Spring AOP

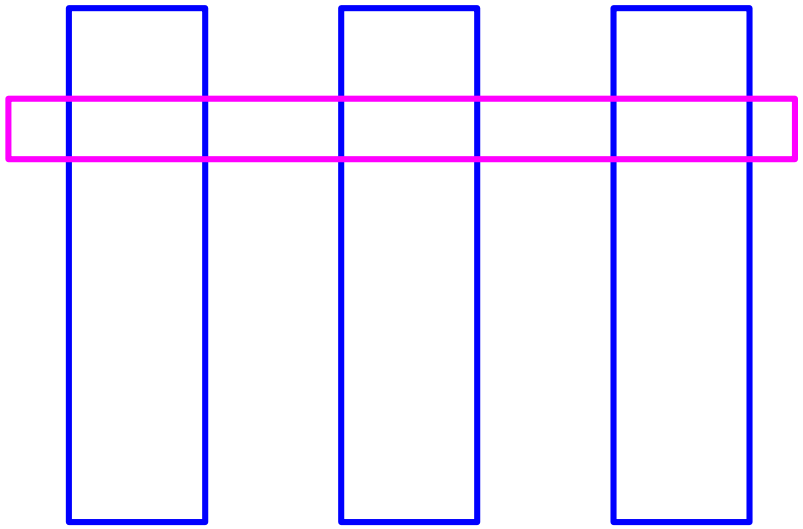
558280-1
2026년 봄학기
5/28/2026
박경신

Spring AOP

- Aspect-oriented programming (AOP)
  - 관점 지향 프로그래밍
  - 쉽게 말해서, 공통된 부분을 좀 더 쉽게 프로그래밍하는 기법
  - 특히 Java 웹 개발에서 매우 일반적인 기술
- 로깅 기능을 매출/유저 관리에 적용하려면?
  - Logger를 상속하여 SalesLogger, UserMgmtLogger를 만듦
- 이 상태에서 에러 로깅을 추가하려면?
  - SalesErrorLogger, UserMgmtErrorLogger를 자식 클래스로 구현

AOP

- 공통 관심 사항을 프로그래밍하여 이 모듈을 여러 코드에 짜넣음 (Weave)
- AspectJ 등으로 구현되어 이미 자바 개발에 사용되어 왔음
- 횡단 관점이란, 각각 다른 핵심 로직 메소드들을 공통 관심 기능 코드(e.g. logging, security, transaction)가 횡단한다는 개념



