Prefer dependecy injection to hardwiring resources

개요

많은 class가 하나 이상의 resources에 의존합니다.

spellchecker 예시를 들어볼텐데, 이 예시에선 spellchecker가 dictionary라는 자원에 의존합니다.

Base design

1. static final resource

```
public class SpellChecker {
  private static final Lexicon dictionary = ...;

  private SpellChecker() {} // Noninstantiable

  public static boolean isValid(String word) { ... }
  public static List<String> suggestions(String typo) { ... }
}
```

2. single-ton spellChecker

```
public class SpellChecker {
  private final Lexicon dictionary = ...;

  private SpellChecker(...) {}
  public static INSTANCE = new SpellChecker(...);

  public static boolean isValid(String word) { ... }
  public static List<String> suggestions(String typo) { ... }
}
```

문제점

이 둘 다 모두 좋지 않은 접근입니다. 왜냐하면! **이들은 오직 하나의 dictionary만 사용할 수 있다고 보기 때문이죠.**

현실에서는 언어 별로 dictionary가 있을 수 있고, 특별한 dictionary가 있을 수 있습니다. 또는 test를 위해 특정 dictionary를 집어넣을 수도 있죠.

side step

1. dictionary를 변경가능하도록 하라

방법

1. final static field를 static field로 선언합니다.
2. static field를 변경할 수 있는 method를 추가합니다.

문제점

이는 concurrent 환경에서 문제가 됩니다.

사용하는 자원에 따라 동작이 바뀌는 상황에서는 static utiliy class나 singleton 모두 적합하지 않습니다.

문제의 포인트

여러 Spellchecker의 여러 instance를 지원하고, 각각의 spellChecker는 client가 원하는 dictionary를 집어넣을 수 있도록 해야 합니다.(책의 번역과 다른 생각)

이것을 가능하게 해주는 것이 바로!! dependency injection 형태입니다.

dictionary는 spellchecker의 dependency이고, 이것은 spellChecker가 생성될 때 주입되어야 합니다.

Dendency Injection

```
public class SpellChekcer {
  private final Lexicon dictionary;

  public SpellChecker(Lexicon dictionary) {
    this.dictionary = Objects.requireNonNull(dictionary);
  }

  public boolean isValid(String word) { ... }
  public List<String> suggestions(String typo) { ... }
}
```

장점

1. DI는 resources가 많더라도 사용가능합니다.
2. immutability를 보장합니다.

   따라서 여러 client가 독립적인 objects를 공유합니다.
3. DI는 동일하게 생성자나, static factories나 builders에도 적용가능합니다.

변형 - Factory Method Pattern

정의

resource factory를 constructor에 넘겨주는 형태입니다.

Supplier<T>

Java 8에서 Supplier<T>가 정확히 그 역할을 하죠.

이는 Supplier에서 생성하는 타입 중 sub type만을 생성하도록 `Supllier<? extends T>` 형태를 취합니다.

예시

```
Mosaic create(Supplier<? extends Tile> tileFactory) { ... }
```

각 tile을 생성하도록 client-provided factory을 사용하는 mosaic을 주입하는 형태입니다.

framework

DI가 아무리 유연하고 테스트하기 쉬워도 매우 큰 프로젝트에서는 코드를 어지럽게 만들 수 있습니다. 따라서 DIframework들이 존재합니다.

요약

resource에 따라 행동이 바뀌는 class를 생성할 때 singleton이나 static utility method를 생성하지 마세요!

그리고 직접적으로 resource를 생성하지마세요!

대신에 resource나, factories를 constructor에 parameter로 넣길 바랍니다.