13. 애노테이션

#0.강의/1.자바로드맵/6.자바-고급2편

애노테이션이 필요한 이유

남은 문제점

- 리플렉션 서블릿은 요청 URL과 메서드 이름이 같다면 해당 메서드를 동적으로 호출할 수 있다. 하지만 요청 이름과 메서드 이름을 다르게 하고 싶다면 어떻게 해야할까?
- 예를 들어서 `/site1`이라고 와도 `page1()`과 같은 메서드를 호출하고 싶다면 어떻게 해야할까?
  - 예를 들어서 메서드 이름은 더 자세히 적고 싶을 수 있다.
- 앞서 `/`와 같이 자바 메서드 이름으로 처리하기 어려운 URL은 어떻게 해결할 수 있을까?
- URL은 주로 `-`(dash)를 구분자로 사용한다. `/add-member`와 같은 URL은 어떻게 해결할 수 있을까?

방금 설명한 문제들은 메서드 이름만으로는 해결이 어렵다. 추가 정보를 코드 어딘가에 적어두고 읽을 수 있어야 한다. 다음 코드를 보자.

```
public class Controller {

    // "/site1"
    public void page1(HttpRequest request, HttpResponse response) {
    }

    // "/"
    public void home(HttpRequest request, HttpResponse response) {
        response.writeBody("<h1>site2</h1>");
    }

    // "/add-member"
    public void addMember(HttpRequest request, HttpResponse response) {
```

```
    }

}
```

만약 리플렉션 같은 기술로 메서드 이름 뿐만 아니라 주석까지 읽어서 처리할 수 있다면 좋지 않을까?
그러면 해당 메서드에 있는 주석을 읽어서 URL 경로와 비교하면 된다. 만약 같다면 해당 주석이 달린 메서드를 호출해 버리는 것이다!
그런데 주석은 코드가 아니다. 따라서 컴파일 시점에 모두 제거된다.
만약 프로그램 실행 중에 읽어서 사용할 수 있는 주석이 있다면 어떨까? 이것이 바로 애노테이션이다.

애노테이션 예제

애노테이션에 대해서 본격적으로 알아보기 전에, 간단한 예제를 통해 실제 우리가 고민한 문제를 애노테이션으로 어떻게 해결하는지 알아보자. 애노테이션에 대한 자세한 내용은 예제 이후에 설명하겠다.

```
package annotation.mapping;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
public @interface SimpleMapping {
    String value();
}
```

- 애노테이션은 `@interface` 키워드를 사용해서 만든다.
- `@SimpleMapping`이라는 애노테이션을 하나 만든다. 내부에는 `String value`라는 속성을 하나 가진다.
- `@Retention`은 뒤에서 설명한다. 지금은 필수로 사용해야 하는 값 정도로 생각하자.

```
package annotation.mapping;

public class TestController {

    @SimpleMapping(value = "/")
    public void home() {
        System.out.println("TestController.home");
    }
```

```
    @SimpleMapping(value = "/site1")
    public void page1() {
        System.out.println("TestController.page1");
    }

}
```

- 애노테이션을 사용할 때는 `@` 기호로 시작한다.
- `home()`에는 `@SimpleMapping(value = "/")` 애노테이션을 붙였다.
- `page1()`에는 `@SimpleMapping(value = "/site1")` 애노테이션을 붙였다.

참고로 애노테이션은 프로그램 코드가 아니다. 예제에서 애노테이션이 붙어있는 `home()`, `page1()` 같은 코드를 호출해도 프로그램에는 아무런 영향을 주지 않는다. 마치 주석과 비슷하다고 이해하면 된다. 다만 일반적인 주석이 아니라, 리플렉션 같은 기술로 실행 시점에 읽어서 활용할 수 있는 특별한 주석이다.

```
package annotation.mapping;

import java.lang.reflect.Method;

public class TestControllerMain {

    public static void main(String[] args) {
        TestController testController = new TestController();

        Class<? extends TestController> aClass = testController.getClass();
        for (Method method : aClass.getDeclaredMethods()) {
            SimpleMapping simpleMapping =
method.getAnnotation(SimpleMapping.class);
            if (simpleMapping != null) {
                System.out.println("["+simpleMapping.value() + "] -> " +
method);
            }
        }
    }
}
```

- `TestController` 클래스의 선언된 메서드를 찾는다.
- 리플렉션이 제공하는 `getAnnotation()` 메서드를 사용하면 붙어있는 애노테이션을 찾을 수 있다.
    - `Class`, `Method`, `Field`, `Constructor` 클래스는 자신에게 붙은 애노테이션을 찾을 수 있는

`getAnnotation()` 메서드를 제공한다.
    - 여기서는 `Method.getAnnotation(SimpleMapping.class)`을 사용했으므로 해당 메서드에 붙은 `@SimpleMapping` 애노테이션을 찾을 수 있다.
- `simpleMapping.value()`를 사용해서 찾은 애노테이션에 지정된 값을 조회할 수 있다.

