이펙티브 자바

Item 39 ~ 41

2024/03/26

발표자: 박지원

Item 39 명명 패턴보다 에너테이션을 사용하라

명명 패턴: 전통적으로 도구, 프레임워크에서 이름을 정해서 사용하는 것

Getter, Setter...

Item 39 명명 패턴보다 에너테이션을 사용하라

명명 패턴의 단점

1. 오타가 나면 안된다.
2. 올바른 프로그램 요소에만 사용될 보장이 되지 않는다.
3. 프로그램 요소를 매개변수로 전달할 방법이 없다(추상적)

Test의 경우 테스트실행 유무을 파악하기 어렵다

Item 39 명명 패턴보다 에너테이션을 사용하라

Annotation: 주석, 프로그램에 추가정보를 메타데이터

@interface 키워드 사용

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}
```

Item 39 명명 패턴보다 에너테이션을 사용하라

@Retention: 에너테이션 범위 설정

RententionPolicy.RUNTIME: 컴파일 이후 런타임 시기에 JVM에 의해 참조 가능

RetentionPolicy.CLASS: .class까지 에너테이션 유지, 런타임에 사라진다.

RetentionPolicy.SOURCE: .java까지 에너테이션 유지

Item 39 명명 패턴보다 에너테이션을 사용하라

```java
@Retention(RetentionPolicy.SOURCE)
@interface SourceRetention
{
    String value() default "Source Retention";
}

@Retention(RetentionPolicy.CLASS)
@interface ClassRetention
{
    String value() default "Class Retention";
}

@Retention(RetentionPolicy.RUNTIME)
@interface RuntimeRetention
{
    String value() default "Runtime Retention";
}

@SourceRetention
class A {
}

@ClassRetention
class B {
}

@RuntimeRetention
class C {
};
```

```java
Annotation a[]
    = A.class.getAnnotations();
Annotation b[]
    = B.class.getAnnotations();
Annotation c[]
    = C.class.getAnnotations();

System.out.println(a.length);

System.out.println(b.length);

System.out.println(c.length);
```

0 0 1

런타임에 C를 제외하고 모두 삭제

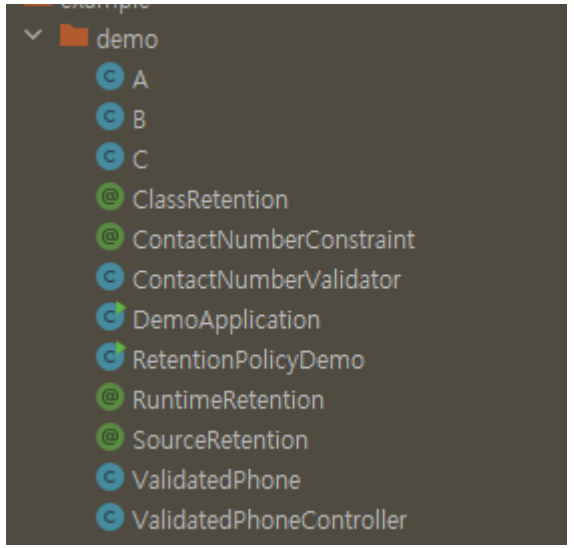
Item 39 명명 패턴보다 에너테이션을 사용하라

.Class 파일 확인하면..

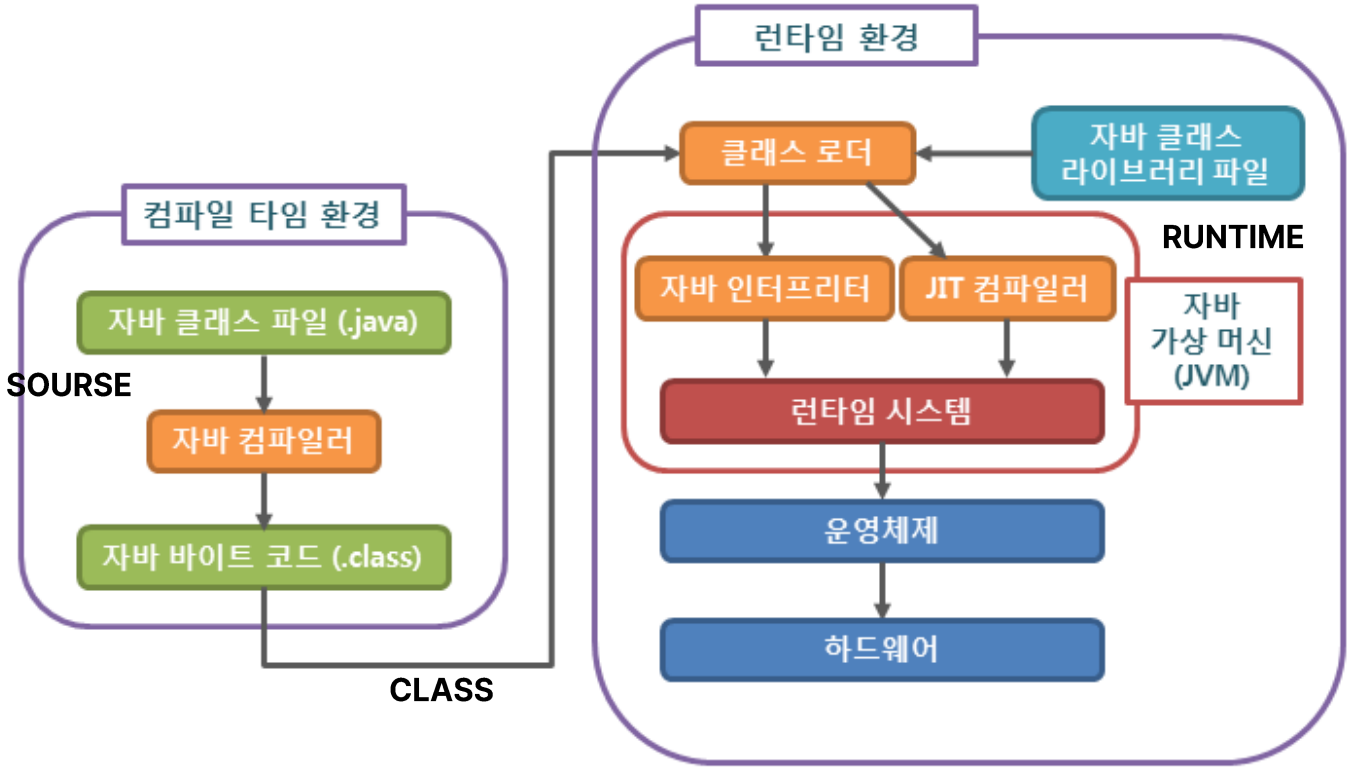
```
class A {
    A() {
    }
}
```

