Luke's Language로 파싱된 이펙티브 자바

터부 Taboo

제작. 홍성혁

객체는 인터페이스를 사용해 참조하라

Item 64 - 구현을 드러내지 마

적합한 인터페이스가 있다면, 인터페이스를 써라

```
// 좋은 예. 인터페이스를 타입으로 사용했다.
Set<Son> sonSet = new LinkedHashSet<>();
```

```
// 나쁜 예. 클래스를 타입으로 사용했다!
LinkedHashSet<Son> sonSet = new LinkedHashSet<>();
```

객체는 인터페이스를 사용해 참조하라

Item 64 - 구현을 드러내지 마

적합한 인터페이스가 있다면, 인터페이스를 써라

```
// 좋은 예. 인터페이스를 타입으로 사용했다.
Set<Son> sonSet = new LinkedHashSet<>();
```

```
// 나쁜 예. 클래스를 타입으로 사용했다!
LinkedHashSet<Son> sonSet = new LinkedHashSet<>();
```

How 보다는 **What**

=> Set이 어떻게 구현되었는지 알 필요가 있는가?

=> 구현은 얼마든지 대체할 수 있다

Item 65 - 리플렉션 쓰지마

리플렉션

java.lang.reflect

반영(reflection)은 컴퓨터 과학 용어로, 컴퓨터 프로그램에서 런타임 시점에 사용되는 자신의 구조와 행위를 관리(type introspection)하고 수정할 수 있는 프로세스를 의미한다. “type introspection”은 객체 지향 프로그램언어에서 런타임에 객체의 형(type)을 결정할 수 있는 능력을 의미한다.

많은 컴퓨터 아키텍처에서, 프로그램 명령은 데이터와 같이 메모리에 적재되므로, 명령과 데이터 사이의 구분은 단지 컴퓨터와 프로그램 언어에 의하여 어떻게 정보가 처리되느냐의 문제이다. 일반적으로, 명령은 실행되고, 데이터는 (명령의 자료로서) 처리된다. 그렇지만, 어떤 언어에서는, 프로그램은 명령 역시 데이터와 같은 방식으로 처리할 수 있기 때문에, reflective 수정이 가능하게 된다. 가장 일반적으로 반영은 스몰토크, 스크립트 언어와 같이 높은 수준의 가상 머신 프로그램 언어에서 주로 많이 사용되고, 자바, C 언어와 같이 선언적이거나 정적 형식의 프로그램 언어에서는 드물게 사용된다.

Item 65 - 리플렉션 쓰지마

리플렉션

java.lang.reflect

반영(reflection)은 컴퓨터 과학 용어로, 컴퓨터 프로그램에서 런타임 시점에 사용되는 자신의 구조와 행위를 관리(type introspection)하고 수정할 수 있는 프로세스를 의미한다. “type introspection”은 객체 지향 프로그램언어에서 런타임에 객체의 형(type)을 결정할 수 있는 능력을 의미한다.

객체지향 프로그래밍(Object-Oriented Programming) 언어(예: Java)에서, 리플렉션(reflection)은 컴파일 시간에 인터페이스, 필드, 메서드의 이름을 알지 못해도 런타임에 클래스, 인터페이스, 필드, 메서드를 검사(inspect)할 수 있도록 합니다. 또한 새로운 객체를 생성(instantiation)하고 메서드를 호출(invocation)할 수 있게 해줍니다.

리플렉션은 소프트웨어 테스트(예: 모의 객체(mock object)의 런타임 생성/인스턴스화)를 위한 구성 요소로 자주 사용됩니다.

리플렉션은 메타프로그래밍(metaprogramming)의 핵심 전략이기도 합니다.

리플렉션보다는 인터페이스를 사용하라

Item 65 - 리플렉션 쓰지마

리플렉션

java.lang.reflect

```
import java.lang.reflect.Method;

// Without reflection
Foo foo = new Foo();
foo.hello();

// With reflection
try {
    Object foo = Foo.class.getDeclaredConstructor().newInstance();

    Method m = foo.getClass().getDeclaredMethod("hello", new Class<?>[0]);
    m.invoke(foo);
} catch (ReflectiveOperationException ignored) {}
```

리플렉션보다는 인터페이스를 사용하라

Item 65 - 리플렉션 쓰지마

리플렉션

java.lang.reflect

이렇게 짱 좋은데 왜 쓰지 말란거야?

Item 65 - 리플렉션 쓰지마

리플렉션

java.lang.reflect

이렇게 짱 좋은데 왜 쓰지 말란거야

- **컴파일타임 타입 검사가 주는 이점을 하나도 누릴 수 없다.** 예외 검사도 마찬가지다. 프로그램이 리플렉션 기능을 써서 존재하지 않는 혹은 접근할 수 없는 메서드를 호출하려 시도하면 (주의해서 대비 코드를 작성해두지 않았다면) 런타임 오류가 발생한다.

- **리플렉션을 이용하면 코드가 지저분하고 장황해진다.** 지루한 일이고, 읽기도 어렵다.

- **성능이 떨어진다.** 리플렉션을 통한 메서드 호출은 일반 메서드 호출보다 훨씬 느리다. 고려해야 하는 요소가 많아 정확한 차이는 이야기하기 어렵지만, 내 컴퓨터에서 입력 매개변수가 없고 `int`를 반환하는 메서드로 실험해보니 11배나 느렸다.

