12. 리플렉션

#0.강의/1.자바로드맵/6.자바-고급2편

리플렉션이 필요한 이유

우리가 앞서 커맨드 패턴으로 만든 서블릿은 아주 유용하지만, 몇 가지 단점이 있다.

- 하나의 클래스에 하나의 기능만 만들 수 있다.
- 새로 만든 클래스를 URL 경로와 항상 매핑해야 한다.

문제1: 하나의 클래스에 하나의 기능만 만들 수 있다.

기능 하나를 만들 때 마다 각각 별도의 클래스를 만들고 구현해야 한다. 이것은 복잡한 기능에서는 효과적이지만, 간단한 기능을 만들 때는 클래스가 너무 많이 만들어지기 때문에 부담스럽다.

```
package was.v5.servlet;

import was.httpserver.HttpRequest;
import was.httpserver.HttpResponse;
import was.httpserver.HttpServlet;

import java.io.IOException;

public class Site1Servlet implements HttpServlet {
    @Override
    public void service(HttpRequest request, HttpResponse response) throws
IOException {
        response.writeBody("<h1>site1</h1>");
    }
}
```

```
package was.v5.servlet;

import was.httpserver.HttpRequest;
import was.httpserver.HttpResponse;
import was.httpserver.HttpServlet;

import java.io.IOException;

public class Site2Servlet implements HttpServlet {
    @Override
    public void service(HttpRequest request, HttpResponse response) throws
IOException {
        response.writeBody("<h1>site2</h1>");
    }
}
```

```
package was.v5.servlet;

import was.httpserver.HttpRequest;
import was.httpserver.HttpResponse;
import was.httpserver.HttpServlet;

import java.io.IOException;

public class SearchServlet implements HttpServlet {
    @Override
    public void service(HttpRequest request, HttpResponse response) throws
IOException {
        String query = request.getParameter("q");

        response.writeBody("<h1>Search</h1>");
        response.writeBody("<ul>");
        response.writeBody("<li>query: " + query + "</li>");
        response.writeBody("</ul>");
    }
}
```

이 문제를 해결할 방법이 없을까?

바로 다음과 같이 하나의 클래스 안에서 다양한 기능을 처리하는 것이다.

```
public class ReflectController {

    public void site1(HttpRequest request, HttpResponse response) {
        response.writeBody("<h1>site1</h1>");
    }

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

    public void search(HttpRequest request, HttpResponse response) {
        String query = request.getParameter("q");

        response.writeBody("<h1>Search</h1>");
        response.writeBody("<ul>");
        response.writeBody("<li>query: " + query + "</li>");
        response.writeBody("</ul>");
    }

}
```

물론 필요하면 클래스를 나눌 수 있게 해도 된다.

- `SiteController`
  - `site1()`
  - `site2()`
- `SearchController`
  - `search()`

이렇게 하면 비슷한 기능을 한 곳에 모을 수 있는 장점도 있고, 작은 기능 하나를 추가할 때 마다 클래스를 계속 만들지 않아도 된다.

문제2: 새로 만든 클래스를 URL 경로와 항상 매핑해야 한다.

```
servletManager.add("/site1", new Site1Servlet());
servletManager.add("/site2", new Site2Servlet());
```

```
servletManager.add("/search", new SearchServlet());
```

- 새로운 기능을 하나 추가할 때 마다 이런 매핑 작업도 함께 추가해야 한다.

그런데 여기서 앞서 본 `ReflectController` 예시를 보자.
URL 경로의 이름과 메서드의 이름이 같다.

- `/site1 -> site1()`
- `/site2 -> site2()`
- `/search -> search()`

만약 URL 경로의 이름과 같은 이름의 메서드를 찾아서 호출할 수 있다면?
예를 들어 `/site1`이 입력되면 `site1()`이라는 메서드를 이름으로 찾아서 호출하는 것이다.
클래스에 있는 메서드의 이름을 찾아서 이렇게 호출할 수 있다면, 번거로운 매핑 작업을 제거할 수 있을 것이다.

자바 프로그램 실행 중에 이름으로 메서드를 찾고, 또 찾은 메서드를 호출하려면 자바의 리플렉션 기능을 먼저 알아야 한다.
리플렉션 기능을 먼저 학습하고, 또 학습한 내용을 기반으로 `ReflectController` 같은 기능을 만들어서 적용해보자.

클래스와 메타데이터

클래스가 제공하는 다양한 정보를 동적으로 분석하고 사용하는 기능을 **리플렉션(Reflection)**이라 한다. 리플렉션을 통해 프로그램 실행 중에 클래스, 메서드, 필드 등에 대한 정보를 얻거나, 새로운 객체를 생성하고 메서드를 호출하며, 필드의 값을 읽고 쓸 수 있다.

리플렉션을 통해 얻을 수 있는 정보는 다음과 같다.

- **클래스의 메타데이터**: 클래스 이름, 접근 제어자, 부모 클래스, 구현된 인터페이스 등.
- **필드 정보**: 필드의 이름, 타입, 접근 제어자를 확인하고, 해당 필드의 값을 읽거나 수정할 수 있다.
- **메서드 정보**: 메서드 이름, 반환 타입, 매개변수 정보를 확인하고, 실행 중에 동적으로 메서드를 호출할 수 있다.
- **생성자 정보**: 생성자의 매개변수 타입과 개수를 확인하고, 동적으로 객체를 생성할 수 있다.