기존의 구현법 vs AOP

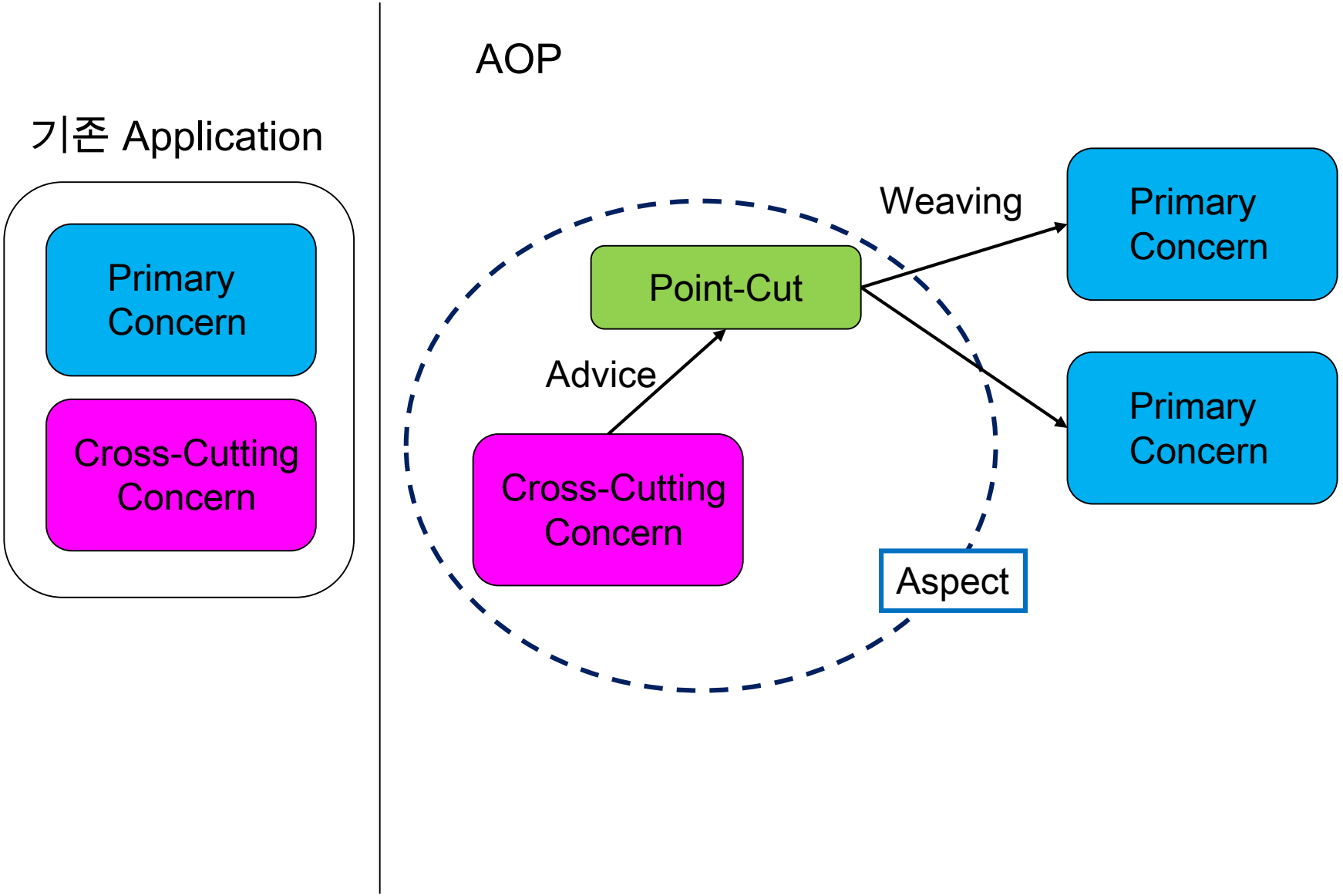
- 기존 구현법 - 공통 기능이 핵심 로직의 밖에 구현되어 호출됨
  - log(); func(); clear();
- AOP - 공통 기능이 밖에 구현되는 것은 같으나 별도의 호출 절차가 등장하지 않음
  - func();

기존의 구현법 vs AOP

```
// 기존 구현
log.info("start");
doSomething();
log.info("end");

// AOP
@Around("execution(* com.example.service.*.*(..))")
public Object logExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {
    long start = System.currentTimeMillis();
    Object result = joinPoint.proceed();
    long end = System.currentTimeMillis();
    System.out.println("Executed in: " + (end - start) + "ms");
    return result;
}
```

기존의 구현법 vs AOP



AOP 기본 용어

- Aspect (Pointcut + Advice)
  - **공통적으로 관심 있는 기능**들 모듈화
- Target
  - Aspect가 적용될 **대상**을 의미하며, 메소드, 클래드 등이 해당됨
- Advice
  - @Before, @After, @Around, @AfterReturning, @AfterThrowing
  - **언제** 어떤 공통 관심 기능(Cross-cutting function)을 핵심 로직(Core concern)에 **적용할지** 정의 – 거래 시도 전 로깅 기능 활성
- JoinPoint
  - **Advice가 적용될 수 있는 지점** – method, field 등
- Pointcut
  - 특정 조건에 의해 **필터링된 JoinPoint** – 특정 메소드에서만 수행 등
- Weaving
  - Pointcut으로 지정된 관심 메소드가 호출될 때 Advice 메소드를 핵심 로직(Core concern)에 적용하는 것

Weaving 방식들

- 컴파일 시 위빙(Weaving)
  - AspectJ 기본
    - 바이트코드 수준에서 직접적인 변경을 수행하여 정적 성능 최적화 가능
- 클래스 로딩 시 위빙
  - 바이트코드 수준에서 코드를 위빙
- 런타임 시 위빙
  - Proxy를 사용하여 실제 core concern 객체 호출
    전후(Before/After)에 cross-cutting concern 객체 호출
  - Spring AOP는 런타임 시 위빙하는 Proxy-based AOP 지원
    - 실행 중에 Spring 컨테이너가 빈(Bean) 생성시 프록시를 동적으로 생성하고 삽입

AspectJ 예시

```
@Aspect
public class LoggingAspect {
    @Before("execution(* com.example.service..*(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println("[AspectJ Before] " + joinPoint.getSignature().getName());
    }
    @AfterReturning(pointcut = "execution(* com.example.service..*(..))", returning
= "result")
    public void logAfterReturning(JoinPoint joinPoint, Object result) {
        System.out.println("[AspectJ AfterReturning] " +
                        joinPoint.getSignature().getName() + " returned: " + result);
    }
    @AfterThrowing(pointcut = "execution(* com.example.service..*(..))", throwing =
"ex")
    public void logException(JoinPoint joinPoint, Throwable ex) {
        System.out.println("[AspectJ Exception] " +
                        joinPoint.getSignature().getName() + " threw: " + ex);
    }
}
```