실행 결과

```
[/] -> public void annotation.mapping.TestController.home()
[/site1] -> public void annotation.mapping.TestController.page1()
```

- `home()` 메서드에 붙은 `@SimpleMapping` 애노테이션의 value 값은 `/`이다.
- `page1()` 메서드에 붙은 `@SimpleMapping` 애노테이션의 value 값은 `/site1`이다.

이 예제를 통해 리플렉션 서블릿에서 해결하지 못했던 문제들을 어떻게 해결해야 하는지 바로 이해가 될 것이다.
애노테이션 기반의 서블릿은 뒤에 만들어보고, 우선은 애노테이션 자체에 대해서 자세히 알아보자.

참고: 애노테이션 단어
자바 애노테이션(Annotation)의 영어 단어 "Annotation"은 일반적으로 "주석" 또는 "메모"를 의미한다.
애노테이션은 코드에 추가적인 정보를 주석처럼 제공한다. 하지만 일반 주석과 달리, 애노테이션은 컴파일러나 런타임에서 해석될 수 있는 메타데이터를 제공한다. 즉, 애노테이션은 코드에 메모를 달아놓는 것처럼 특정 정보나 지시를 추가하는 도구로, 코드에 대한 메타데이터를 표현하는 방법이다.
따라서 "애노테이션"이라는 이름은 코드에 대한 추가적인 정보를 주석처럼 달아놓는다는 뜻이다.

애노테이션 정의

```
package annotation.basic;

import util.MyLogger;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
public @interface AnnoElement {
    String value();
```

```
    int count() default 0;
    String[] tags() default {};

    //MyLogger data(); // 다른 타입은 적용X
    Class<? extends MyLogger> annoData() default MyLogger.class; // 클래스 정보는
가능
}
```

- 애노테이션은 `@interface` 키워드로 정의한다.
- 애노테이션은 속성을 가질 수 있는데, 인터페이스와 비슷하게 정의한다.

애노테이션 정의 규칙

데이터 타입

- 기본 타입 (int, float, boolean 등)
- String
- `Class` (메타데이터) 또는 인터페이스
- enum
- 다른 애노테이션 타입
- 위의 타입들의 배열
- 앞서 설명한 타입 외에는 정의할 수 없다. 쉽게 이야기해서 일반적인 클래스를 사용할 수 없다.
  - 예) `Member`, `User`, `MyLogger`

default 값

- 요소에 default 값을 지정할 수 있다.
- 예: `String value() default "기본 값을 적용합니다.";`

요소 이름

- 메서드 형태로 정의된다.
- 괄호()를 포함하되 매개변수는 없어야 한다.

반환 값

- `void`를 반환 타입으로 사용할 수 없다.

예외

- 예외를 선언할 수 없다.

특별한 요소 이름

`value`라는 이름의 요소를 하나만 가질 경우, 애노테이션 사용 시 요소 이름을 생략할 수 있다.

애노테이션 사용

```
package annotation.basic;

@AnnoElement(value = "data", count = 10, tags = {"t1", "t2"})
public class ElementData1 {

}
```

```
package annotation.basic;

import java.util.Arrays;

public class ElementData1Main {

    public static void main(String[] args) {
        Class<ElementData1> annoClass = ElementData1.class;
        AnnoElement annotation = annoClass.getAnnotation(AnnoElement.class);

        String value = annotation.value();
        System.out.println("value = " + value);

        int count = annotation.count();
        System.out.println("count = " + count);

        String[] tags = annotation.tags();
        System.out.println("tags = " + Arrays.toString(tags));
    }
}
```

실행 결과

```
value = data
count = 10
```

```
tags = [t1, t2]
```

```
package annotation.basic;

@AnnoElement(value = "data", tags = "t1")
public class ElementData2 {

}
```

- default 항목은 생략 가능
- 배열의 항목이 하나인 경우 `{}` 생략 가능

```
package annotation.basic;

// 입력 요소가 하나인 경우 value 키 생략 가능
@AnnoElement("data")
public class ElementData3 {

}
```

- 입력 요소가 하나인 경우 `value` 키워드 생략 가능
- `value="data"` 와 같다.

메타 애노테이션

애노테이션을 정의하는데 사용하는 특별한 애노테이션을 메타 애노테이션이라 한다.

다음과 같은 메타 애노테이션이 있다. 하나씩 알아보자.

