11. null 대신 Optional

null 대신 Optional 클래스

> null의 문제점과 Optional을 통한 안전한 코드 작성법

---

0. 읽기 전...

갑자기 11장이 나왔지만, 우선 중요한 내용하다가 추후 다시 포스팅 할 예정이다.
우선 11장부터 학습하자.

1. null이란?

null은 참조 변수가 **아무 객체도 가리키고 있지 않음**을 의미한다. 즉, **주소 값이 없는** 상태이다.

null의 특징

- **원시 타입**(int, double, boolean)에는 null을 담을 수 없다
- **객체 참조 타입**(String, Integer, List 등)에만 사용 가능하다

---

2. 값이 없는 상황을 어떻게 처리할까?

다음과 같은 코드를 살펴보자.

```
public String getCarInsuranceName(Person person) {
    return person.getCar().getInsurance().getName();
}
```

위 코드는 다음과 같은 문제를 가지고 있다:

- 만약 **Person이 null**이라면?
- **getCar()가 null**을 반환한다면?
- **getInsurance()가 null**을 반환한다면?

모든 경우에 **NullPointerException(NPE)**이 발생한다.

2.1 보수적인 자세로 NPE 줄이기

예기치 않은 **NullPointerException(NPE)**을 피하려면 어떻게 해야 할까?

다양한 **null 확인 코드를 추가**하여 null 예외 문제를 피하는 방법이 있다.

```
if (person != null) {
    Car car = person.getCar();
    if (car != null) {
        Insurance insurance = car.getInsurance();
        if (insurance != null) {
            return insurance.getName();
        }
    }
}
return "Unknown";
```

변수를 참조할 때마다 **null을 확인**하며, 중간 과정에 하나라도 null이 있으면 **"Unknown"**을 반환한다.

각각의 조건문을 통해 모든 변수가 null인지 의심하게 된다. if 문이 추가되면서 **코드 들여쓰기 수준이 증가**하게 되었으며, 이 같은 반복 패턴을 **깊은 의심(deep doubting)**이라고 부른다.

2.2 null 때문에 발생하는 문제

에러의 근원

NullPointerException(NPE)이 발생한다.

코드를 어지럽힌다

중첩된 null 확인 코드가 추가되면서 **가독성이 떨어진다**.

아무 의미가 없다

null은 **아무 의미가 없으며** 값의 부재를 표현하기에 적절하지 않다.

자바 철학에 위배된다

자바는 개발자로부터 **포인터를 숨겼다**. 하지만 **null 포인터**가 예외이다.

형식 시스템에 구멍을 만든다

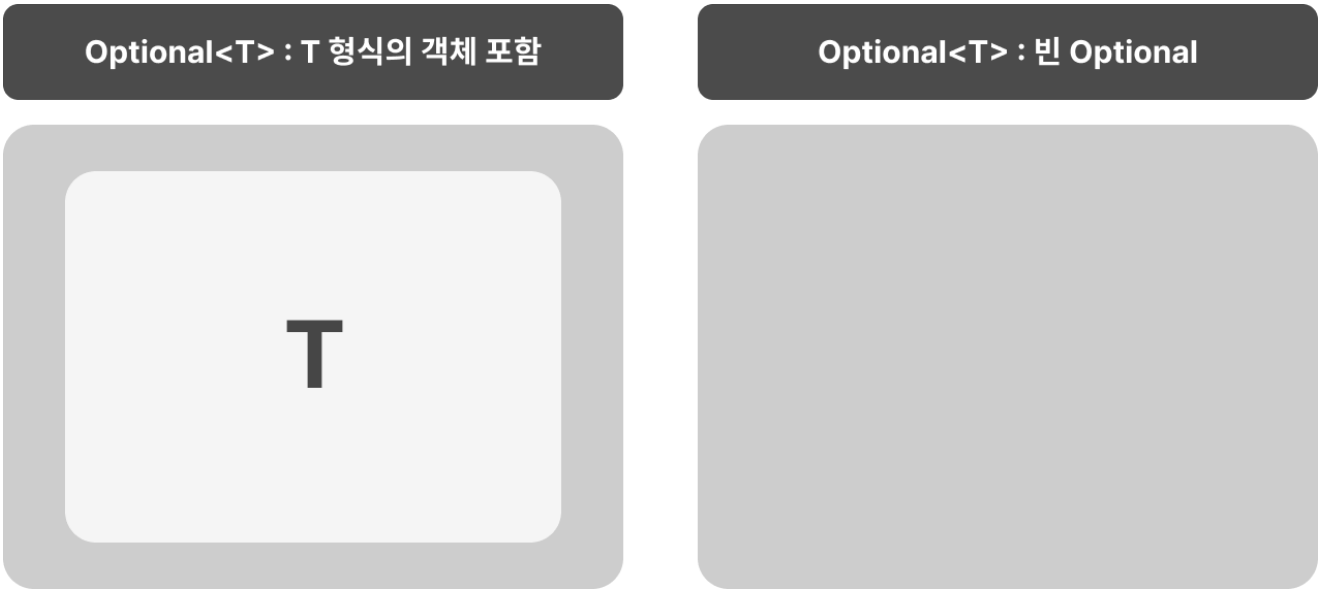
- null은 **무형식**이며 정보를 포함하고 있지 않다
- 모든 참조 타입에 null을 할당할 수 있다
- null이 시스템의 다른 부분으로 전파되면 **어떤 의미로 사용되었는지 알 수 없다**