```
@ClassRetention
class B {
    B() {
    }
}
```

```
@RuntimeRetention
class C {
    C() {
    }
}
```



Item 39 명명 패턴보다 에너테이션을 사용하라

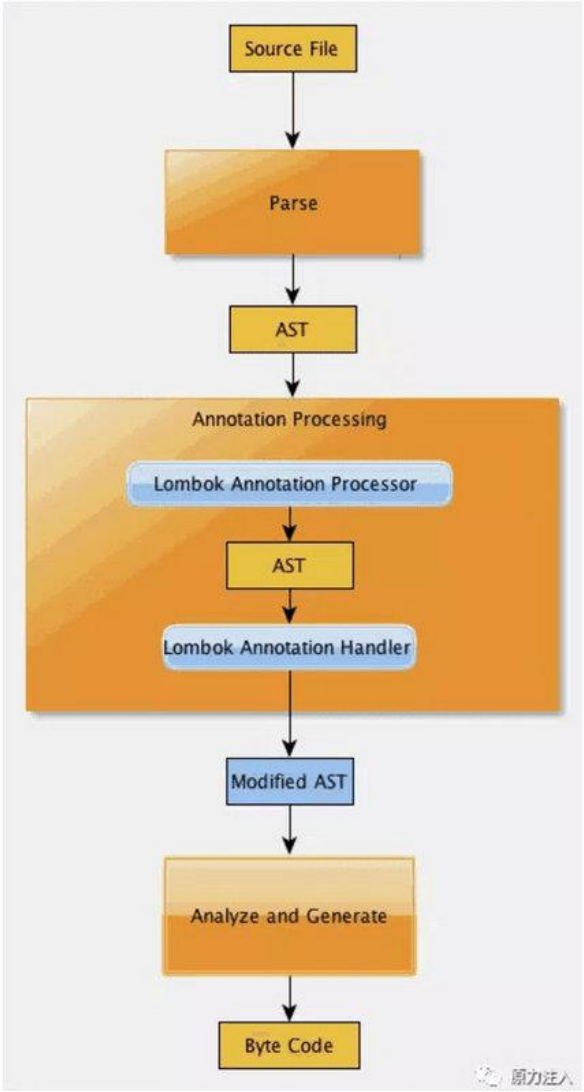


Item 39 명명 패턴보다 에너테이션을 사용하라

Abstract Syntax Tree(AST)

Processing 과정에서 코드를 주입한다.(AST 트리 수정)

Compiler가 AST를 통한 Byte code 생성



Item 39 명명 패턴보다 에너테이션을 사용하라

RententionPolicy.RUNTIME: Notnull, Id, GeneratedValue

RetentionPolicy.CLASS: Nonnull

RetentionPolicy.SOURCE: Getter, Setter, NoArgsconstructor

.class의 경우 Gradle로 빌드된 jar파일의 경우 소스파일(.java)가 포함되지 않고 (.class) 파일만 존재 컴파일 단계에서 해결할 수 없는 경우 CLASS로 사용함

Item 39 명명 패턴보다 에너테이션을 사용하라

@Target: 에너테이션 적용 위치를 설정한다.

ElementType.METHOD

ElementType.FIELD

ElementType.CONSTRUCTOR

ElementType.PARAMETER

Item 39 명명 패턴보다 에너테이션을 사용하라

Custom validation annotation