스프링 AOP

- 프록시 기반의 AOP 지원
  - 외부에서 메소드 호출시에만 AOP 적용 가능
- 자바이므로 AspectJ 사용하여 컴파일/로딩시 AOP 가능
- 설정 정보를 이용해서 어떤 빈 객체에 프록시를 씌울 지 결정

Proxy 클래스 예시 (Spring AOP 내부 동작)

```
public class $Proxy0 implements MyService {
    private MyService target;
    public $Proxy0(MyService target) {
        this.target = target;
    }
    public void doSomething() {
        System.out.println("[Before Advice 실행]");
        target.doSomething();
        System.out.println("[After Advice 실행]");
    }
}
```

Client

↓

Proxy 객체 생성

↓

Before Advice 실행

↓

Target 객체 실제 메소드 호출

↓

AfterReturning or AfterAdvice 실행

↓

결과 반환

구현 가능한 Advice 종류

- @Before
  - 대상 메소드가 실행되기 전에 Advice를 실행
- @After
  - 대상 메소드가 실행된 후에 무조건 Advice 실행
- @AfterReturning
  - 대상 메소드가 정상적으로 실행되고 반환된 후에 Advice 실행
- @AfterThrowing
  - 대상 메소드에서 예외 처리 후에 Advice 실행
- @Around
  - 대상 메소드 실행 전, 후 또는 예외가 발생 시 Advice 실행

Spring AOP 구현

- 스프링 AOP 의존 추가
- 공통 기능 (Cross-cutting concern)을 구현
  - Aspect
    - AOP로 정의하는 클래스를 지정
  - Pointcut
    - AOP 기능을 메소드, Annotation 등 어디에 적용시킬지 지점을 설정
  - 공통 기능을 구현한 메소드에 원하는 시점에 따라 Advice를 택하여 아래 주석을 적용함
    - Before
    - After
    - AfterReturning
    - AfterThrowing
    - Around

Spring Boot AOP dependency 추가

- pom.xml 에 Spring AOP 추가

```
<!-- Spring AOP -->
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

Aspect 구현하기

- @Aspect 사용하여 Spring AOP를 적용할 자바 클래스를 지정함
- 빈을 자바 주석으로 설정하는 것과 유사함
  - @Config ->@Aspect
- 다만 Advisor(Pointcut+Advice)가 Aspect 클래스 안에 포함됨

```
@Aspect
@Component
public class ServiceExecutionTimeAspect {
...
```

Advice 구현

- Around advice(메소드 실행 전후, 또는 익셉션 발생시) 구현해보기
  - 어떤 Pointcut이 실행될 때 실행에 소비되는 시간을 체크하기 위한 advice

```
public Object trace(ProceedingJoinPoint joinPoint) throws Throwable {
        long start = System.currentTimeMillis();

        Object result = joinPoint.proceed();
        long executionTime = System.currentTimeMillis() - start;

        System.out.println(joinPoint.getSignature() + " executed in " +
executionTime + "ms");
        return result;
}
```

  - advice에는 시점(around? before?)이 명시되어 있지 않음
  - 인자로 전달되는 joinPoint가 중요함

@Around 설정 추가

Advice 정의

Pointcut 대상

- kr.ac.dankook.ace.springbootaop.service 패키지의 모든 메소드에 대해 **Around** advice로 **trace** 메소드 실행

```
@Around("execution(* kr.ac.dankook.ace.springbootaop.service.*.*(..))")
public Object trace(ProceedingJoinPoint joinPoint) throws Throwable {
        long start = System.currentTimeMillis();

        Object result = joinPoint.proceed();
        long executionTime = System.currentTimeMillis() - start;

        System.out.println(joinPoint.getSignature() + " executed in " + executionTime
+ "ms");
        return result;
}
```

  - **Advisor(Pointcut+Advice)**를 만들어 빈으로 등록하면, 자동 프록시 생성기가 등록된 Advisor를 찾고, 스프링 빈들에 자동으로 Pointcut이 매칭되는 경우 프록시를 적용해줌

Aspect 실행해보기

- 아무 Service 빈 이용해서 실행해보기