- `@Retention`
  - `RetentionPolicy.SOURCE`
  - `RetentionPolicy.CLASS`
  - `RetentionPolicy.RUNTIME`
- `@Target`
- `@Documented`

- `@Inherited`

@Retention

애노테이션의 생존 기간을 지정한다.

```
package java.lang.annotation;

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Retention {
    RetentionPolicy value();
}
```

```
public enum RetentionPolicy {
    SOURCE,
    CLASS,
    RUNTIME
}
```

- `RetentionPolicy.SOURCE` : 소스 코드에만 남아있다. 컴파일 시점에 제거된다.
- `RetentionPolicy.CLASS` : 컴파일 후 class 파일까지는 남아있지만 자바 실행 시점에 제거된다. (기본 값)
- `RetentionPolicy.RUNTIME` : 자바 실행 중에도 남아있다. 대부분 이 설정을 사용한다.

@Target

애노테이션을 적용할 수 있는 위치를 지정한다.

```
package java.lang.annotation;

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target {
    ElementType[] value();
}
```

```
public enum ElementType {
    TYPE,
    FIELD,
    METHOD,
    PARAMETER,
    CONSTRUCTOR,
    LOCAL_VARIABLE,
    ANNOTATION_TYPE,
    PACKAGE,
    TYPE_PARAMETER,
    TYPE_USE,
    MODULE,
    RECORD_COMPONENT;
}
```

- 이름만으로 충분히 이해가 될 것이다. 주로 `TYPE`, `FIELD`, `METHOD`를 사용한다.

@Documented

자바 API 문서를 만들 때 해당 애노테이션이 함께 포함되는지 지정한다. 보통 함께 사용한다.

@Inherited

자식 클래스가 애노테이션을 상속 받을 수 있다.

이 애노테이션은 뒤에서 더 자세히 알아보겠다.

적용 예시

```
package annotation.basic;

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE})
@Documented
public @interface AnnoMeta {
}
```

- `@Retention`: `RUNTIME` 자바 실행 중에도 애노테이션 정보가 남아있다. 따라서 런타임에 리플렉션을 통해서 읽을 수 있다. 만약 다른 설정을 적용한다면 자바 실행 시점에 애노테이션이 사라지므로 리플렉션을 통해서 읽을 수 없다.
- `@Target`: `ElementType.METHOD`, `ElementType.TYPE` 메서드와 타입(클래스, 인터페이스, enum 등)에 `@AnnoMeta` 애노테이션을 적용할 수 있다. 다른 곳에 적용하면 컴파일 오류가 발생한다.
- `@Documented`: 자바 API 문서를 만들 때 해당 애노테이션이 포함된다.

```
package annotation.basic;

@AnnoMeta // 타입에 적용
public class MetaData {

    //@AnnoMeta // 필드에 적용 - 컴파일 오류
    private String id;

    @AnnoMeta // 메서드에 적용
    public void call() {

    }

    public static void main(String[] args) throws NoSuchMethodException {
        AnnoMeta typeAnno = MetaData.class.getAnnotation(AnnoMeta.class);
        System.out.println("typeAnno = " + typeAnno);

        AnnoMeta methodAnno =
MetaData.class.getMethod("call").getAnnotation(AnnoMeta.class);
        System.out.println("methodAnno = " + methodAnno);
    }

}
```

- 타입과 메서드에 해당 애노테이션을 적용할 수 있다.
- 필드에 적용하면 컴파일 오류가 발생한다.
    - 자바 언어는 컴파일 시점에 `@Target` 메타 애노테이션을 읽어서 지정한 위치가 맞는지 체크한다.

실행 결과

```
typeAnno = @annotation.basic.AnnoMeta()
methodAnno = @annotation.basic.AnnoMeta()
```

`@Retention(RetentionPolicy.RUNTIME)`을 다른 타입으로 변경해보면 애노테이션을 찾을 수 없는 것을 확인할 수 있다.

실행 결과 - @Retention(RetentionPolicy.CLASS) 변경

```
typeAnno = null
methodAnno = null
```

애노테이션과 상속

모든 애노테이션은 `java.lang.annotation.Annotation` 인터페이스를 묵시적으로 상속 받는다.

```
package java.lang.annotation;

public interface Annotation {
    boolean equals(Object obj);
    int hashCode();
    String toString();
    Class<? extends Annotation> annotationType();
}
```

`java.lang.annotation.Annotation` 인터페이스는 개발자가 직접 구현하거나 확장할 수 있는 것이 아니라, 자바 언어 자체에서 애노테이션을 위한 기반으로 사용된다. 이 인터페이스는 다음과 같은 메서드를 제공한다.

- `boolean equals(Object obj)` : 두 애노테이션의 동일성을 비교한다.
- `int hashCode()` : 애노테이션의 해시코드를 반환한다.
- `String toString()` : 애노테이션의 문자열 표현을 반환한다.
- `Class<? extends Annotation> annotationType()` : 애노테이션의 타입을 반환한다.