```
public class SampleTestValidator implements ConstraintValidator<SampleTest, String>

  @Override
  public boolean isValid(String value, ConstraintValidatorContext context) { // 2
    if (value == null) {
      return false;
    }

    return value.matches("^[a-zA-Z]*$");
  }
}
```

```
@Target(ElementType.FIELD) // 1
@Retention(RetentionPolicy.RUNTIME) // 2
@Constraint(validatedBy = SampleTestValidator.class) // 3
public @interface SampleTest {
  String message() default "휴대폰 번호"; // 4
  Class[] groups() default {};
  Class[] payload() default {};
}
```

Item 39 명명 패턴보다 에너테이션을 사용하라

reflection을 이용한 테스트 (Runtime)

```
public class Sample {
    @Test
    public static void m1() { }        // 성공해야 한다.
    public static void m2() { }
    @Test
    public static void m3() {    // 실패해야 한다.
        throw new RuntimeException("실패");
    }
    public static void m4() { }  // 테스트가 아니다.
    @Test
    public void m5() { }   // 잘못 사용한 예: 정적 메서드가 아니다
    public static void m6() { }
    @Test
    public static void m7() {    // 실패해야 한다.
        throw new RuntimeException("실패");
    }
    public static void m8() { }
}
```

```
Class<?> testClass = Class.forName("item12.Sample1");
for (Method m : testClass.getDeclaredMethods()) {
    if (m.isAnnotationPresent(Test1.class)) {
        tests++;
        try {
            m.invoke(null);
            passed++;
        } catch (InvocationTargetException wrappedExc) {
            Throwable exc = wrappedExc.getCause();
            System.out.println(m + " 실패: " + exc);
        } catch (Exception exc) {
            System.out.println("잘못 사용한 @Test: " + m);
        }
    }
}
```

발표자: 박지원

Item 39 명명 패턴보다 에너테이션을 사용하라

특정 예외에 대한 처리, 여러가지 예외에 대한 처리

한정적 타입 토큰 사용

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Throwable> value();  // 매개변수
}
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest2 {
    Class<? extends Exception>[] value();   // Class 객체의 배열
}
```

Item 39 명명 패턴보다 에너테이션을 사용하라

@Repeatable: 컨테이너 에너테이션을 정의해야함

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Repeatable(ExceptionTestContainer.class)  //컨테이너 애너테이션 cla
public @interface ExceptionTest3 {
    Class<? extends Throwable> value();
}
```

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTestContainer {
    ExceptionTest3[] value();     // value 메서드 정의
}
```

Item 39 명명 패턴보다 에너테이션을 사용하라

명명패턴보다 에너테이션을 사용하자

에너테이션은 의도의 숨길 수 있으니 주의하자

발표자: 박지원

Item 40 @Override 에너테이션을 일관되게 사용하라

항상 습관화 합시다....

```
public boolean equals(Bigram b) {
    return b.first == first && b.second == second;
}
```

```
@Override public boolean equals(Object o) {
    if (!(o instanceof Bigram))
        return false;
    Bigram b = (Bigram) o;
    return b.first == first && b.second == second;
}
```

Item 41 정의하려는 것이 타입이라면 마커 인터페이스를 사용하자

마커 인터페이스: 메서드를 달지 않고, 속성 표시가 목적인 인터페이스

```
public interface Serializable {
}
```

@Serializable

Item 41 정의하려는 것이 타입이라면 마커 인터페이스를 사용하자

마커 인터페이스는 구현 클래스의 인스턴스를 구분하는 용도로 사용할 수 있다

마커 에너테이션은 런타임이 되서야 발견할 수 있다.

컴파일 에러는 IDLE에서 해준건가??

Item 41 정의하려는 것이 타입이라면 마커 인터페이스를 사용하자

마커 인터페이스는 적용대상을 더 정밀하게 지정할 수 있다.

@Target으로 한계가 있다.

마커에너테이션이 거대한 에너테이션 시스템이라는 점에서 일관성을 지키기 유리하다는 의견

마커 인터페이스를 사용하자