```
@Service
public class TaskService {
    public String performTask(String taskName) {
        System.out.println("TaskService.performTask(String) ...");
        return "Task " + taskName + " Completed";
    }
}
@RestController
public class TaskController {
    @Autowired
    private TaskService service;
    @GetMapping("/task/{name}")
    public String task(@PathVariable String name) {
        return service.performTask(name);
    }
}
```

```
TaskService.performTask(String) ...
String kr.ac.dankook.ace.springbootaop.service.TaskService.performTask(String)
executed in 1ms
```

결과

- Service 클래스들에는 아무 변화를 주지않고
  - 핵심 로직 코드를 수정하지 않고
- 메소드 실행 프로파일을 출력하는 기능을 추가
  - 공통 기능을 추가함
- 이후 원하는 메소드들에 대해 Aspect 설정만 추가하면 추가적인 코드 수정 없이 프로파일링이 가능해짐

Advice 타입별 클래스 작성법 : Before

- @Before advice 메소드는 대개 리턴이 void
- advice가 먼저 실행되므로 advice에서 예외 발생시 pointcut의 메소드가 실행되지 않음
- 사용 예시 - 어떤 동작에 대해 접근 권한이 있는지 확인해서 적합하지 않는 경우 예외처리, 동작 실행하지 않음

```
@Before("execution(* kr.ac.dankook.ace.springbootaop.controller.*.*(..))")
public void logBefore(JoinPoint joinPoint) {
        log.info("Before: " + joinPoint.getSignature().getName());
}
```

Advice 타입별 클래스 작성법 : After

- @After advice 메소드
- 실행 성공/실패 여부에 관계없이 언제나 사용

```
@After("execution(* kr.ac.dankook.ace.springbootaop.controller.*.*(..))")
public void logAfter(JoinPoint joinPoint) {
        log.info("After: " + joinPoint.getSignature().getName());
}
```

Advice 타입별 클래스 작성법 : After Returning

- @AfterReturning advice 메소드
- pointcut의 리턴값을 이용해 어떤 동작을 수행하고 싶은 경우 **returning** 속성 사용

```
@AfterReturning(pointcut = "execution(*
kr.ac.dankook.ace.springbootaop.controller.*.*(..))", returning = "result")
public void logAfterReturning(JoinPoint joinPoint, Object result) {
        log.info("AfterReturning: " + joinPoint.getSignature().getName() + "
result: " + result);
}
```

Advice 타입별 클래스 작성법 : After Throwing

- @AfterThrowing advice 메소드
- AfterReturning과 동일하나 exception 발생시만 실행
- 예외 정보를 받기 위해 **throwing** 속성 사용

```
@AfterThrowing(pointcut = "execution(*
kr.ac.dankook.ace.springbootaop.controller.*.*(..))", throwing = "e")
public void logAfterThrowing(JoinPoint joinPoint, Throwable e) {
        log.info("AfterThrowing: " + joinPoint.getSignature().getName() + "
exception: " + e.getMessage());
}
```

- 예외 발생시 정보 수집이나 로깅을 위해 사용

Advice 타입별 클래스 작성법 : Around

- @Around advice 메소드
- 구현 방식에 따라 앞서 설명한 모든 advice 종류 구현 가능
- 가장 범용적
- 반드시 첫번째 매개변수로 **ProceedingJoinPoint** 타입을 지정해야 함

```
@Around("execution(* kr.ac.dankook.ace.springbootaop.controller.*.*(..))")
public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
        log.info("Around before: " + joinPoint.getSignature().getName());
        Object result = joinPoint.proceed();
        log.info("Around after: " + joinPoint.getSignature().getName());
        return result;
}
```

Around advice 예제

- 캐시 사용하기
- 어떤 데이터베이스에 쿼리를 날리기 전에 캐시 메모리에 해당 데이터가 있는지를 확인해서
  - 캐시에 있다면 - 캐시에서 바로 데이터 반환
  - 캐시에 없다면 - 쿼리 후 캐시에 해당 데이터 추가

```
@Component
public class PersonCache {
    private Map<Long, Person> cache = new HashMap<>();
    public Person getFromCache(Long id) {
        System.out.println("PersonCache.getFromCache() Fetching person data...");
        return cache.get(id);
    }
    public void putToCache(Person person) {
        System.out.println("PersonCache.putToCache() Saving person data...");
        cache.put(person.getId(), person);
    }
}
```

Around advice 예제

- service.PeopleService.findById에 Around advice로 구현

