

ITEM 39

명명 패턴보다 애너테이션을 사용하라

명명 패턴(NAMING PATTERN)

코드의 메서드나 클래스 이름 자체에 규칙(패턴)을 부여해서
런타임이나 프레임워크가 그 이름을 보고 특정 동작을 하도록 하는 방식

명명 패턴



```
1  import junit.framework.TestCase;
2
3  public class CalculatorTest extends TestCase {
4      public void testAdd() {
5          assertEquals(2, Calculator.add(1, 1));
6      }
7  }
```

테스트 프레임워크인 JUnit은 버전 3까지 테스트 메서드 이름은 반드시 test로 시작해야 했다.



```
1  import org.junit.Test;
2  import static org.junit.Assert.*;
3
4  public class CalculatorTest {
5      @Test
6      public void add() {
7          assertEquals(2, Calculator.add(1, 1));
8      }
9  }
```

JUnit 4부터는 애너테이션 기반으로 전환되면서, 메서드 이름에 test 접두사를 붙일 필요가 없어졌다.

명명 패턴의 단점

1 오타에 민감하다.



```
1 public class CalculatorTest extends TestCase {  
2     // 의도는 add 메서드를 테스트하려고 했지만, 오타가 있음!  
3     public void tsetAdd() {  
4         assertEquals(2, Calculator.add(1, 1));  
5     }  
6 }
```

2 올바른 프로그램 요소에서만 사용되리라 보증할 방법이 없다.



```
1 public class TestSafety {  
2     public void testSomething() {  
3         // ...  
4     }  
5 }
```



애니메이션



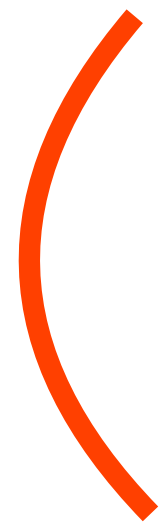
애너테이션

**애너테이션(Annotation)은 자바 코드에 메타데이터(추가 정보를) 붙이는 수단으로,
컴파일러나 런타임, 도구 등에 의해 읽혀서 특별한 처리를 하도록 돕는 기능**



애너테이션

메타애너테이션



```
@Target({ElementType.ANNOTATION_TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@API(status = STABLE, since = "5.0")  
@Testable  
public @interface Test {  
}
```


메타애너테이션

애너테이션 선언에 다른 애너테이션을 **메타 애너테이션**이라 한다.

메타애너테이션

애너테이션 선언에 `다`는 애너테이션을 메타 애너테이션이라 한다.

`Retention(RetentionPolicy.RUNTIME)` 메타애너테이션은 `@Test`가 런타임에도 유지되어야 한다는 표시이다

만약 이 메타애너테이션을 생략하면 테스트 도구는 `@Test`를 인식할 수 없다.

한편 `@Target(ElementType.METHOD)` 메타애너테이션은 `@Test`가 반드시 메서드 선언에서만 사용돼야 한다고 알려준다.


```

@Target({ElementType.ANNOTATION_TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@API(status = STABLE, since = "5.0")
@Testable
public @interface Test {
}

```

@Retention - 애너테이션이 실제로 적용되고 유지 되는 범위

RetentionPolicy.RUNTIME - 컴파일 이후에도 JVM에 의해 계속 참조가 가능
.class 파일에도 남고 런타임에도 유지 -> 리플렉션으로 읽을 수 있음

RetentionPolicy.CLASS - 컴파일러가 클래스를 참조할 때 까지 유효
.class 파일에는 남지만 JVM 런타임에는 제거 -> 리플렉션으로는 안 보임

RetentionPolicy.SOURCE - 컴파일 전까지 유효, 컴파일 이후에는 사라진다.
컴파일 시점에만 사용되고 .class 파일에는 남지 않음

1 RetentionPolicy.RUNTIME - 컴파일 이후에도 JVM에 의해 계속 참조가 가능

주로 코딩 편의(오타 검출, 자동완성), 컴파일 타임 코드 생성용 프로세서에 사용돼요.

@Deprecated

@Component

@Autowired

@Entity

@Test

@Valid

@Aspect

2 RetentionPolicy.CLASS (기본값) - 컴파일러가 클래스를 참조할 때 까지 유효

.class 파일엔 남아 있지만, JVM이 로딩할 때는 버립니다.

기본 @Retention을 생략하면 여기에 해당합니다.

