2장 - 아이템 5번

🎯 자원에 의존하는 클래스는 전역 객체로 만들면 문제가 생긴다

클래스가 내부적으로 하나 이상의 자원에 의존하고, 그 자원이 클래스 동작에 영향을 준다면 싱글턴 과 정적 유틸리티 클래스는 사용하지 않는 것이 좋다.

```
public class SpellChecker {
    private final Lexicon dictionary = ...;  // 자원에 의존함 (dictionary)

    private SpellChecker(..) {} //객체 생성 방지 (private 생성자)

    public static SpellChecker INSTANCE = new SpellChecker(..); // 싱글턴

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

1. **사전(Lexicon) 을 바꿀 수 없다**
    - SpellChecker 는 사전을 내부에서 직접 생성해버린다 `private final Lexicon dictionary = ..;`
    - `static` 프로그램 시작과 동시에 딱 하나만 만들어진다. 클래스 단위로 공유 되고, `new` 추가 인스턴스를 생성 불가
    - `SpellChecker(..)` 생성자에서 Lexicon 사전을 내부에서 결정한다.
    - 프로그램 전체에서 **SpellChecker 객체는 단 하나만 존재하고 그 객체는 생성할 때 넣어준 사전만 사용하** 기에
      나중에 “다른 사전으로 바꾸고 싶다” 거나 “테스트용 사전 넣고 싶다” 는 불가능
2. **테스트 불가능하다.**
    - `private` 생성자 → 외부에서 호출 불가
    - 테스트용 `Mock Lexicon` (테스트용 가짜 사전) 을 넣어 다른 객체를 만들수 없다. **단위 테스트 어려움**
    - 자원(Lexicon) 에 의존하는 클래스에 `private` 생성자 / 싱글턴 떄문에 의존성을 주입 할 수 없다. **유연성 상실**

💡 의존성을 바깥으로 분리하여 외부로부터 주입 받도록 해야한다

이 자원들을 클래스가 직접 만들게 해서는 안된다. 대신 필요한 자원을 (혹은 그 자원을 만들어주는 factory)를 생성자에 (혹은 정적 factory나 빌더에) 넘겨주자

> factory란,
> 호출할 때마다 특정 타입의 인스턴스를 반복해서 만들어주는 객체

1. **의존성 주입 (DI) + 불변 객체**

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

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

- `private final Lexicon dictionary;` 한번 초기화하면 변경 불가 **불변**
  여러 클라이언트가 공유해도 안전하다
- `this.dictionary = Objects.requireNonNull(dictionary);` 외부에서 사전을 주입받아 멤버 변수에 저장
  매개변수가 null 이면 NPE 발생 **안전하게 non-null 객체만 맴버 변수에 넣도록 보장**
- 다양한 Lexicon 구현체 사용이 가능하고, 테스트용 Mock Lexicon 사용 가능
- 여러 SpellChecker 인스턴스를 만들어 다른 사전으로 동시에 사용 가능

2. **factory 매서드 패턴**

```
private SpellChecker(Supplier<Lexicon> dictionary) {
    this.dictionary = Objects.requireNonNull(dictionary.get());
}
```

- `Supplier<Lexicon>` Java 8 부터 제공되는 함수형 인터페이스 `get()` T 타입 객체를 매번 생성하거나 제공
- factory 역할을 수행
- `Supplier` 을 사용하여 사전(Lexicon)을 즉시 생성하지 않고 필요할때만 생성 가능하다. 생성 시점과 방식을 외부에서
  컨트롤 할 수 있다 **유연성 증가**
- `Supplier<T>` 를 입력으로 받는 메서드는 **한정적 와일드카드 타입**을 사용해
  `Supplier<? extends Tile>` factory 에서 반환할 객체의 하위 타입도 허용 / 유연한 타입 매개변수 제한

🛠️ 사용 예제

```
public class SpellChecker {
    private final Lexicon dictionary;  // final : 불변

    private SpellChecker(Supplier<Lexicon> dictionary) {
        this.dictionary = Objects.requireNonNull(dictionary);
    }

    public boolean isValid(String word) { return true; }
    public List<String> suggestions(String typo) { return null; }
```

```
    public static void main(String[] args) {
        Lexicon lexicon = new KoreanDictionary(); //객체 생성
            //외부에서 사전을 주입(DI)하여 SpellChecker가 그 자원에 의존하도록 만든 부분
        SpellChecker spellChecker = new SpellChecker(() -> lexicon);
        spellChecker.isValid("hello");
    }
}

interface Lexicon{}

class KoreanDictionary implements Lexicon {}
class TestDictionary implements Lexicon {} // test가 가능해짐
```

- `Supplier<Lexicon> dictionary` 를 사용하여 `dictionary` 대신 f**actory 역할을 하는 Supplier**를 받음
- `dictionary.get()` → Lexicon 객체를 제공
- 객체 생성 시점과 방식 제어 가능하다 **유연성 제공**
- 실제 사전이 없어도 테스트 가능 **단위 테스트 용이**
- 자원을 생성자에 넣어주는 것 자체가 의존성 주입 → **코드의 유연성과 재사용성 향상**