모든 애노테이션은 기본적으로 `Annotation` 인터페이스를 확장하며, 이로 인해 자바에서 애노테이션은 특별한 형태의 인터페이스로 간주된다. 하지만 자바에서 애노테이션을 정의할 때, 개발자가 명시적으로 `Annotation` 인터페이스를 상속하거나 구현할 필요는 없다. 애노테이션을 `@interface` 키워드를 통해 정의하면, 자바 컴파일러가 자동으로

`Annotation` 인터페이스를 확장하도록 처리해준다.

애노테이션 정의

```
public @interface MyCustomAnnotation {}
```

자바가 자동으로 처리

```
public interface MyCustomAnnotation extends java.lang.annotation.Annotation {}
```

애노테이션과 상속

- 애노테이션은 다른 애노테이션이나 인터페이스를 직접 상속할 수 없다.
- 오직 `java.lang.annotation.Annotation` 인터페이스만 상속한다.
- 따라서 애노테이션 사이에는 상속이라는 개념이 존재하지 않는다.

@Inherited

애노테이션을 정의할 때 `@Inherited` 메타 애노테이션을 붙이면, 애노테이션을 적용한 클래스의 자식도 해당 애노테이션을 부여 받을 수 있다.
단 주의할 점으로 이 기능은 클래스 상속에서만 작동하고, 인터페이스의 구현체에는 적용되지 않는다.

예제로 자세히 알아보자.

```
package annotation.basic.inherited;

import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Inherited // 클래스 상속시 자식도 애노테이션 적용
@Retention(RetentionPolicy.RUNTIME)
public @interface InheritedAnnotation {
}
```

- `InheritedAnnotation`은 `@Inherited`를 가진다.

```

```

```java
package annotation.basic.inherited;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
public @interface NoInheritedAnnotation {
}
```

- `NoInheritedAnnotation`은 `@Inherited`를 가지지 않는다.

```java
package annotation.basic.inherited;

@InheritedAnnotation
@NoInheritedAnnotation
public class Parent {
}
```

- `Parent`에는 `@InheritedAnnotation`, `@NoInheritedAnnotation` 모두 붙어있다.

```java
package annotation.basic.inherited;

public class Child extends Parent {
}
```

- `Child`는 `@InheritedAnnotation` 애노테이션을 상속 받는다.
  - `@Inherited` 메타 애노테이션이 붙어있다.
- `@NoInheritedAnnotation`는 상속 받지 못한다.
  - `@Inherited` 메타 애노테이션이 붙어있지 않다.

```java
package annotation.basic.inherited;

@InheritedAnnotation
@NoInheritedAnnotation
public interface TestInterface {
}
```

- `TestInterface`에는 `@InheritedAnnotation`, `@NoInheritedAnnotation` 모두 붙어있다.

```
package annotation.basic.inherited;

public class TestInterfaceImpl implements TestInterface {
}
```

- 인터페이스의 구현에서는 애노테이션을 상속 받을 수 없다.
- 참고로 인터페이스 부모와 인터페이스 자식의 관계에서도 애노테이션을 상속 받을 수 없다.

```
package annotation.basic.inherited;

import java.lang.annotation.Annotation;

public class InheritedMain {

    public static void main(String[] args) {
        print(Parent.class);
        print(Child.class);
        print(TestInterface.class);
        print(TestInterfaceImpl.class);
    }

    private static void print(Class<?> clazz) {
        System.out.println("class: " + clazz);
        for (Annotation annotation : clazz.getAnnotations()) {
            System.out.println(" - " +
annotation.annotationType().getSimpleName());
        }
        System.out.println();
    }
}
```

실행 결과

```
class: class annotation.basic.inherited.Parent
 - InheritedAnnotation
 - NoInheritedAnnotation

class: class annotation.basic.inherited.Child
```

```
 - InheritedAnnotation

class: interface annotation.basic.inherited.TestInterface
 - NoInheritedAnnotation
 - InheritedAnnotation

class: class annotation.basic.inherited.TestInterfaceImpl
```

- `Child`: `InheritedAnnotation` 상속
- `TestInterfaceImpl`: 애노테이션을 상속 받을 수 없음

@Inherited가 클래스 상속에만 적용되는 이유

1. 클래스 상속과 인터페이스 구현의 차이

- 클래스 상속은 자식 클래스가 부모 클래스의 속성과 메서드를 상속받는 개념이다. 즉, 자식 클래스는 부모 클래스의 특성을 이어받으므로, 부모 클래스에 정의된 애노테이션을 자식 클래스가 자동으로 상속받을 수 있는 논리적 기반이 있다.
- 인터페이스는 메서드의 시그니처만을 정의할 뿐, 상태나 행위를 가지지 않기 때문에, 인터페이스의 구현체가 애노테이션을 상속한다는 개념이 잘 맞지 않는다.

