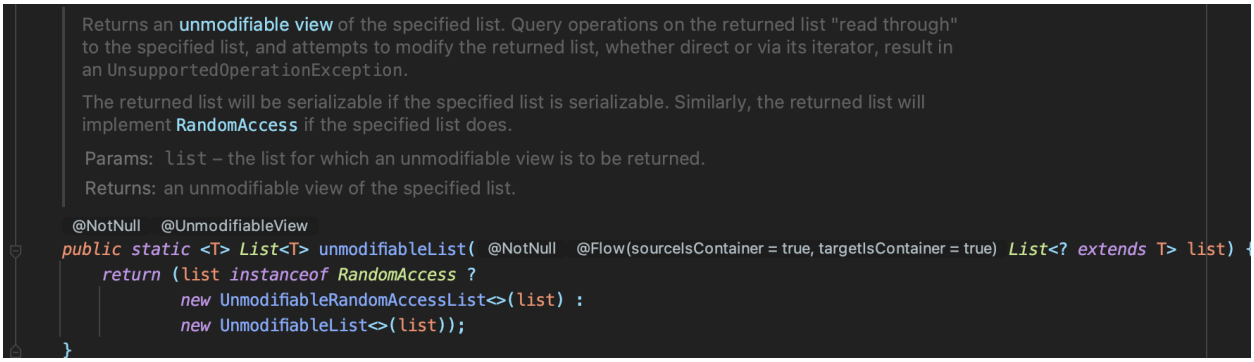
2022.09.08

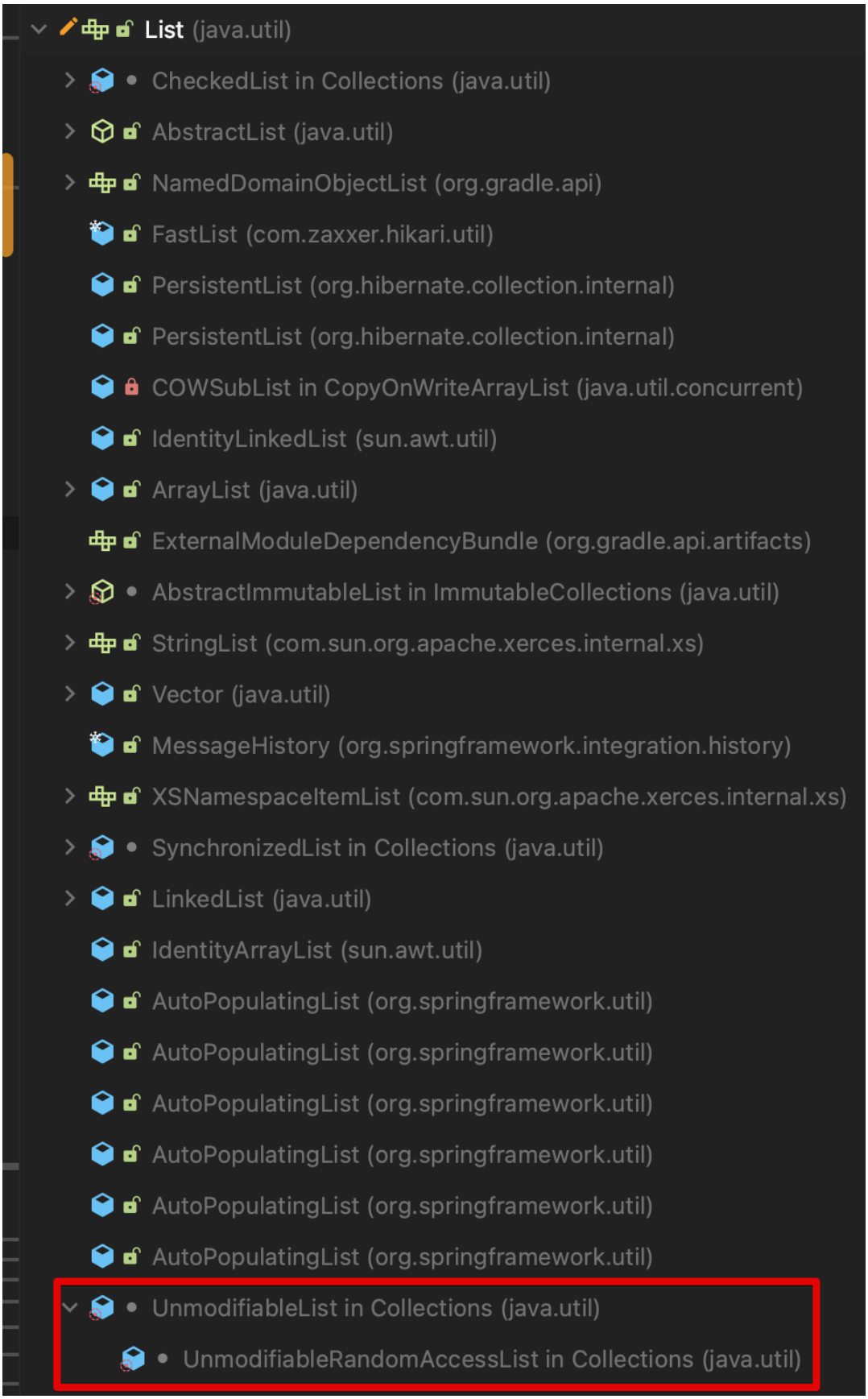
아이템 1. 생성자 대신 정적 팩터리 메서드를 고려하라.