참고 - 리플랙션 용어
"리플렉션(Reflection)"이라는 용어는 영어 단어 "reflect"에서 유래된 것으로, "반사하다" 또는 "되돌아보다"라는 의미를 가지고 있다. 리플렉션은 프로그램이 실행 중에 자기 자신의 구조를 들여다보고, 그 구조를 변경하거나 조작할 수 있

는 기능을 의미한다. 쉽게 말해, 리플렉션을 통해 클래스, 메서드, 필드 등의 메타데이터를 런타임에 동적으로 조사하고 사용할 수 있다. 이는 마치 거울에 비친 자신을 보는 것과 같이, 프로그램이 자기 자신의 내부를 '반사(reflect)'하여 들여다본다는 의미이다.

클래스 메타데이터 조회

```
package reflection.data;

public class BasicData {

    public String publicField;
    private int privateField;

    public BasicData() {
        System.out.println("BasicData.BasicData");
    }

    private BasicData(String data) {
        System.out.println("BasicData.BasicData: " + data);
    }

    public void call() {
        System.out.println("BasicData.call");
    }

    public String hello(String str) {
        System.out.println("BasicData.hello");
        return str + " hello";
    }

    private void privateMethod() {
        System.out.println("BasicData.privateMethod");
    }

    void defaultMethod() {
        System.out.println("BasicData.defaultMethod");
    }

    protected void protectedMethod() {
        System.out.println("BasicData.protectedMethod");
    }
```

```
}
```

- 예제를 위한 기본 클래스이다.

```
package reflection;

import reflection.data.BasicData;

public class BasicV1 {

    public static void main(String[] args) throws ClassNotFoundException {
        // 클래스 메타데이터 조회 방법 3가지

        // 1. 클래스에서 찾기
        Class<BasicData> basicDataClass1 = BasicData.class;
        System.out.println("basicDataClass1 = " + basicDataClass1);

        // 2. 인스턴스에서 찾기
        BasicData basicInstance = new BasicData();
        Class<? extends BasicData> basicDataClass2 = basicInstance.getClass();
        System.out.println("basicDataClass2 = " + basicDataClass2);

        // 3. 문자로 찾기
        String className = "reflection.data.BasicData"; // 패키지명 주의
        Class<?> basicDataClass3 = Class.forName(className);
        System.out.println("basicDataClass3 = " + basicDataClass3);
    }
}
```

실행 결과

```
basicDataClass1 = class reflection.data.BasicData
BasicData.BasicData
basicDataClass2 = class reflection.data.BasicData
basicDataClass3 = class reflection.data.BasicData
```

클래스의 메타데이터는 `Class` 라는 클래스로 표현된다. 그리고 `Class` 라는 클래스를 획득하는 3가지 방법이 있다.

클래스에서 찾기

```
Class<BasicData> basicDataClass1 = BasicData.class
```

클래스명에 `.class`를 사용하면 획득할 수 있다.

인스턴스에서 찾기

```
BasicData helloInstance = new BasicData();
Class<? extends BasicData> basicDataClass2 = helloInstance.getClass();
```

인스턴스에서 `.getClass()` 메서드를 호출하면 획득할 수 있다.
반환 타입을 보면 `Class<? extends BasicData>`로 표현되는데, 실제 인스턴스가 `BasicData` 타입일 수도 있지만, 그 자식 타입일 수도 있기 때문이다.

이해를 돕기 위해 다음 예를 보자.

```
Parent parent = new Child();
Class<? extends Parent> parentClass = parent.getClass();
```

`Parent` 타입을 통해 `getClass()`를 호출했지만, 실제 인스턴스는 `Child`이다. 따라서 제네릭에서 자식 타입도 허용할 수 있도록 `? extends Parent`를 사용한다.

문자로 찾기

```
String className = "reflection.data.BasicData"; // 패키지명 주의
Class<?> basicData3 = Class.forName(className);
```

이 부분이 가장 흥미로운데, 단순히 문자로 클래스의 메타데이터를 조회할 수 있다. 예를 들어서 콘솔에서 사용자 입력으로 원하는 클래스를 동적으로 찾을 수 있다는 뜻이다.

기본 정보 탐색

이렇게 찾은 클래스 메타데이터로 어떤 일들을 할 수 있는지 알아보자.

```
package reflection;
```

```java
import reflection.data.BasicData;

import java.lang.reflect.Modifier;
import java.util.Arrays;

public class BasicV2 {

    public static void main(String[] args) throws ClassNotFoundException {
        Class<BasicData> basicData = BasicData.class;

        System.out.println("basicData.getName() = " + basicData.getName());
        System.out.println("basicData.getSimpleName() = " +
basicData.getSimpleName());
        System.out.println("basicData.getPackage() = " +
basicData.getPackage());

        System.out.println("basicData.getSuperclass() = " +
basicData.getSuperclass());
        System.out.println("basicData.getInterfaces() = " +
Arrays.toString(basicData.getInterfaces()));

        System.out.println("basicData.isInterface() = " +
basicData.isInterface());
        System.out.println("basicData.isEnum() = " + basicData.isEnum());
        System.out.println("basicData.isAnnotation() = " +
basicData.isAnnotation());

        int modifiers = basicData.getModifiers();
        System.out.println("basicData.getModifiers() = " + modifiers);
        System.out.println("isPublic = " + Modifier.isPublic(modifiers));
        System.out.println("Modifier.toString() = " +
Modifier.toString(modifiers));
    }
}
```