3 RetentionPolicy.SOURCE - 컴파일 전까지 유효, 컴파일 이후에는 사라진다.

.class 파일에 남고, 실제 애플리케이션 실행 중 Class.getAnnotation(...) 등으로 읽어 사용할 수 있습니다.

@Override

@SuppressWarnings("unchecked")

@FunctionalInterface

@NonNull

**스프링의 DI·AOP, JPA 매핑, 테스트 런너, JSON 직렬화·검증 등
런타임에 반드시 애노테이션 정보를 참조해야 하는 경우에 씁니다.**

```

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Reflective(ExceptionHandlerReflectiveProcessor.class)
public @interface ExceptionHandler {

    Exceptions handled by the annotated method.
    This is an alias for exception .

    @AliasFor("exception")
    Class<? extends Throwable>[] value() default {};

    Exceptions handled by the annotated method. If empty, will default to any exceptions listed in the
    method argument list.

    Since: 6.2

    @AliasFor("value")
    Class<? extends Throwable>[] exception() default {};

    Media Types that can be produced by the annotated method.

    Since: 6.2

    String[] produces() default {};

}

```

예외 처리 로직을 실행 중에 찾아내려면 꼭 필요
Javadoc 생성 시 이 어노테이션을 포함시켜 문서화

AOT컴파일 지원을 위한 메타 애노테이션

애노테이션에 선언된 속성 중 이름이 `value`인 경우
애노테이션 요소를 하나만 설정할 땐 `value =` 를 생략하고
바로 값을 전달할 수 있다.

@AliasFor는 애노테이션 속성간에 별칭 관계를
정의해주는 메타애노테이션이다.

같은 의미의 속성 `value`, `exception` 처럼 이름만 다르게 두고 싶을 때 둘 중 어느
하나에 값을 주면 다른 하나에도 자동으로 반영되도록 해준다.

ITEM 39

명명 패턴보다 애너테이션을 사용하라


강철원

실습 : 커스텀 애너테이션 만들어보기


1. 의존성 추가하기

```
implementation 'org.springframework.boot:spring-boot-starter-aop'
```


2. config 파일 만들기



```
1  @Configuration
2  @EnableAspectJAutoProxy
3  public class AopConfig {
4  }
5
```



@Aspect로 선언된 빈(Aspect)을 찾아서,
그 포인트컷(Advice가 적용될 지점)에 맞춰 런타임에
프록시 객체를 만들어 줍니다.

스프링에게 AspectJ 기반의 자동 프록시 생성 기능을 활성화하라고 지시하는 애노테이션입니다.

런타임 프록시 생성

스프링 컨테이너가 시작될 때 @Aspect 빈을 찾아 해당 대상(target) 빈에 프록시를 씌워 줍니다.

이후 클라이언트가 빈의 메서드를 호출하면, 실제 로직 전에 Advice가 실행되고,

그 뒤에 원래 로직이 실행되는 구조가 됩니다.

3, 애너테이션 제작



```
1  @Retention(RetentionPolicy.RUNTIME)
2  @Target(ElementType.METHOD)
3  public @interface PrintMessage {
4      String value() default "메서드 호출됨!";
5
6      int times() default 1;
7  }
```

```
1  @Aspect
2  @Component
3  public class PrintMessageAspect {
4
5      @Around("@annotation(printMessage)")
6      public Object around(ProceedingJoinPoint pjp, PrintMessage printMessage) throws Throwable {
7
8          final String value = printMessage.value();
9          final int times = printMessage.times();
10
11          // 메서드 실행 전
12          for (int i = 0; i < times; i++) {
13              System.out.printf(">>> [%s] 출력%n", value);
14          }
15
16          Object result = pjp.proceed(); // 실제 메서드 호출
17
18          // 메서드 실행 후
19          System.out.println(">>> [PrintMessage] 메서드 실행 완료");
20          return result;
21      }
22  }
```

@Aspect로 선언된 빈(Aspect)을 찾아서, 그 포인트컷(Advice가 적용될 지점)에 맞춰 런타임에 프록시 객체를 만들어 줍니다.

애너테이션 설명

@Aspect

이 클래스가 AOP의 Aspect임을 표시합니다.
스프링이 런타임에 이 빈을 스캔해 Advice로 인식한다.

@Around

Pointcut 표현식으로, @PrintMessage 애노테이션이 붙은 모든 메서드를 가로챈다.

ProceedingJoinPoint

가로챈 대상 (Join Point), 즉 실제 호출될 객체와 메서드 정보를 담고 있다.