2. 인터페이스와 다중 구현, 다이아몬드 문제

인터페이스는 다중 구현이 가능하다. 만약 인터페이스의 애노테이션을 구현 클래스에서 상속하게 되면 여러 인터페이스의 애노테이션 간의 충돌이나 모호한 상황이 발생할 수 있다.

이제 애노테이션의 기본적인 내용들은 모두 알아보았다. 이런 애노테이션을 어떻게 잘 활용할 수 있는지 예제를 통해 알아보자.

애노테이션 활용 - 검증기

각종 클래스의 정보들을 검증(validation)하는 기능을 만들어보자.

```
package annotation.validator;

public class Team {
```

```
    private String name;
    private int memberCount;

    public Team(String name, int memberCount) {
        this.name = name;
        this.memberCount = memberCount;
    }

    public String getName() {
        return name;
    }

    public int getMemberCount() {
        return memberCount;
    }
}
```

```
package annotation.validator;

public class User {

    private String name;
    private int age;

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

}
```

```java
package annotation.validator;

import static util.MyLogger.log;

public class ValidatorV1Main {

    public static void main(String[] args) {
        User user = new User("user1", 0);
        Team team = new Team("", 0);

        try {
            log("== user 검증 ==");
            validateUser(user);
        } catch (Exception e) {
            log(e);
        }

        try {
            log("== team 검증 ==");
            validateTeam(team);
        } catch (Exception e) {
            log(e);
        }
    }

    private static void validateUser(User user) {
        if (user.getName() == null || user.getName().isEmpty()) {
            throw new RuntimeException("이름이 비어있습니다.");
        }
        if (user.getAge() < 1 || user.getAge() > 100) {
            throw new RuntimeException("나이는 1과 100 사이여야 합니다.");
        }

    }

    private static void validateTeam(Team team) {
        if (team.getName() == null || team.getName().isEmpty()) {
            throw new RuntimeException("이름이 비어있습니다.");
        }
        if (team.getMemberCount() < 1 || team.getMemberCount() > 999) {
            throw new RuntimeException("회원 수는 1과 999 사이여야 합니다.");
        }
    }
```

```
}
```

- `User` 객체의 이름과 나이를 검증한다.
- `Team` 객체의 이름과 나이를 검증한다.

실행 결과

```
17:40:20.412 [     main] == user 검증 ==
17:40:20.414 [     main] java.lang.RuntimeException: 나이는 1과 100 사이여야 합니다.
17:40:20.414 [     main] == team 검증 ==
17:40:20.414 [     main] java.lang.RuntimeException: 이름이 비어있습니다.
```

여기서는 값이 비었는지 검증하는 부분과 숫자의 범위를 검증하는 2가지 부분이 있다.
코드를 잘 보면 뭔가 비슷한 것 같으면서도 `User`, `Team`이 서로 완전히 다른 클래스이기 때문에 재사용이 어렵다. 그리고 각각의 필드 이름도 서로 다르고, 오류 메시지도 다르다. 그리고 검증해야 할 값의 범위도 다르다.

이후에 다른 객체들도 검증해야 한다면 비슷한 검증 기능을 계속 추가해야 한다.
이런 문제를 애노테이션을 사용해서 해결해보자.

애노테이션 기반 검증기

@NotEmpty - 빈 값을 검증하는데 사용

```
package annotation.validator;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface NotEmpty {
    String message() default "값이 비어있습니다.";
}
```

- `message`: 검증에 실패한 경우 출력할 오류 메시지

@Range - 숫자의 범위를 검증하는데 사용

```java
package annotation.validator;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Range {
    int min();
    int max();
    String message() default "범위를 넘었습니다.";
}
```

- `min` : 최소 값
- `max` : 최대 값
- `message` : 검증에 실패한 경우 출력할 오류 메시지

User에 검증용 애노테이션을 추가하자.

```java
package annotation.validator;

public class User {

    @NotEmpty(message = "이름이 비어있습니다.")
    private String name;

    @Range(min = 1, max = 100, message = "나이는 1과 100 사이여야 합니다.")
    private int age;

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
```

```
        return age;
    }

}
```

Team에 검증용 애노테이션을 추가하자.

```
package annotation.validator;

public class Team {

    @NotEmpty(message = "이름이 비어있습니다.")
    private String name;

    @Range(min = 1, max = 999, message = "회원 수는 1과 999 사이여야 합니다.")
    private int memberCount;

    public Team(String name, int memberCount) {
        this.name = name;
        this.memberCount = memberCount;
    }

    public String getName() {
        return name;
    }

    public int getMemberCount() {
        return memberCount;
    }
}
```