- from, of, valueOf, instance, create, getType …
- 이름을 가질 수 있다.
- 호출될 때마다 인스턴스를 새로 생성하지 않아도 된다.

```
...

public static final Boolean TRUE = new Boolean(true);

/**
 * The {@code Boolean} object corresponding to the primitive
 * value {@code false}.
 */
public static final Boolean FALSE = new Boolean(false);

...

@HotSpotIntrinsicCandidate
public static Boolean valueOf(boolean b) {
    return (b ? TRUE : FALSE);
}
```

- 반환 타입의 하위 타입 객체를 반환할 수 있는 능력이 있다.
  - 인터페이스 기반 프레임워크를 만드는 핵심 기술




List (java.util)
CheckedList in Collections (java.util)
AbstractList (java.util)
NamedDomainObjectList (org.gradle.api)
FastList (com.zaxxer.hikari.util)
PersistentList (org.hibernate.collection.internal)
PersistentList (org.hibernate.collection.internal)
COWSubList in CopyOnWriteArrayList (java.util.concurrent)
IdentityLinkedList (sun.awt.util)
ArrayList (java.util)
ExternalModuleDependencyBundle (org.gradle.api.artifacts)
AbstractImmutableList in ImmutableCollections (java.util)
StringList (com.sun.org.apache.xerces.internal.xs)
Vector (java.util)
MessageHistory (org.springframework.integration.history)
XSNamespaceItemList (com.sun.org.apache.xerces.internal.xs)
SynchronizedList in Collections (java.util)
LinkedList (java.util)
IdentityArrayList (sun.awt.util)
AutoPopulatingList (org.springframework.util)
AutoPopulatingList (org.springframework.util)
AutoPopulatingList (org.springframework.util)
AutoPopulatingList (org.springframework.util)
AutoPopulatingList (org.springframework.util)
AutoPopulatingList (org.springframework.util)
UnmodifiableList in Collections (java.util)
UnmodifiableRandomAccessList in Collections (java.util)

아이템 2. 생성자에 매개변수가 많다면 빌더패턴을 고려하라.