리플렉션보다는 인터페이스를 사용하라

Item 65 - 리플렉션 쓰지마

리플렉션

java.lang.reflect

이렇게 짱 좋은데 왜 쓰지 말란거야?

코드 65-1 리플렉션으로 생성하고 인터페이스로 참조해 활용한다.

```
public static void main(String[] args) {
    // 클래스 이름을 Class 객체로 변환
    Class<? extends Set<String>> cl = null;
    try {
        cl = (Class<? extends Set<String>>) // 비검사 형변환!
                Class.forName(args[0]);
    } catch (ClassNotFoundException e) {
        fatalError("클래스를 찾을 수 없습니다.");
    }

    // 생성자를 얻는다.
    Constructor<? extends Set<String>> cons = null;
    try {
        cons = cl.getDeclaredConstructor();
    } catch (NoSuchMethodException e) {
        fatalError("매개변수 없는 생성자를 찾을 수 없습니다.");
    }
    // 집합의 인스턴스를 만든다.
    Set<String> s = null;
    try {
        s = cons.newInstance();
    } catch (IllegalAccessException e) {
        fatalError("생성자에 접근할 수 없습니다.");
    } catch (InstantiationException e) {
        fatalError("클래스를 인스턴스화할 수 없습니다.");
    } catch (InvocationTargetException e) {
        fatalError("생성자가 예외를 던졌습니다: " + e.getCause());
    } catch (ClassCastException e) {
        fatalError("Set을 구현하지 않은 클래스입니다.");
    }

    // 생성한 집합을 사용한다.
    s.addAll(Arrays.asList(args).subList(1, args.length));
    System.out.println(s);
}

private static void fatalError(String msg) {
    System.err.println(msg);
    System.exit(1);
}
```

리플렉션보다는 인터페이스를 사용하라

Item 65 - 리플렉션 쓰지마

리플렉션

java.lang.reflect

이렇게 짱 좋은데 왜 쓰지 말란거야?

드물긴 하지만, 리플렉션은 런타임에 존재하지 않을 수도 있는 다른 클래스, 메서드, 필드와의 의존성을 관리할 때 적합하다. 이 기법은 버전이 여러 개 존재하는 외부 패키지를 다룰 때 유용하다. 가동할 수 있는 최소한의 환경, 즉 주로 가장 오래된 버전만을 지원하도록 컴파일한 후, 이후 버전의 클래스와 메서드 등은 리플렉션으로 접근하는 방식이다. 이렇게 하려면 접근하려는 새로운 클래스나 메서드가 런타임에 존재하지 않을 수 있다는 사실을 반드시 감안해야 한다. 즉, 같은 목적을 이룰 수 있는 대체 수단을 이용하거나 기능을 줄여 동작하는 등의 적절한 조치를 취해야 한다.

핵심 정리

리플렉션은 복잡한 특수 시스템을 개발할 때 필요한 강력한 기능이지만, 단점도 많다. 컴파일타임에는 알 수 없는 클래스를 사용하는 프로그램을 작성한다면 리플렉션을 사용해야 할 것이다. 단, 되도록 객체 생성에만 사용하고, 생성한 객체를 이용할 때는 적절한 인터페이스나 컴파일타임에 알 수 있는 상위 클래스로 형변환해 사용해야 한다.

네이티브 메서드는 신중히 사용하라

Item 66 - 자바만 써

자바 네이티브 인터페이스

자바 네이티브 인터페이스(Java Native Interface, JNI)는 자바 가상 머신(JVM)위에서 실행되고 있는 자바코드가 네이티브 응용 프로그램(하드웨어와 운영 체제 플랫폼에 종속된 프로그램들) 그리고 C, C++ 그리고 어샘블리 같은 다른 언어들로 작성된 라이브러리들을 호출하거나 반대로 호출되는 것을 가능하게 하는 프로그래밍 프레임워크이다.

“네이티브 인터페이스” 가 뭔데?

네이티브 메서드는 신중히 사용하라

Item 66 - 자바만 써

자바 네이티브 인터페이스

자바 네이티브 인터페이스(Java Native Interface, JNI)는 자바 가상 머신(JVM)위에서 실행되고 있는 자바코드가 네이티브 응용 프로그램(하드웨어와 운영 체제 플랫폼에 종속된 프로그램들) 그리고 C, C++ 그리고 어샘블리 같은 다른 언어들로 작성된 라이브러리들을 호출하거나 반대로 호출되는 것을 가능하게 하는 프로그래밍 프레임워크이다.

“네이티브 인터페이스” 가 뭔데?
네이티브 하다는 것은 상대적인 것

4층 - 자바 런타임

3층 - C/C++ <-- 네이티브

2층 - 어셈블리

1층 - 기계어

네이티브 메서드는 신중히 사용하라

Item 66 - 자바만 써

자바 네이티브 인터페이스, 왜 쓰지마?

1. 메모리 훼손에 안전하지 않다.
2. 이식성이 낮다.
3. 디버깅이 어렵다.
4. 주의하지 않으면 속도가 느려진다.
5. 가비지 컬렉터가 네이티브 메모리를 회수 할 수 없으며, 추적도 불가능하다.
6. 자바와 네이티브 코드의 경계를 넘을때마다 비용이 추가된다.
7. 접착 코드도 따로 만들어야 한다.

E.O.D.