실행 결과

```
basicData.getName() = reflection.data.BasicData
basicData.getSimpleName() = BasicData
basicData.getPackage() = package reflection.data
basicData.getSuperclass() = class java.lang.Object
```

```
basicData.getInterfaces() = []
basicData.isInterface() = false
basicData.isEnum() = false
basicData.isAnnotation() = false
basicData.getModifiers() = 1
isPublic = true
Modifier.toString() = public
```

- 클래스 이름, 패키지, 부모 클래스, 구현한 인터페이스, 수정자 정보등 다양한 정보를 획득할 수 있다.

참고로 수정자는 접근 제어자와 비 접근 제어자(기타 수정자)로 나눌 수 있다.

- 접근 제어자: `public`, `protected`, `default` (`package-private`), `private`
- 비 접근 제어자: `static`, `final`, `abstract`, `synchronized`, `volatile` 등

`getModifiers()`를 통해 수정자가 조합된 숫자를 얻고, `Modifier`를 사용해서 실제 수정자 정보를 확인할 수 있다.

메서드 탐색과 동적 호출

클래스 메타데이터를 통해 클래스가 제공하는 메서드의 정보를 확인해보자.

메서드 메타데이터

```
package reflection;

import reflection.data.BasicData;

import java.lang.reflect.Method;

public class MethodV1 {

    public static void main(String[] args) {
        Class<BasicData> helloClass = BasicData.class;

        System.out.println("====== methods() =====");
        Method[] methods = helloClass.getMethods();
        for (Method method : methods) {
```

```
            System.out.println("method = " + method);
        }

        System.out.println("====== declaredMethods() =====");
        Method[] declaredMethods = helloClass.getDeclaredMethods();
        for (Method method : declaredMethods) {
            System.out.println("declaredMethod = " + method);
        }

    }
}
```

- `Class.getMethods()` 또는 `Class.getDeclaredMethods()`를 호출하면 `Method`라는 메서드의 메타 데이터를 얻을 수 있다. 이 클래스는 메서드의 모든 정보를 가지고 있다.

getMethods() vs getDeclaredMethods()

- `getMethods()` : 해당 클래스와 상위 클래스에서 상속된 **모든 public 메서드를 반환**
- `getDeclaredMethods()` : **해당 클래스에서 선언된 모든 메서드를 반환**하며, 접근 제어자에 관계없이 반환. 상속된 메서드는 포함하지 않음

실행 결과

```
====== methods() =====
method = public void reflection.data.BasicData.call()
method = public java.lang.String
reflection.data.BasicData.hello(java.lang.String)
method = public final void java.lang.Object.wait(long,int) throws
java.lang.InterruptedException
method = public final void java.lang.Object.wait() throws
java.lang.InterruptedException
method = public final native void java.lang.Object.wait(long) throws
java.lang.InterruptedException
method = public boolean java.lang.Object.equals(java.lang.Object)
method = public java.lang.String java.lang.Object.toString()
method = public native int java.lang.Object.hashCode()
method = public final native java.lang.Class java.lang.Object.getClass()
method = public final native void java.lang.Object.notify()
method = public final native void java.lang.Object.notifyAll()

====== declaredMethods() =====
```

```
declaredMethod = public void reflection.data.BasicData.call()
declaredMethod = private void reflection.data.BasicData.privateMethod()
declaredMethod = void reflection.data.BasicData.defaultMethod()
declaredMethod = protected void reflection.data.BasicData.protectedMethod()
declaredMethod = public java.lang.String
reflection.data.BasicData.hello(java.lang.String)
```

동적 메서드 호출

`Method` 객체를 사용해서 메서드를 직접 호출할 수 도 있다.

```
package reflection;

import reflection.data.BasicData;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class MethodV2 {

    public static void main(String[] args) throws NoSuchMethodException,
InvocationTargetException, IllegalAccessException {
        // 정적 메서드 호출 - 일반적인 메서드 호출
        BasicData helloInstance = new BasicData();
        helloInstance.call(); // 이 부분은 코드를 변경하지 않는 이상 정적이다.

        // 동적 메서드 호출 - 리플렉션 사용
        Class<? extends BasicData> helloClass = helloInstance.getClass();
        String methodName = "hello";

        // 메서드 이름을 변수로 변경할 수 있다.
        Method method1 = helloClass.getDeclaredMethod(methodName,
String.class);
        Object returnValue = method1.invoke(helloInstance, "hi");
        System.out.println("returnValue = " + returnValue);
    }
}
```

- 리플렉션을 사용하면 매우 다양한 체크 예외가 발생한다.

실행 결과

```
BasicData.BasicData
BasicData.call
BasicData.hello
returnValue = hi hello
```

일반적인 메서드 호출 - 정적

인스턴스의 참조를 통해 메서드를 호출하는 방식이 일반적인 메서드 호출 방식이다.

이 방식은 코드를 변경하지 않는 이상 `call()` 대신 다른 메서드로 변경하는 것이 불가능하다.