- https://www.baeldung.com/kotlin/builder-pattern

아이템 5. 자원을 직접 명시하지 말고 의존 객체 주입을 사용하라.

```
// (X)
public class SpellChecker {
  private final Lexicon dictionary = new ...
}

// (O) 유연함
public class SPellChecker {
  private final Lexicon dictionary;

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

아이템 11. equals를 재정의하려거든 hashcode도 재정의하라.

- 반대의 상황은 성립하지 않는다.
- equals가 두 객체를 같다고 판단 → 두 객체의 hashcode 값도 같은 값을 반환해야 한다.
- equals가 다르더라도 hashcode 값도 다른 값을 반환할 필요는 없다.

```
Map<PhoneNumber, String> m = new HashMap<>();
m.put(new PhoneNumber(1, 2, 3), "제니");

// 아래 코드의 결과값은?
System.out.println(m.get(new PhoneNumber(1, 2, 3)));

---

Map<PhoneNumber, String> m = new HashMap<>();
PhoneNumber p1 = new PhoneNumber(10, 20, 30);
PhoneNumber p2 = new PhoneNumber(10, 20, 30);
```

```
m.put(p1, "J");
m.put(p2, "J");

// 아래 코드의 결과는?
System.out.println(m.size());
```

▼ 정답

Hash를 사용한 컬렉션(HashMap, HashSet 등등)은 key를 결정할 때 hashCode()를 사용

```
public class Main {

    public static void main(String[] args) {
        Map<PhoneNumber, String> m = new HashMap<>();
        m.put(new PhoneNumber(1, 2, 3), "제니");

        System.out.println(m.get(new PhoneNumber(1, 2, 3)));

        PhoneNumber p1 = new PhoneNumber(10, 20, 30);
        PhoneNumber p2 = new PhoneNumber(10, 20, 30);

        m.put(p1, "J");
        m.put(p2, "J");

        // 1
        System.out.println(m.size());
    }

}

class PhoneNumber {
```

```
    ...

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        PhoneNumber that = (PhoneNumber) o;
        return num1 == that.num1 && num2 == that.num2 && num3 == that.num3;
    }

    @Override
    public int hashCode() {
        return Objects.hash(num1, num2, num3);
    }
}
```

아이템 12. toString을 항상 재정의하라

- 이상적으로는 객체가 가진 주요 정보를 모두 반환하는게 좋으나, 객체가 거대하거나(필드가 많을경우) 필수적인 요약정보를 담자.
- toString을 재정의한 클래스는 사용하기도 즐겁고 그 클래스를 사용한 시스템을 디버깅하기 쉽게 해준다.
- toString은 객체에 관한 명확하고 유용한 정보를 읽기 좋은 형태로 반환해야 한다.

아이템 15. 변경 가능성을 최소화하라

- 생성자는 불변식 설정이 모두 완료된, 초기화가 완벽히 끝난 상태의 객체를 생성해야 한다.
- `불변 클래스`: 인스턴스의 내부 값을 수정할 수 없는 클래스(ex. String)

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
```

```
class Main {
    public static void main(String[] args) {
        String s = new String("Hello");
        changeString(s);
```

```
        // 결과는? Hello or Hello World! or 에러
        System.out.println(s);
    }

    private static void changeString(String s) {
        s += " World!";
    }
}
```

- 클래스를 불변으로 만들기 위한 다섯가지 규칙
  1. 객체의 상태를 변경하는 메서드(setter 등)를 제공하지 않는다.
  2. 클래스를 확장할 수 없도록 한다.(final)
  3. 모든 필드를 final(const val ?)로 선언
  4. 모든 필드를 private으로 선언
  5. 자신 외에는 내부의 가변 컴포넌트에 접근할 수 없도록 한다.
- 불변 객체의 장점?
  - 스레드 안전하여 동기화할 필요가 없다.
  - 단순하다.
  - 실패 원자성을 제공한다.
    - 실패 원자성: ‘메서드에서 예외가 발생한 후에도 그 객체는 여전히 유효한 상태여야 한다.’
    - 불변 객체의 메서드는 내부 상태를 바꾸지 않으므로