이제 검증기를 만들어보자.

```
package annotation.validator;

import java.lang.reflect.Field;

public class Validator {

    public static void validate(Object obj) throws Exception {
```

```
        Field[] fields = obj.getClass().getDeclaredFields();

        for (Field field : fields) {
            field.setAccessible(true);

            if (field.isAnnotationPresent(NotEmpty.class)) {
                String value = (String) field.get(obj);
                NotEmpty annotation = field.getAnnotation(NotEmpty.class);
                if (value == null || value.isEmpty()) {
                    throw new RuntimeException(annotation.message());
                }
            }

            if (field.isAnnotationPresent(Range.class)) {
                long value = field.getLong(obj);
                Range annotation = field.getAnnotation(Range.class);
                if (value < annotation.min() || value > annotation.max()) {
                    throw new RuntimeException(annotation.message());
                }
            }
        }
    }
}
```

- 전달된 객체에 선언된 필드를 모두 찾아서 `@NotEmpty`, `@Range` 애노테이션이 붙어있는지 확인한다.
  - `isAnnotationPresent()`
- 애노테이션이 있는 경우 각 애노테이션의 속성을 기반으로 검증 로직을 수행한다. 만약 검증에 실패하면 애노테이션에 적용한 메시지를 예외에 담아서 던진다.

```
package annotation.validator;

import static util.MyLogger.log;

public class ValidatorV2Main {

    public static void main(String[] args) {
        User user = new User("user1", 0);
        Team team = new Team("", 0);

        try {
            log("== user 검증 ==");
```

```
            Validator.validate(user);
        } catch (Exception e) {
            log(e);
        }

        try {
            log("== team 검증 ==");
            Validator.validate(team);
        } catch (Exception e) {
            log(e);
        }
    }
}
```

실행 결과

```
17:50:19.220 [     main] == user 검증 ==
17:50:19.229 [     main] java.lang.RuntimeException: 나이는 1과 100 사이여야 합니다.
17:50:19.230 [     main] == team 검증 ==
17:50:19.230 [     main] java.lang.RuntimeException: 이름이 비어있습니다.
```

검증용 애노테이션과 검증기를 사용한 덕분에, 어떤 객체든지 애노테이션으로 간단하게 검증할 수 있게 되었다.

- 예를 들어, `@NotEmpty` 애노테이션을 사용하면 필드가 비었는지 여부를 편리하게 검증할 수 있고, `@Range(min=1, max=100)`와 같은 애노테이션을 통해 숫자의 범위를 쉽게 제한할 수 있다. 이러한 애노테이션 기반 검증 방식은 중복되는 코드 작성 없이도 유연한 검증 로직을 적용할 수 있어 유지보수성을 높여준다.
- `User` 클래스와 `Team` 클래스에 각각의 필드 이름이나 메시지들이 다르더라도, 애노테이션의 속성 값을 통해 필드 이름을 지정하고, 오류 메시지도 일관되게 정의할 수 있다. 예를 들어, `@NotEmpty(message = "이름은 비어 있을 수 없습니다")` 처럼 명시적인 메시지를 작성할 수 있으며, 이를 통해 다양한 클래스에서 공통된 검증 로직을 재사용할 수 있게 되었다.
- 또한, 새로 추가되는 클래스나 필드에 대해서도 복잡한 로직을 별도로 구현할 필요 없이 적절한 애노테이션을 추가하는 것만으로 검증 로직을 쉽게 확장할 수 있다. 이처럼 애노테이션 기반 검증을 도입하면 코드의 가독성과 확장성이 크게 향상되며, 일관된 규칙을 유지할 수 있어 전체적인 품질 관리에도 도움이 된다.
- 이제 클래스들이 서로 다르더라도, 일관되고 재사용 가능한 검증 방식을 사용할 수 있게 되었다.

참고

자바 진영에서는 애노테이션 기반 검증 기능을 Jakarta(Java) Bean Validaton이라는 이름으로 표준화 했다.
다양한 검증 애노테이션과 기능이 있고, 스프링 프레임워크, JPA 같은 기술들과도 함께 사용된다.

Jakarta Bean Validation에 대한 더 자세한 내용은 스프링 MVC 강의에서 다룬다.

- https://beanvalidation.org/
- https://hibernate.org/validator/

자바 기본 애노테이션

`@Override`, `@Deprecated`, `@SuppressWarnings`와 같이 자바 언어가 기본으로 제공하는 애노테이션도 있다. 참고로 앞서 설명한 `@Retention`, `@Target`도 자바 언어가 기본으로 제공하는 애노테이션이지만, 이것은 애노테이션 자체를 정의하기 위한 메타 애노테이션이고, 지금 설명한 내용은 코드에 직접 사용하는 애노테이션이다.