`helloInstance.call()`

호출하는 메서드가 이미 코드로 작성되어서 **정적**으로 변경할 수 없는 없는 상태이다.

동적 메서드 호출 - 리플렉션 사용

```
String methodName = "hello";
// 메서드 이름을 변수로 변경할 수 있다.
Method method1 = helloClass.getMethod(methodName, String.class);
Object returnValue = method1.invoke(helloInstance, "hi");
```

리플렉션을 사용하면 동적으로 메서드를 호출할 수 있다.

```
Method method1 = helloClass.getMethod(methodName, String.class)
```

- 클래스 메타데이터가 제공하는 `getMethod()`에 메서드 이름, 사용하는 매개변수의 타입을 전달하면 원하는 메서드를 찾을 수 있다.
- 여기서는 hello라는 이름에 `String` 매개변수가 있는 `hello(String)` 메서드를 찾는다.

```
Object returnValue = method1.invoke(helloInstance, "hi");
```

- `Method.invoke()` 메서드에 실행할 인스턴스와 인자를 전달하면, 해당 인스턴스에 있는 메서드를 실행할 수 있다.
- 여기서는 `BasicData helloInstance = new BasicData()` 인스턴스에 있는 `hello(String)` 메서드를 호출한다.

여기서 메서드를 찾을 때 `helloClass.getMethod(methodName, String.class)`에서 `methodName` 부분

이 String 변수로 되어 있는 것을 확인할 수 있다. `methodName`은 변수이므로 예를 들어 사용자 콘솔 입력을 통해서 얼마든지 호출할 `methodName`을 변경할 수 있다.
따라서 여기서 호출할 메서드 대상은 정적으로 딱 코드에 정해진 것이 아니라, 언제든지 동적으로 변경할 수 있다. 그래서 동적 메서드 호출이라 한다.

동적 메서드 호출 - 예시

동적 메서드 호출을 예시를 통해 확인해보자.

```
package reflection.data;

public class Calculator {

    public int add(int a, int b) {
        return a + b;
    }

    public int sub(int a, int b) {
        return a - b;
    }
}
```

```
package reflection;

import reflection.data.Calculator;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.Scanner;

public class MethodV3 {

    public static void main(String[] args) throws NoSuchMethodException,
InvocationTargetException, IllegalAccessException {
        Scanner scanner = new Scanner(System.in);
        System.out.print("호출 메서드: ");
        String methodName = scanner.nextLine();

        System.out.print("숫자1: ");
```

```
        int num1 = scanner.nextInt();
        System.out.print("숫자2: ");
        int num2 = scanner.nextInt();

        Calculator calculator = new Calculator();
        // 호출할 메서드를 변수 이름으로 동적으로 선택

        Class<? extends Calculator> aClass = Calculator.class;
        Method method = aClass.getMethod(methodName, int.class, int.class);

        Object returnValue = method.invoke(calculator, num1, num2);
        System.out.println("returnValue = " + returnValue);
    }
}
```

실행 결과1

```
호출 메서드: add
숫자1: 1
숫자2: 2
returnValue = 3
```

실행 결과2

```
호출 메서드: sub
숫자1: 10
숫자2: 8
returnValue = 2
```

필드 탐색과 값 변경

리플렉션을 활용해서 필드를 탐색하고 또 필드의 값을 변경하도록 활용해보자.

필드 탐색

```java
package reflection;

import reflection.data.BasicData;

import java.lang.reflect.Field;

public class FieldV1 {

    public static void main(String[] args) {
        Class<BasicData> helloClass = BasicData.class;

        System.out.println("====== fields() =====");
        Field[] fields = helloClass.getFields();
        for (Field field : fields) {
            System.out.println("field = " + field);
        }

        System.out.println("====== declaredFields() =====");
        Field[] declaredFields = helloClass.getDeclaredFields();
        for (Field field : declaredFields) {
            System.out.println("declaredField = " + field);
        }

    }
}
```

실행 결과

```
====== fields() =====
field = public java.lang.String reflection.data.BasicData.publicField

====== declaredFields() =====
declaredField = public java.lang.String reflection.data.BasicData.publicField
declaredField = private int reflection.data.BasicData.privateField
```

fields() vs declaredFields()

앞서 설명한 `getMethods()` vs `getDeclaredMethods()`와 같다.

- `fields()` : 해당 클래스와 상위 클래스에서 상속된 **모든 public 필드를 반환**
- `declaredFields()` : **해당 클래스에서 선언된 모든 필드를 반환**하며, 접근 제어자에 관계없이 반환. 상속된 필

드는 포함하지 않음

필드 값 변경

필드 값 변경 예제를 위해 간단한 사용자 데이터를 만들어보자.

```
package reflection.data;

public class User {
    private String id;
    private String name;
    private Integer age;

    public User() {
    }

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

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }
```

```
    public void setAge(Integer age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "User{" +
                "id='" + id + '\'' +
                ", name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

- 여기서 모든 필드가 `private` 접근 제어자라는 점을 주의해서 살펴보자.