---

3. Optional 클래스 소개

java.util.Optional는 **값이 있을 수도 있고, 없을 수도 있다**를 나타내는 **컨테이너 클래스**이다.

Wrapper 객체로, 내부에 실제 값(또는 null)을 한 칸 담고 있는 **상자**로 비유할 수 있다.



Optional의 동작 방식

- 값이 있으면 **Optional 클래스가 값을 감싼다**
- 값이 없으면 **Optional.empty()** 메서드로 Optional을 반환한다

Optional.empty()는 **싱글톤 인스턴스**를 반환한다. null을 참조하면 NPE가 발생하지만,
Optional.empty()는 Optional 객체이므로 활용이 가능하다.

Optional의 내부 구조

- Optional은 내부적으로 **value**라는 필드를 가진다
- **of()**는 null을 허용하지 않는다
- **ofNullable()**은 null일 경우 **빈 Optional**을 반환하며, null이 아닌 경우에는 **Optional로 감싼 값 객체**를 반환한다
- **empty()**는 **싱글톤 EMPTY 인스턴스**를 재사용한다

> Optional을 이용하면 값이 없는 상황이 데이터에 문제가 있는 것인지 아니면 알고리즘의 버그인지 **명확하게 구분**할 수 있다.

> 하지만 **모든 null 참조를 Optional로 대체하는 것은 바람직하지 않다**.

> Optional은 **더 이해하기 쉬운 API를 설계**하도록 돕는 것이다.

3.1 Optional의 본래 목적

"없을 수 있는 값"을 반환하거나 전달할 때, **타입 차원에서 명시적으로 표현**한다.

이는 **메서드 반환 값**에서 가장 큰 힘을 발휘한다.

예시:

```
Optional<User> findById(Long id);
```

위와 같이 반환 타입을 보는 것만으로도 "이 메서드는 User를 찾지 못할 수 있구나"라는 것을 알 수 있다.

3.2 모든 null을 Optional로 바꾸면 생기는 문제

1) 필드에 사용할 경우

```
class User {
    private Optional<String> name;
    ...
}
```

문제점:

- 객체 안의 필드가 Optional이면, 그 객체를 사용할 때마다 **map(), orElse()**로 감싸야 한다
- **Jackson, JPA** 같은 프레임워크와 **호환성 문제**가 발생한다

> 이럴 때는 null 대신 **빈 문자열 또는 기본 값 사용**을 추천한다.

2) 매개변수에 사용할 경우

```
void updateUser(Optional<User> userOpt);
```

문제점:

- 호출하는 쪽에서 매번 **.ofNullable()**을 해야 한다
- 매개변수는 null을 허용하고 **내부에서 처리**하는 것이 더 낫다

3) 좋은 사용법

메서드 반환 타입으로 사용하여 값이 없을 수 있음을 명확히 표현한다.

```
Optional<User> findByEmail(String email);
```

4) 나쁜 경우

- **필드**에 사용
- **매개변수**에 사용
- **컬렉션**에 사용 (컬렉션은 **빈 리스트 반환**이 더 직관적이다)

---

4. Optional 적용 패턴

4.1 Optional 객체 만들기

Optional을 사용하려면 먼저 Optional 객체를 만들어야 한다.