@Override

```
package java.lang;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

- 메서드 재정의가 정확하게 잘 되었는지 컴파일러가 체크하는데 사용한다.

```
package annotation.java;

public class OverrideMain {

    static class A {
        public void call() {
            System.out.println("A.call");
        }
    }

    static class B extends A {

        //@Override // 주석 풀면 컴파일 오류 발생
```

```
        public void callllll() {
            System.out.println("B.call");
        }
    }

    public static void main(String[] args) {
        A a = new B();
        a.call();
    }
}
```

- `B` 클래스는 `A` 클래스를 상속 받았다.
- `A.call()` 메서드를 `B` 클래스가 재정의하려고 시도한다. 이때 실수로 오타가 발생해서 재정의가 아니라 자식 클래스에 `callllll()`이라는 새로운 메서드를 정의해버렸다.
- 개발자의 의도는 `A.call()` 메서드의 재정의였지만, 자바 언어는 이것을 알 길이 없다. 자바 문법상 그냥 B에 `callllll()`이라는 새로운 메서드가 하나 만들어졌을 뿐이다.

실행 결과

```
A.call
```

이럴 때 `@Override` 애노테이션을 사용한다. 이 애노테이션을 붙이면 자바 컴파일러가 메서드 재정의 여부를 체크해 준다. 만약 문제가 있다면 컴파일을 통과하지 않는다!
개발자의 실수를 자바 컴파일러가 잡아주는 좋은 애노테이션이기 때문에, 사용을 강하게 권장한다.

`@Override`의 `@Retention(RetentionPolicy.SOURCE)` 부분을 보자.

- `RetentionPolicy.SOURCE`로 설정하면 컴파일 이후에 `@Override` 애노테이션은 제거된다.
- `@Override`는 컴파일 시점에만 사용하는 애노테이션이다. 런타임에는 필요하지 않으므로 이렇게 설정되어 있다.

@Deprecated

`@Deprecated`는 더 이상 사용되지 않는다는 뜻이다. 이 애노테이션이 적용된 기능은 사용을 권장하지 않는다.
예를 들면 다음과 같은 이유이다.

- 해당 요소를 사용하면 오류가 발생할 가능성이 있다.
- 호환되지 않게 변경되거나 향후 버전에서 제거될 수 있다.
- 더 나은 최신 대체 요소로 대체되었다.

- 더 이상 사용되지 않는 기능이다.

```
package java.lang;

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, MODULE,
PARAMETER, TYPE})
public @interface Deprecated {}
```

```
package annotation.java;

public class DeprecatedClass {

    public void call1() {
        System.out.println("DeprecatedClass.call1");
    }

    @Deprecated
    public void call2() {
        System.out.println("DeprecatedClass.call2");
    }

    @Deprecated(since = "2.4", forRemoval = true)
    public void call3() {
        System.out.println("DeprecatedClass.call3");
    }
}
```

- `@Deprecated`: 더는 사용을 권장하지 않는 요소이다.
  - `since`: 더 이상 사용하지 않게된 버전 정보
  - `forRemoval`: 미래 버전에 코드가 제거될 예정이다.

```
package annotation.java;

public class DeprecatedMain {
    public static void main(String[] args) {
        System.out.println("DeprecatedMain.main");
```

```
        DeprecatedClass dc = new DeprecatedClass();
        dc.call1();
        dc.call2(); // IDE 경고
        dc.call3(); // IDE 경고(심각)
    }
}
```

- `@Deprecated`만 있는 코드를 사용할 경우 IDE에서 경고를 나타낸다.
- `@Deprecated` + `forRemoval`이 있는 경우 IDE는 빨간색으로 심각한 경고를 나타낸다.

실행 결과

```
DeprecatedMain.main
DeprecatedClass.call1
DeprecatedClass.call2
DeprecatedClass.call3
```

`@Deprecated`는 컴파일 시점에 경고를 나타내지만, 프로그램은 작동한다.

@SuppressWarnings

이름 그대로 경고를 억제하는 애노테이션이다. 자바 컴파일러가 문제를 경고하지만, 개발자가 해당 문제를 잘 알고 있기 때문에, 더는 경고하지 말라고 지시하는 애노테이션이다.