```
package reflection;

import reflection.data.User;

import java.lang.reflect.Field;

public class FieldV2 {

    public static void main(String[] args) throws Exception {
        User user = new User("id1", "userA", 20);
        System.out.println("기존 이름 = " + user.getName());

        Class<? extends User> aClass = user.getClass();
        Field nameField = aClass.getDeclaredField("name");

        // private 필드에 접근 허용, private 메서드도 이렇게 호출 가능
        nameField.setAccessible(true);
        nameField.set(user, "userB");
        System.out.println("변경된 이름 = " + user.getName());
    }
}
```

- 사용자의 이름이 `userA`인데, 리플렉션을 사용해서 `name` 필드에 직접 접근한 다음에 `userB`로 이름을 변경해 보자.

```
Field nameField = aClass.getDeclaredField("name")
```

- `name`이라는 필드를 조회한다.
- 그런데 `name` 필드는 `private` 접근 제어자를 사용한다. 따라서 직접 접근해서 값을 변경하는 것이 불가능하다.

```
nameField.setAccessible(true)
```

- 리플렉션은 `private` 필드에 접근할 수 있는 특별한 기능을 제공한다.
- 참고로 `setAccessible(true)` 기능은 `Method`도 제공한다. 따라서 `private` 메서드를 호출할 수도 있다.

```
nameField.set(user, "userB")
```

- `user` 인스턴스에 있는 `nameField`의 값을 `userB`로 변경한다.

실행 결과

```
기존 이름 = userA
변경된 이름 = userB
```

리플렉션과 주의사항

리플렉션을 활용하면 `private` 접근 제어자에도 직접 접근해서 값을 변경할 수 있다. 하지만 이는 객체 지향 프로그래밍의 원칙을 위반하는 행위로 간주될 수 있다. `private` 접근 제어자는 클래스 내부에서만 데이터를 보호하고, 외부에서의 직접적인 접근을 방지하기 위해 사용된다. 리플렉션을 통해 이러한 접근 제한을 무시하는 것은 캡슐화 및 유지보수성에 악영향을 미칠 수 있다. 예를 들어, 클래스의 내부 구조나 구현 세부 사항이 변경될 경우 리플렉션을 사용한 코드는 쉽게 깨질 수 있으며, 이는 예상치 못한 버그를 초래할 수 있다.

따라서 리플렉션을 사용할 때는 반드시 신중하게 접근해야 하며, 가능한 경우 접근 메서드(예: getter, setter)를 사용하는 것이 바람직하다. 리플렉션은 주로 테스트나 라이브러리 개발 같은 특별한 상황에서 유용하게 사용되지만, 일반적인 애플리케이션 코드에서는 권장되지 않는다. 이를 무분별하게 사용하면 코드의 가독성과 안전성을 크게 저하시킬 수 있다.

그럼 어떤 경우에 사용하면 좋은지 다음 필드 활용에서 알아보자.

리플렉션 - 활용 예제

여러분의 프로젝트에서 데이터를 저장해야 하는데, 저장할 때는 반드시 `null`을 사용하면 안된다고 가정해보자.

이 경우 `null` 값을 다른 기본 값으로 모두 변경해야 한다.

- `String`이 `null`이면 `""`(빈 문자)로 변경한다.
- `Integer`가 `null`이면 `0`으로 변경한다.

활용 예시를 위해 `Team` 클래스를 만들자.

```
package reflection.data;

public class Team {

    private String id;
    private String name;

    public Team() {
    }

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

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
    @Override
    public String toString() {
        return "Team{" +
                "id='" + id + '\'' +
                ", name='" + name + '\'' +
                '}';
    }
}
```

```
package reflection;

import reflection.data.Team;
import reflection.data.User;

public class FieldV3 {

    public static void main(String[] args) throws Exception {
        User user = new User("id1", null, null);
        Team team = new Team("team1", null);
        System.out.println("===== before =====");
        System.out.println("user = " + user);
        System.out.println("team = " + team);

        if (user.getId() == null) {
            user.setId("");
        }
        if (user.getName() == null) {
            user.setName("");
        }
        if (user.getAge() == null) {
            user.setAge(0);
        }

        if (team.getId() == null) {
            team.setId("");
        }
        if (team.getName() == null) {
            team.setName("");
        }
        System.out.println("===== after =====");
        System.out.println("user = " + user);
```

```
        System.out.println("team = " + team);
    }
}
```

실행 결과

```
===== before =====
user = User{id='id1', name='null', age=null}
team = Team{id='team1', name='null'}
===== after =====
user = User{id='id1', name='', age=0}
team = Team{id='team1', name=''}
```

- `User`, `Team` 객체에 입력된 정보 중에 `null` 데이터를 모두 기본 값으로 변경해야 한다고 가정해보자.
  - `String`이 `null`이면 `""`(빈 문자)로 변경한다.
  - `Integer`가 `null`이면 `0`으로 변경한다.

이 문제를 해결하려면 각각의 객체에 들어있는 데이터를 직접 다 찾아서 값을 입력해야 한다. 만약 `User`, `Team` 뿐만 아니라 `Order`, `Cart`, `Delivery` 등등 수 많은 객체에 해당 기능을 적용해야 한다면 매우 많은 번거로운 코드를 작성해야 할 것이다.

이번에는 리플렉션을 활용해서 이 문제를 깔끔하게 해결해보자.

리플렉션을 활용한 필드 기본 값 도입