1) 빈 Optional 만들기

정적 팩토리 메서드 **Optional.empty()**로 빈 Optional 객체를 얻을 수 있다.

```
Optional<Car> optCar = Optional.empty();
```

2) null이 아닌 값으로 Optional 만들기

정적 팩토리 메서드 **Optional.of()**로 null이 아닌 값을 포함하는 Optional을 만들 수 있다.

```
Optional<Car> optCar = Optional.of(car);
```

주의: car가 null이면 **NullPointerException**이 발생한다.

3) null 값으로 Optional 만들기

Optional.ofNullable()로 null 값을 저장할 수 있는 Optional 객체를 만들 수 있다.

```
Optional<Car> optCar = Optional.ofNullable(car);
// car가 null이면 빈 Optional 객체를 반환한다
```

4.2 map으로 Optional의 값을 추출하고 변환

보통 객체의 정보를 추출할 때 Optional을 사용할 때가 많다.

```
String name = null;
if (insurance != null) {
    name = insurance.getName();
}
```

이런 유형의 패턴에 사용할 수 있게 **map 메서드**를 지원한다.

```
Optional<Insurance> optInsurance = Optional.ofNullable(insurance);
Optional<String> name = optInsurance.map(Insurance::getName);
```

스트림의 map 메서드와 **개념적으로 비슷**하다. 여기서 Optional 객체를 **최대 요소의 개수가 1개 이하인 데이터 컬렉션**으로 생각할 수도 있다.

- Optional이 **값을 포함**하면 map의 인수로 제공된 함수가 값을 변환한다
- Optional이 **비어있으면** 아무 일도 일어나지 않는다 (map() 메서드가 실행되지 않는다)

4.3 flatMap으로 Optional 객체 연결

다음과 같은 도메인 모델을 생각해 보자.

```
public class Person {
    private Optional<Car> car;
    public Optional<Car> getCar() { return car; }
}

public class Car {
    private Optional<Insurance> insurance;
    public Optional<Insurance> getInsurance() { return insurance; }
}

public class Insurance {
    private String name;
    public String getName() { return name; }
}
```

다음과 같이 체이닝을 시도해 보자.

```
Optional<Person> optPerson = Optional.of(person);
Optional<String> name =
    optPerson.map(Person::getCar)
             .map(Car::getInsurance)
             .map(Insurance::getName);
```

> **위 코드는 컴파일되지 않는다.**

이유:

- 첫 번째 map 메서드에서 **getCar()**는 **Optional** 형식의 객체를 반환한다
- 하지만 **map의 결과**는 **Optional<Optional>** 형식의 객체이다
- 두 번째 map 메서드에서 **getInsurance()**는 또 다른 Optional 객체를 반환하므로 **getInsurance()를 지원하지 않는다**

여기서 **flatMap 메서드**를 사용하여 이러한 문제를 해결할 수 있다.

flatMap은 인수로 받은 함수를 적용해서 생성된 각각의 스트림에서 **콘텐츠만 남긴다**. 즉, 함수를 적용해 생성된 모든 스트림이 **하나의 스트림으로 병합되어 평준화**된다.

```
String name = optPerson.flatMap(Person::getCar)      // 1
                       .flatMap(Car::getInsurance)    // 2
                       .map(Insurance::getName)       // 3
                       .orElse("Unknown");            // 4
```

설명:

1. **Person::getCar**가 **Optional**를 반환한다. map을 사용하면 다시 감싸서 중첩이 된다. 이를 풀어주는 것이 **flatMap()**이다. 1번의 반환 값은 **Optional**이다
2. **Optional**를 반환한다
3. **Optional**을 반환한다
4. 결과 Optional이 비어있으면 **기본 값을 사용**한다

