든 함수를 수정해야함 → 유지보수가 어려워짐

이러한 요구사항을 효율적으로 해결하기 위해서, 핵심 로직과 공통 로직을 분리, 모듈화하는 AOP를 적용

# AOP

Aspect oriented programming

공통 관심 사함, 핵심 관심 사항 분리

비즈니스 로직을 기준으로 핵심, 부가적인 로직으로 나누어 각각 모듈화

- target: aspect가 적용되는 곳
- advice: aspect의 기능에 대한 구현체
- join point: advice가 target에 적용되는 시점
- point cut: join point의 집합

## Aspect

```
@Aspect
public class LogAop {
    ...
}
```

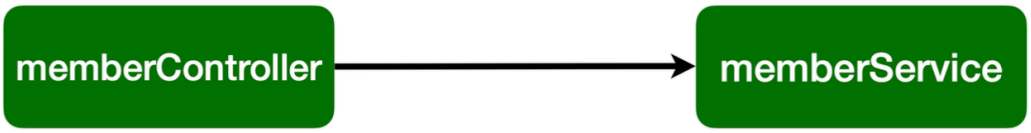
해당 클래스가 AOP클래스로 사용하는 클래스임을 알려주는 어노테이션
해당 어노테이션이 부여되었다고 해서 자동으로 빈으로 등록되는 것이 아니기 때문에 따로 등록해줘야 함

## Around

```
@Aspect
public class LogAop {
    @Around("execution(* com.java.ex.Example(..))")
    public Object logging(ProceedingJoinPoint joinPoint) throws Throwable {
        try {
            ...
            return joinPoint.proceed();
        } finally {
            ...
        }
}
```

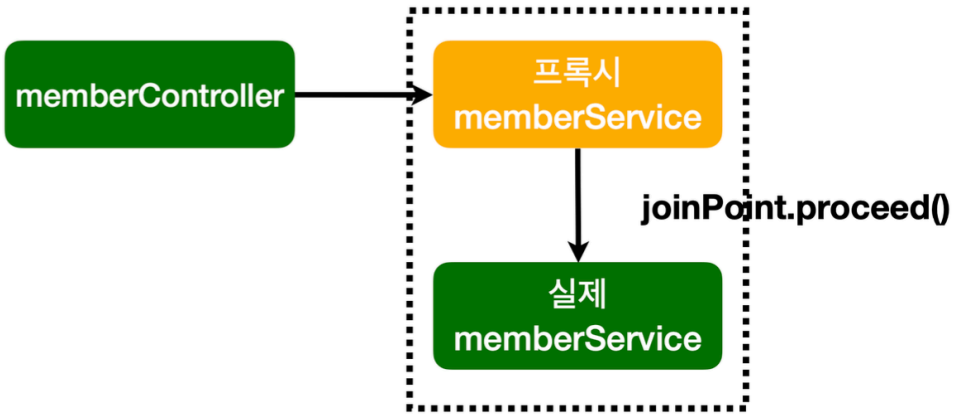
비즈니스 로직의 실패 여부와 관계 없이 전후로 실행되도록 하는 advice

# 스프링 컨테이너




- `memberController`는 `memberService`의 기능들을 사용하며 의존관계가 수립
- 컨트롤러에서는 `memberServce`의 실제 객체에 접근하여 로직을 수행
- `memberService` 에서 공통적인 기능들을 추가하려면 각각의 메서드에 기능들을 추가

# 스프링 컨테이너



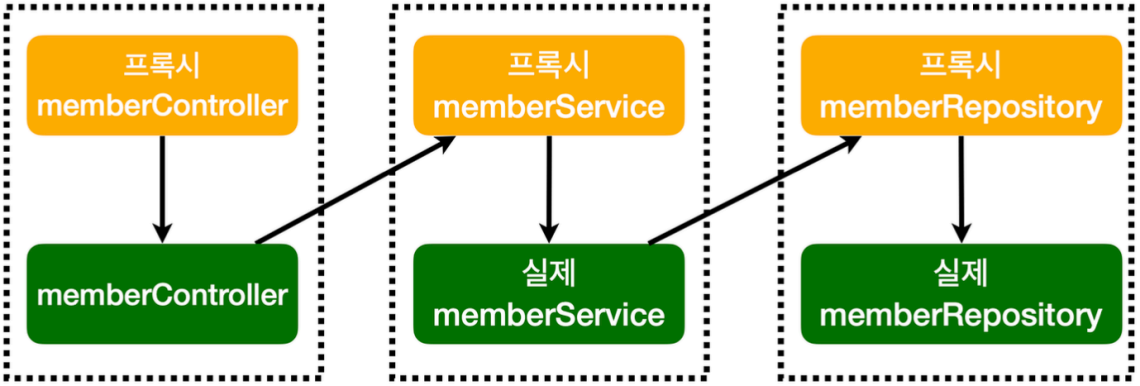

- `memberController`와 `memberService`간에 프록시 객체인 `memberServiceProxy`가 추가
- `memberController`는 실제 객체에 바로 접근하지 않고 프록시 객체를 거쳐 접근하여 로직 수행

## 스프링 컨테이너


- `Client`가 `Request`를 하였을 때 스프링 컨테이너 내부의 의존관계를 보여줌
- `memberController` 가 `memberService` 에 메서드를 호출해 로직을 수행.

  `memberService`에서는 `memberRepository` 를 호출하여 저장소는 DB에 접근해 데이터를 조회, 수정, 삭제함
- 모두 어딘가를 거치지 않고 실제 인스턴스에 접근

## 스프링 컨테이너




- 모든 객체들에 대해 프록시 객체가 생성되고 의존관계도 프록시 객체를 통해 이뤄짐
- 각각 객체의 기능을 수행하여 메서드를 호출하면 요청을 프록시 객체가 받아들여 로직 전, 후로 추가적인 작업을 수행