```
package reflection;

import java.lang.reflect.Field;

public class FieldUtil {

    public static void nullFieldToDefault(Object target) throws
IllegalAccessException {
        Class<?> aClass = target.getClass();
        Field[] declaredFields = aClass.getDeclaredFields();
        for (Field field : declaredFields) {
            field.setAccessible(true);
            if (field.get(target) != null) {
```

```
                continue;
            }

            if (field.getType() == String.class) {
                field.set(target, "");
            } else if (field.getType() == Integer.class) {
                field.set(target, 0);
            }
        }
    }

}
```

- 어떤 객체든 받아서 기본 값을 적용하는 유틸리티 클래스를 만들어보자.

이 유틸리티는 필드의 값을 조사한 다음에 `null` 이면 기본 값을 적용한다.

- `String` 이 `null` 이면 `""` (빈 문자)로 변경한다.
- `Integer` 가 `null` 이면 `0` 으로 변경한다.

```
package reflection;

import reflection.data.Team;
import reflection.data.User;

import java.lang.reflect.Field;

public class FieldV4 {

    public static void main(String[] args) throws Exception {
        User user = new User("id1", null, null);
        Team team = new Team("team1", null);
        System.out.println("===== before =====");
        System.out.println("user = " + user);
        System.out.println("team = " + team);

        FieldUtil.nullFieldToDefault(user);
        FieldUtil.nullFieldToDefault(team);
        System.out.println("===== after =====");
        System.out.println("user = " + user);
        System.out.println("team = " + team);
    }
```

```
}
```

실행 결과

```
===== before =====
user = User{id='id1', name='null', age=null}
team = Team{id='team1', name='null'}
===== after =====
user = User{id='id1', name='', age=0}
team = Team{id='team1', name=''}
```

리플렉션을 사용한 덕분에 `User`, `Team` 뿐만 아니라 `Order`, `Cart`, `Delivery` 등등 수 많은 객체에 매우 편리하게 기본 값을 적용할 수 있게 되었다.
이처럼 리플렉션을 활용하면 기존 코드로 해결하기 어려운 공통 문제를 손쉽게 처리할 수도 있다.

생성자 탐색과 객체 생성

리플렉션을 활용하면 생성자를 탐색하고, 또 탐색한 생성자를 사용해서 객체를 생성할 수 있다.

생성자 탐색

```
package reflection;

import java.lang.reflect.Constructor;

public class ConstructV1 {

    public static void main(String[] args) throws Exception {
        Class<?> aClass = Class.forName("reflection.data.BasicData");

        System.out.println("====== constructors() =====");
        Constructor<?>[] constructors = aClass.getConstructors();
        for (Constructor<?> constructor : constructors) {
            System.out.println(constructor);
```

```
        }

        System.out.println("====== declaredConstructors() =====");
        Constructor<?>[] declaredConstructors =
aClass.getDeclaredConstructors();
        for (Constructor<?> constructor : declaredConstructors) {
            System.out.println(constructor);
        }
    }
}
```

실행 결과

```
====== constructors() =====
public reflection.data.BasicData()

====== declaredConstructors() =====
public reflection.data.BasicData()
private reflection.data.BasicData(java.lang.String)
```

생성자 활용

```
package reflection;

import java.lang.reflect.Constructor;
import java.lang.reflect.Method;

public class ConstructV2 {

    public static void main(String[] args) throws Exception {
        Class<?> aClass = Class.forName("reflection.data.BasicData");

        Constructor<?> constructor =
aClass.getDeclaredConstructor(String.class);
        constructor.setAccessible(true);
        Object instance = constructor.newInstance("hello");
        System.out.println("instance = " + instance);
```

```
        Method method1 = aClass.getDeclaredMethod("call");
        method1.invoke(instance);
    }
}
```

실행 결과

```
BasicData.BasicData: hello
instance = reflection.data.BasicData@7a81197d
BasicData.call
```

`Class.forName("reflection.data.BasicData")`을 사용해서 클래스 정보를 동적으로 조회했다.

`getDeclaredConstructor(String.class)`: 생성자를 조회한다.

- 여기서는 매개변수로 `String`을 사용하는 생성자를 조회한다.

`constructor.setAccessible(true)`를 사용해서 private 생성자를 접근 가능하게 만들었다.

```
private BasicData(String data) {
    System.out.println("BasicData.BasicData: " + data);
}
```

```
Object instance = constructor.newInstance("hello")
```

- 찾은 생성자를 사용해서 객체를 생성한다. 여기서는 "hello"라는 인자를 넘겨준다.

```
Method method1 = aClass.getDeclaredMethod("call");
method1.invoke(instance);
```

- 앞서 생성한 인스턴스에 `call`이라는 이름의 메서드를 동적으로 찾아서 호출한다.

이번 예제를 잘 보면, 클래스를 동적으로 찾아서 인스턴스를 생성하고, 메서드도 동적으로 호출했다. 코드 어디에도 `BasicData`의 타입이나 `call()` 메서드를 직접 호출하는 부분을 직접 코딩하지 않았다.

클래스를 찾고 생성하는 방법도, 그리고 생성한 클래스의 메서드를 호출하는 방법도 모두 동적으로 처리한 것이다.

참고

여러분이 스프링 프레임워크나 다른 프레임워크 기술들을 사용해보면, 내가 만든 클래스를 프레임워크가 대신 생성해 줄 때가 있다. 그때가 되면 방금 학습한 리플렉션과 동적 객체 생성 방법들이 떠오를 것이다.

HTTP 서버6 - 리플렉션 서블릿

우리가 앞서 커맨드 패턴으로 만든 서블릿은 아주 유용하지만, 몇가지 단점이 있다.

- 하나의 클래스에 하나의 기능만 만들 수 있다.
- 새로 만든 클래스를 URL 경로와 항상 매핑해야 한다.

이제 리플렉션의 기본 기능을 학습했으니, 처음에 설명한 리플렉션을 활용한 서블릿 기능을 만들어보자.
개발자는 다음과 같이 간단한 기능만 만들면 된다.