이렇게 되면 null을 확인하느라 조건 분기문을 추가해서 코드를 복잡하게 만들지 않으면서 **쉽게 이해**할 수 있다.

flatMap과 map의 차이

> **map()**은 내부 값을 꺼내서 함수를 적용한 후 **다시 상자에 넣는다**

> **Person::getCar**가 반환하는 것이 이미 **Optional**라는 상자이다

> 그럼 **map()**이 그 상자를 또 감싸서 **Optional<Optional>**가 되어버린다

> **flatMap()**은 이미 Optional로 감싸진 결과를 **평평하게(flat)** 만들어준다

4.3.1 도메인 모델에 Optional을 사용했을 때 직렬화가 불가능하다고?

자바 언어 아키텍트인 **브라이언 고츠(Brian Goetz)**는 Optional의 용도가 **선택형 반환 값을 지원하는 것**이라고 명확히 말했다.

Optional 클래스는 **필드 형식으로 사용하지 않을 것**으로 가정하여 **Serializable 인터페이스를 구현하지 않는다**.

따라서 도메인 모델에 Optional을 사용한다면 **직렬화 모델을 사용하는 도구나 프레임워크에 문제가 될 수 있다**.

4.4 Optional 스트림 조작

자바 9에서는 Optional을 포함하는 스트림을 쉽게 처리할 수 있게 Optional에 **stream() 메서드**를 추가했다.

```
public Set<String> getCarInsuranceNames(List<Person> persons) {
    return persons.stream()
        .map(Person::getCar)                                    //
Stream<Optional<Car>>로 변환
        .map(optCar -> optCar.flatMap(Car::getInsurance))       //
Stream<Optional<Insurance>>로 변환
        .map(optIns -> optIns.map(Insurance::getName))          //
Stream<Optional<String>>로 변환
        .flatMap(Optional::stream)                              //
Stream<Optional<String>>을 Stream<String>으로 변환
```

```
        .collect(toSet());                                    // 중복 제거
}
```

설명:

- 첫 번째 map 변환 수행 후 **Stream<Optional>**를 얻는다
- 두 개의 map 연산을 이용해 **Optional → Optional**로 변환한다
- 마지막으로 **Optional**으로 변환한다

하지만 중간 결과가 비어있을 수 있고, 마지막 결과를 얻으려면 **빈 Optional을 제거하고 값을 언랩**해야 한다.

전통적인 방식

```
Stream<Optional<String>> stream = ...
Set<String> result = stream.filter(Optional::isPresent)
                           .map(Optional::get)
                           .collect(toSet());
```

stream() 메서드 사용

Optional 클래스의 **stream() 메서드**를 이용하면 **한 번의 연산**으로 같은 결과를 얻을 수 있다.

4.5 디폴트 액션과 Optional 언랩

4.5.1 디폴트 액션

orElse(T other)

- 값이 있으면 그대로 반환하고, 없으면 **other**를 반환한다
- **other는 항상 평가**되므로 무거운 연산은 피해야 한다

```
Optional<String> opt = Optional.empty();
String value = opt.orElse("기본 값");
```

orElseGet(Supplier<? extends T> supplier)

- 값이 있으면 그대로 반환하고, 없으면 **Supplier의 실행 결과**를 반환한다
- **orElse와 달리 필요할 때만 실행**한다
- 무거운 연산이나 **Lazy 로딩**에 적합하다

```
Optional<String> opt = Optional.empty();
String value = opt.orElseGet(() -> expensiveOperation());
```

orElseThrow()

- 값이 있으면 반환하고, 없으면 **NoSuchElementException**을 발생시킨다

```
Optional<String> opt = Optional.empty();
String value = opt.orElseThrow(); // 예외 발생
```

orElseThrow(Supplier<? extends X> exceptionSupplier)

- 값이 없으면 예외를 던지는데, 이때 **예외를 커스터마이징** 가능하다

```
Optional<String> opt = Optional.empty();
String value = opt.orElseThrow(
    () -> new IllegalArgumentException("값이 없음")
);
```

4.5.2 언랩

get()