```
package java.lang;

@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, MODULE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    String[] value();
}
```

```
package annotation.java;

import java.util.ArrayList;
import java.util.List;
```

```java
public class SuppressWarningCase {

    @SuppressWarnings("unused")
    public void unusedWarning() {
        // 사용되지 않는 변수 경고 억제
        int unusedVariable = 10;
    }

    @SuppressWarnings("deprecation")
    public void deprecatedMethod() {
        // 더 이상 사용되지 않는 메서드 호출
        java.util.Date date = new java.util.Date();
        int date1 = date.getDate();
    }

    @SuppressWarnings({"rawtypes", "unchecked"})
    public void uncheckedCast() {
        // 제네릭 타입 캐스팅 경고 억제, raw type 사용 경고
        List list = new ArrayList();

        // 제네릭 타입과 관려도니 unchecked 경고
        List<String> stringList = (List<String>)list;
    }

    @SuppressWarnings("all")
    public void suppressAllWarning() {
        // 모든 경고 억제
        java.util.Date date = new java.util.Date();
        date.getDate();
        List list = new ArrayList();
        List<String> stringList = (List<String>)list;
    }
}
```

`@SuppressWarnings`에 사용하는 대표적인 값들은 다음과 같다.

- **all**: 모든 경고를 억제
- **deprecation**: 사용이 권장되지 않는(deprecated) 코드를 사용할 때 발생하는 경고를 억제
- **unchecked**: 제네릭 타입과 관련된 unchecked 경고를 억제
- **serial**: Serializable 인터페이스를 구현할 때 serialVersionUID 필드를 선언하지 않은 경우 발생하는 경고를 억제
- **rawtypes**: 제네릭 타입이 명시되지 않은(raw) 타입을 사용할 때 발생하는 경고를 억제

- **unused**: 사용되지 않는 변수, 메서드, 필드 등을 선언했을 때 발생하는 경고를 억제

정리

자바 백엔드 개발자가 되려면 스프링, JPA 같은 기술은 필수로 배워야 한다. 그런데 처음 스프링이나 JPA 같은 기술을 배우면, 기존에 자바 문법으로는 잘 이해가 안되는 마법 같은 일들이 벌어진다.

이러한 프레임워크들은 리플렉션과 애노테이션을 활용하여 다음의 "마법 같은" 기능들을 제공한다

1. 의존성 주입 (Dependency Injection): 스프링은 리플렉션을 사용하여 객체의 필드나 생성자에 자동으로 의존성을 주입한다. 개발자는 단순히 `@Autowired` 애노테이션만 붙이면 된다.
2. ORM (Object-Relational Mapping): JPA는 애노테이션을 사용하여 자바 객체와 데이터베이스 테이블 간의 매핑을 정의한다. 예를 들어, `@Entity`, `@Table`, `@Column` 등의 애노테이션으로 객체-테이블 관계를 설정한다.
3. AOP (Aspect-Oriented Programming): 스프링은 리플렉션을 사용하여 런타임에 코드를 동적으로 주입하고, `@Aspect`, `@Before`, `@After` 등의 애노테이션으로 관점 지향 프로그래밍을 구현한다.
4. 설정의 자동화: `@Configuration`, `@Bean` 등의 애노테이션을 사용하여 다양한 설정을 편리하게 적용한다.
5. 트랜잭션 관리: `@Transactional` 애노테이션만으로 메서드 레벨의 DB 트랜잭션 처리가 가능해진다.

이러한 기능들은 개발자가 비즈니스 로직에 집중할 수 있게 해주며, 보일러플레이트(지루한 반복) 코드를 크게 줄여준다. 하지만 이 "마법"의 이면에는 리플렉션과 애노테이션을 활용한 복잡한 메타프로그래밍이 숨어 있다.
프레임워크의 동작 원리를 깊이 이해하기 위해서는 리플렉션과 애노테이션에 대한 이해가 필수다. 이를 통해 프레임워크가 제공하는 편의성과 그 이면의 복잡성 사이의 균형을 잡을 수 있으며, 필요에 따라 프레임워크를 효과적으로 커스터마이징하거나 최적화할 수 있게 된다.

스프링이나 JPA 같은 프레임워크들은 이번에 학습한 리플렉션과 애노테이션을 극대화해서 사용한다.
리플렉션과 애노테이션을 배운 덕분에 여러분은 이런 기술이 마법이 아니라, 리플렉션과 애노테이션을 활용한 고급 프로그래밍 기법이라는 것을 이해할 수 있을 것이다. 그리고 이러한 이해를 바탕으로, 프레임워크의 동작 원리를 더 깊이 파악하고 효과적으로 활용할 수 있게 될 것이다.

결론적으로, 자바 백엔드 개발자가 되기 위해 스프링과 JPA를 배우는 과정에서 리플렉션과 애노테이션은 더 이상 어렵고 낯선 개념이 아니라, 프레임워크의 동작 원리를 이해하고 활용하는 데 필수적인 도구임을 알게 될 것이다. 이러한 도구들을 잘 활용하면 개발자로서 한 단계 더 성장하게 될 것이며, 복잡한 문제 상황도 잘 해결할 수 있는 강력한 무기를 얻게 될 것이다.