```
public class ReflectController {

    public void site1(HttpRequest request, HttpResponse response) {
        response.writeBody("<h1>site1</h1>");
    }

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

    public void search(HttpRequest request, HttpResponse response) {
        String query = request.getParameter("q");

        response.writeBody("<h1>Search</h1>");
        response.writeBody("<ul>");
        response.writeBody("<li>query: " + query + "</li>");
        response.writeBody("</ul>");
    }

}
```

- 개발자는 이런 Xxx컨트롤러라는 기능만 개발하면 된다.
- 이러한 컨트롤러의 메서드를 리플렉션으로 읽고 호출할 것이다.

참고 - 컨트롤러 용어

컨트롤러는 애플리케이션의 제어 흐름을 '제어(control)'한다. 요청을 받아 적절한 비즈니스 로직을 호출하고, 그 결과를 뷰에 전달하는 등의 작업을 수행한다.

URL 경로의 이름과 메서드의 이름이 같다.

- `/site1 -> site1()`
- `/site2 -> site2()`
- `/search -> search()`

리플렉션을 활용하면 메서드 이름을 알 수 있다. 예를 들어 `/site1`이 입력되면 `site1()`이라는 메서드를 이름으로 찾아서 호출하는 것이다. 이렇게 하면 번거로운 매핑 작업을 제거할 수 있다.

물론 필요하면 서로 관련된 기능은 하나의 클래스로 모으도록, 클래스를 나눌 수 있게 해도 된다.

- `SiteController`
  - `site1()`
  - `site2()`
- `SearchController`
  - `search()`

리플렉션을 처리하는 서블릿 구현

앞서 설명했듯이 서블릿은 자바 진영에서 이미 표준으로 사용하는 기술이다. 따라서 서블릿은 그대로 사용하면서 새로운 기능을 구현해보자. (물론 우리가 만든 서블릿은 자바 표준은 아니지만, 여기서는 자바 표준 서블릿을 사용한다고 가정하자)

앞서 만든 HTTP 서버에서 `was.httpserver` 패키지는 다른 곳에서 제공하는 변경할 수 없는 라이브러리라고 가정하자. 우리는 `was.httpserver`의 코드를 전혀 변경하지 않고, 그대로 재사용하면서 기능을 추가하겠다.

v5 코드

- HTTP 서버와 관련된 부분 - `was.httpserver` 패키지
  - `HttpServer`, `HttpRequestHandler`, `HttpRequest`, `HttpResponse`
  - `HttpServlet`, `HttpServletManager`
  - `InternalErrorServlet`, `NotFoundServlet` (`was.httpserver.servlet` 패키지)
- 서비스 개발을 위한 로직 - `v5.servlet` 패키지

- `HomeServlet`
- `Site1Servlet`
- `Site2Servlet`
- `SearchServlet`

컨트롤러

```
package was.v6;

import was.httpserver.HttpRequest;
import was.httpserver.HttpResponse;

public class SiteControllerV6 {

    public void site1(HttpRequest request, HttpResponse response) {
        response.writeBody("<h1>site1</h1>");
    }

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

}
```

- `/site1`, `/site2`를 처리한다.
- `site1()`, `site2()`는 단순하고 서로 비슷한 기능을 제공한다고 가정하겠다.

```
package was.v6;

import was.httpserver.HttpRequest;
import was.httpserver.HttpResponse;

public class SearchControllerV6 {

    public void search(HttpRequest request, HttpResponse response) {
        String query = request.getParameter("q");

        response.writeBody("<h1>Search</h1>");
        response.writeBody("<ul>");
        response.writeBody("<li>query: " + query + "</li>");
```

```
            response.writeBody("</ul>");
        }
    }
```

- `/search`를 처리한다.
- `search()`는 복잡하고 단독 기능을 제공한다고 가정하겠다.

참고로 XxxController에서 호출 대상이 되는 메서드는 반드시 `HttpRequest`, `HttpResponse`를 인자로 받아야 한다. 이 부분은 뒤에서 설명한다.

기존 코드에서 이런 컨트롤러들을 어떻게 호출하도록 만들 수 있을까?
바로 서블릿에 이런 기능을 구현하면 된다.

리플렉션 서블릿

```
package was.httpserver.servlet.reflection;

import was.httpserver.HttpRequest;
import was.httpserver.HttpResponse;
import was.httpserver.HttpServlet;
import was.httpserver.PageNotFoundException;

import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.List;

public class ReflectionServlet implements HttpServlet {

    private final List<Object> controllers;

    public ReflectionServlet(List<Object> controllers) {
        this.controllers = controllers;
    }