- Optional 내부 값을 반환한다
- 값이 있으면 해당 값을 반환하고, 없으면 **NoSuchElementException**을 발생시킨다
- **반드시 값이 있다고 가정하는 상황에서만 사용**해야 한다

```
Optional<String> opt = Optional.of("TEST");
String value = opt.get();
```

isPresent()

- 값이 있으면 **true**, 없으면 **false**이다
- 전통적인 null 체크와 비슷하다
- 최근에는 **ifPresent()**를 더 권장한다

isEmpty()

- 값이 없으면 **true**이다

ifPresent(Consumer<? super T> action)

- 값이 있으면 **Consumer**를 실행한다

```
opt.ifPresent(v -> System.out.println(v));
```

ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)

- **첫 번째 매개변수**: 값이 있으면 실행한다

- **두 번째 매개변수**: 값이 없으면 실행한다

```
opt.ifPresentOrElse(
    v -> System.out.println(v),
    () -> System.out.println("empty")
);
```

4.6 필터로 특정 값 거르기

Optional 객체에 **filter 메서드**를 이용해서 코드를 재구현할 수 있다.

```
Optional<Insurance> opt = ...;
opt.filter(insurance ->
    "CambridgeInsurance".equals(insurance.getName())
).ifPresent(x -> System.out.println("OK"));
```

동작 방식:

- Optional 객체가 값을 가지며 **프레디케이트와 일치**하면 filter 메서드는 그 값을 반환한다
- 그렇지 않으면 **빈 Optional 객체**를 반환한다

Optional은 **최대 1개의 요소를 포함할 수 있는 스트림**과 같다.

4.6.1 Optional 클래스의 추가적인 메서드

filter()

- 값이 존재하며 **프레디케이트와 일치**하면 값을 포함하는 Optional을 반환한다
- 값이 없거나 프레디케이트와 일치하지 않으면 **빈 Optional**을 반환한다

stream()

- 값이 존재하면 **존재하는 값만 포함하는 스트림**을 반환한다
- 값이 없으면 **빈 스트림**을 반환한다

or()

- 값이 존재하면 **같은 Optional**을 반환한다
- 값이 없으면 **Supplier에서 만든 Optional**을 반환한다

이외에도 추가적인 메서드들이 있는데, 구글에서 검색하거나 공식 문서를 참고하자.

---

5. 기본형 Optional

Optional에도 기본형 특화가 있다. **OptionalInt, OptionalLong, OptionalDouble** 등이 있다.

하지만 **기본형 Optional은 별로 추천하지 않는다**.

추천하지 않는 이유

1) 성능 이득이 거의 없다

- 기본형 특화는 내부적으로 단순히 **Optional** 같은 **오토박싱을 피하는 정도**이다
- 하지만 Optional은 **최대 요소 수가 1개**이므로 **성능상 거의 이득을 볼 수 없다**

2) 기능 제약

- 기본형 특화는 **map, flatMap, filter 등을 지원하지 않는다**
- 이를 통해 생성한 결과는 **다른 일반 Optional과 혼용하지 못한다**

---

6. Optional 사용 시 주의사항

✅ 좋은 사용법

1. **메서드 반환 타입**으로 사용
2. 값이 없을 수 있음을 **타입으로 명시**
3. **orElse(), orElseGet(), orElseThrow()** 등으로 안전하게 처리

❌ 나쁜 사용법

1. **필드**에 사용하지 않기
2. **매개변수**에 사용하지 않기
3. **컬렉션**의 요소로 사용하지 않기
4. **기본형 Optional** 사용하지 않기

---

7. 정리

Optional은 null 참조를 안전하게 다루기 위한 도구이다.

- **값이 없을 수 있음**을 타입 시스템으로 명시한다
- **map, flatMap, filter** 등의 메서드로 함수형 프로그래밍 스타일을 지원한다
- **null 체크 지옥**에서 벗어나 **더 읽기 쉬운 코드**를 작성할 수 있다

하지만 **모든 null을 Optional로 바꾸는 것은 아니다**. Optional의 본래 목적인 **메서드 반환 타입**에 주로 사용하고, 필드나 매개변수에는 사용을 자제해야 한다.

> **Optional은 "값이 없을 수 있다"는 사실을 API로 명확하게 표현하는 도구이다.**