```
@Aspect
@Component
public class PersonCacheAspect {
    @Autowired
    private PersonCache personCache;
    @Around("execution(*
kr.ac.dankook.ace.springbootaop.service.PeopleService.findById(..))")
    public Object cacheObject(ProceedingJoinPoint joinPoint) throws Throwable {
        Object[] args = joinPoint.getArgs();
        Long id = (Long) args[0];
        Person cachedPerson = personCache.getFromCache(id);
        if (cachedPerson != null) {
            System.out.println("Returning person from cache: " + id);
            return cachedPerson;
        }
        Person result = (Person) joinPoint.proceed();
        personCache.putToCache(result);
        System.out.println("Person added to cache: " + id);
        return result;
    }
}
```

Around advice 예제

- 실행 테스트

```
@RestController
@RequiredArgsConstructor
@RequestMapping("/people")
public class PeopleController {
    private final PeopleService service;
    @GetMapping("/{id}")
    public Person getById(@PathVariable ("id") long id) {
        return service.findById(id);
    }
}
```

```
PersonCache.getFromCache() Fetching person data...
PersonCache.putToCache() Saving person data...
Person added to cache: 1
PersonCache.getFromCache() Fetching person data...
PersonCache.putToCache() Saving person data...
Person added to cache: 2
PersonCache.getFromCache() Fetching person data...
Returning person from cache: 1
```

Pointcut 표현식

- Pointcut 표현식을 사용하여 Advice 메소드가 적용될 비지니스 메소드를 정확하게 필터링할 수 있음

  - 위 예시는 controller 패키지에 있는 모든 클래스의 모든 메서드, 그리고 그 메서드의 파라미터 타입과 파라미터 수에 상관 없이 Pointcut을 적용

Pointcut 표현식

- Pointcut 지시자
  - execution – 메소드 실행 JoinPoint를 매칭. 스프링 AOP에서 가장 많이 사용하고, 기능도 복잡
  - within – 특정 타입(클래스) 내의 JoinPoint에 대해 매칭
  - args – 인자가 주어진 타입의 인스턴스인 JoinPoint로 매칭
  - this – 스프링 빈 객체(Spring AOP Proxy)를 대상으로 하는 JoinPoint
  - target – Target 객체를 대상으로 하는 JoinPoint
  - @target – 실행 객체의 클래스에 주어진 타입 주석이 있는 JoinPoint
  - @within – 주어진 주석이 있는 타입 내 JoinPoint
  - @annotation – 주어진 주석을 가지고 있는 JoinPoint를 매칭
  - @args – 전달된 실제 인수의 런타임 타입이 주어진 타입의 주석을 갖는 JoinPoint
  - bean – 스프링 전용 Pointcut 지시자, 빈 이름으로 Pointcut 지정

Advice 타입별 클래스 작성

- 각 Advice 타입별로 @Before, @AfterRunning, @AfterThrowing, @After, @Around 사용
  - @AfterRunning, @AfterThrowing의 경우 각각 returning, throwing 속성 사용 가능

pointcut의 args에 접근 가능

```
@AfterReturning(pointcut = "args(memberId,info)", returning = "result",
argNames="joinPoint,memberId, info,result")
public void traceReturn(JoinPoint joinPoint, String memberId, UpdateInfo
info, boolean result) {
    System.out.printf("[TA] 정보 수정: 대상회원=%s, 수정정보=%s,
결과=%s\n", memberId, info, result);
}
```

JoinPoint 인터페이스

- Advice 메소드를 의미있게 구현하려면, 호출한 비지니스 메소드의 정보가 필요한데, 이때 JoinPoint 인터페이스가 제공하는 메소드를 사용함

Method	Description
Signature getSignature()	호출된 메소드에 대한 정보를 반환
Object[] getArgs()	호출된 메소드에 넘겨준 인자(arguments)를 반환
Object getTarget()	대상 객체를 반환
String getName()	메소드의 이름을 반환
String toLongString()	메소드 리턴 타입, 이름, 매개변수를 패키지 경로까지 포함하여 반환

@Pointcut

- @Pointcut을 이용해 pointcut을 재사용할 수 있음
  - 같은 클래스의 @Pointcut 메소드는 메소드 이름만

```
@Aspect
@Component
public class TaskLoggingAspect {
    @Pointcut("within(kr.ac.dankook.ace.springbootaop.service.TaskService)")
    private void taskServiceMethods() {}

    // Advice that runs before the methods matched by the pointcut
    @Before("taskServiceMethods()")
    public void beforeMethod() {
        System.out.println("A method in TaskService is about to be executed.");
    }
}
```

  - 다른 클래스는 classname.taskServiceMethods()
  - 패키지가 다르면 fully qualified name 사용