    @Override
    public void service(HttpRequest request, HttpResponse response) throws
IOException {
        String path = request.getPath();
```

```
        for (Object controller : controllers) {
            Class<?> aClass = controller.getClass();
            Method[] methods = aClass.getDeclaredMethods();
            for (Method method : methods) {
                String methodName = method.getName();
                if (path.equals("/" + methodName)) {
                    invoke(controller, method, request, response);
                    return;
                }
            }
        }
        throw new PageNotFoundException("request=" + path);
    }

    private static void invoke(Object controller, Method method, HttpRequest
request, HttpResponse response) {
        try {
            method.invoke(controller, request, response);
        } catch (InvocationTargetException | IllegalAccessException e) {
            throw new RuntimeException(e);
        }
    }
}
```

- `was.httpserver.servlet.reflection` 패키지 위치를 주의하자. 다른 프로젝트에서도 필요하면 사용할 수 있다.
- `List<Object> controllers`: 생성자를 통해 여러 컨트롤러들을 보관할 수 있다.

이 서블릿은 요청이 오면 모든 컨트롤러를 순회한다. 그리고 선언된 메서드 정보를 통해 URL의 요청 경로와 메서드 이름이 맞는지 확인한다. 만약 메서드 이름이 맞다면 `invoke`를 통해 해당 메서드를 동적으로 호출한다. 이때 `HttpRequest`, `HttpResponse` 정보도 함께 넘겨준다. 따라서 대상 메서드는 반드시 `HttpRequest`, `HttpResponse`를 인자로 받아야 한다.
이미 앞서 리플렉션에서 학습한 내용들이기 때문에 이해하기 어렵지는 않을 것이다.

```
package was.v6;

import was.httpserver.HttpServer;
import was.httpserver.HttpServlet;
import was.httpserver.ServletManager;
```

```java
import was.httpserver.servlet.DiscardServlet;
import was.v5.servlet.HomeServlet;
import was.httpserver.servlet.reflection.ReflectionServlet;

import java.io.IOException;
import java.util.List;

public class ServerMainV6 {

    private static final int PORT = 12345;

    public static void main(String[] args) throws IOException {
        List<Object> controllers = List.of(new SiteControllerV6(), new
SearchControllerV6());
        HttpServlet reflectionServlet = new ReflectionServlet(controllers);

        ServletManager servletManager = new ServletManager();
        servletManager.setDefaultServlet(reflectionServlet);
        servletManager.add("/", new HomeServlet());
        servletManager.add("/favicon.ico", new DiscardServlet());

        HttpServer server = new HttpServer(PORT, servletManager);
        server.start();
    }
}
```

실행 결과

- 실행 결과는 기존과 같다.

new ReflectionServlet(controllers)

리플렉션 서블릿을 생성하고, 사용할 컨트롤러들을 인자로 전달한다.

servletManager.setDefaultServlet(reflectionServlet)

- 이 부분이 중요하다. 리플랙션 서블릿을 기본 서블릿으로 등록하는 것이다. 이렇게 되면 다른 서블릿에서 경로를 찾지 못할 때 우리가 만든 리플렉션 서블릿이 항상 호출된다!
- 그리고 다른 서블릿은 등록하지 않는다. 따라서 항상 리플렉션 서블릿이 호출된다.
- 그런데 아쉽게도 `HomeServlet`은 등록해야 한다. 왜냐하면 `/`라는 이름은 메서드 이름으로 매핑할 수 없기 때문이다.
    - `HomeServlet`은 여기서 크게 중요한 부분은 아니므로 `was.v5.servlet`에 있는 클래스를 import 해서 사용하자.

- `/favicon.ico`도 마찬가지로 메서드 이름으로 매핑할 수 없다. 왜냐하면 `favicon.ico`라는 이름으로 메서드를 만들 수 없기 때문이다.

작동 순서

- 웹 브라우저가 `/site1`을 요청한다.
- 서블릿 매니저는 등록된 서블릿 중에 `/site1`을 찾는다.
- 등록된 서블릿 중에 `/site1`을 찾을 수 없다.
  - 등록된 서블릿은 `/=HomeServlet, /favicon.ico=DiscardServlet` 이다.
- 기본 서블릿(default Servlet)으로 등록한 `ReflectionServlet`을 호출한다.
- `ReflectionServlet`은 컨트롤러를 순회하면서 `site1()`이라는 이름의 메서드를 찾아서 호출한다.
  - 등록한 `SiteControllerV6`, `SearchControllerV6`을 순회한다.
  - 이때 `HttpRequest`, `HttpResponse`도 함께 전달한다.
- `site1()` 메서드가 실행된다.

정리

기존 HTTP 서버의 코드를 전혀 변경하지 않고, 서블릿만 잘 구현해서 완전히 새로운 기능을 도입했다.

덕분에 앞서 커맨드 패턴으로 만든 서블릿의 단점을 해결할 수 있었다.

- 하나의 클래스에 하나의 기능만 만들 수 있다.
- 새로 만든 클래스를 URL 경로와 항상 매핑해야 한다.

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 다른 이름의 메서드를 호출하고 싶다면 어떻게 해야할까?
  - 예를 들어서 메서드 이름은 더 자세히 적고 싶을 수 있다.
- 앞서 `/`, `/favicon.ico`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?

정리