Optional

소유자	종수 김
태그	

Optional이 필요한 이유

null

- 자바에서 null은 '값이 없음'을 표현하는 가장 기본적인 방법.
- null을 잘못 사용하거나, null 참조에 대해 메서드를 호출하면 NPE가 발생하여 프로그램이 예기치 않게 종료.

가독성 저하

- null을 반환하거나, 사용하게 되면 if(obj ≠ null)과 같은 불 필요 코드 생성

의도가 드러나지 않음

- null을 반환할 수 있다는 사실을 명확히 알기가 어려움.
- 호출하는 입장에서 '반드시 값이 존재한다'라고 가정했다가 아닐 수도 있음

Optional

- 위와 같은 문제를 해결하고자 자바 8부터 도입된 클래스
- 값이 있을수도 있고, 없을 수도 있음. 을 명시적으로 표현.
- Optional을 사용하면 '빈 값'을 표현할 때, 더 이상 null 자체를 넘겨주지 않고 Optional.empty()처럼 의도가 드러나는 객체 사용 가능.
- 이를 통해 체크 로직을 간결하게 만들고 NPE가 발생할 수 있는 부분을 더 쉽게 파악 가능.

```
package optional;

import java.util.HashMap;
import java.util.Map;

public class OptionalStartMain1 {
    private static final Map<Long, String> map = new HashMap<>();
    static {
```

```
        map.put(1L, "Kim");
        map.put(2L, "Seo");
    }
    public static void main(String[] args) {
        findAndPrint(1L); // 값이 있음
        findAndPrint(3L); // 값이 없음
    }
    // 이름이 있으면 이름을 대문자로출력, 없으면 "UNKNOWN"
    static void findAndPrint(Long id) {
        String name = findNameById(id);
        // 1. NPE 발생
        // System.out.println("name = " + name.toUpperCase());

        // 2. if 문을 활용한 null 체크
        if (name != null) {
            System.out.println(id + " : " + name.toUpperCase());
        }else {
            System.out.println("UNKNOWN");
        }
    }


    static String findNameById(Long id) {
        return map.get(id);
    }

}
```

- map.get(id)의 결과가 존재하지 않으면, null을 반환.
- 따라서 null 체크 로직이 항상 들어가야 함.
    - 이를 빠트리면 NPE가 발생
- 반환 타입이 String이므로 문자열이 반환될 것 같으나, 실제로는 null이 반환될 수 있음.

Optional 사용 예

```
package optional;

import java.util.HashMap;
```

```
import java.util.Map;
import java.util.Optional;

public class OptionalStartMain2 {
    private static final Map<Long, String> map = new HashMap<>();
    static {
        map.put(1L, "Kim");
        map.put(2L, "Seo");
    }
    public static void main(String[] args) {
        findAndPrint(1L); // 값이 있음
        findAndPrint(3L); // 값이 없음
    }
    // 이름이 있으면 이름을 대문자로출력, 없으면 "UNKNOWN"
    static void findAndPrint(Long id) {
        Optional<String> optName = findNameById(id);
        String name = optName.orElse("UNKNOWN");

        System.out.println(name);
    }


    static Optional<String> findNameById(Long id) {
        String findName = map.get(id);
        Optional<String> optionalName = Optional.ofNullable(findName);
        return optionalName;
    }

}
```

- Optional<String>을 반환함으로 써 사용자가 반환 결과가 있을수도 있고, 없을수도 있겠음을 명시적으로 인지.
- orElse(“”)를 통해, 값이 없는 경우 null이 아닌 대체 문자열을 지정.
- Optional.ofNullable()을 통해 null이 될 수도 있는 값을 Optional로 감싼다.

Optional 소개

자바 Optional 클래스 코드

```
package java.util;

public final class Optional<T> {
    private final T value;
    ...
}
```

정의

- `java.util.Optional<T>`는 "존재할 수도 있고 존재하지 않을 수도 있는" 값을 감싸는 일종의 컨테이너 클래스이다.
- 내부적으로 `null`을 직접 다루는 대신, `Optional` 객체에 감싸서 `Optional.empty()` 또는 `Optional.of(value)` 형태로 다룬다.

등장 배경

- "값이 없을 수 있다"는 상황을 프로그래머가 명시적으로 처리하도록 유도하고, 런타임 `NullPointerException`을 사전에 예방하기 위해 도입되었다.
- 코드를 보는 사람이나 협업하는 팀원 모두가, 해당 메서드의 반환값이 비어있을 수도 있음을 알 수 있게 되어 오류를 줄일 수 있다.

참고

- `Optional`은 "값이 없을 수도 있다"는 상황을 반환할 때 주로 사용된다.
- "항상 값이 있어야 하는 상황"에서는 `Optional`을 사용할 필요 없이 그냥 해당 타입을 바로 사용하거나 예외를 던지는 방식이 더 좋을 수 있다.

Optional의 생성과 값 획득

Optional 생성

1. Optional.of
    a. 내부 값이 확실히 null이 아닐 때
    b. null 전달 시 NPE 발생
2. Optional.ofNullable
    a. 값이 Null일 수도 있고, 아닐 수도 있을 때
    b. null 이면 Optional.empty()반환
3. Optional.empty

a. 명시적으로 값이 없음을 표현

Optional 값 획득

▼ 예제

Optional의 값을 확인하거나, 획득하는 메서드

1. `isPresent()`, `isEmpty()`
   - 값이 있으면 `true`
   - 값이 없으면 `false`를 반환. 간단 확인용.
   - `isEmpty()` : 자바 11 이상에서 사용 가능, 값이 비어있으면 `true`, 값이 있으면 `false`를 반환
2. `get()`
   - 값이 있는 경우 그 값을 반환
   - 값이 없으면 `NoSuchElementException` 발생.
   - 직접 사용 시 주의해야 하며, 가급적이면 `orElse`, `orElseXxx` 계열 메서드를 사용하는 것이 안전.
3. `orElse(T other)`
   - 값이 있으면 그 값을 반환
   - 값이 없으면 `other`를 반환.
4. `orElseGet(Supplier<? extends T> supplier)`
   - 값이 있으면 그 값을 반환
   - 값이 없으면 `supplier` 호출하여 생성된 값을 반환.
5. `orElseThrow(...)`
   - 값이 있으면 그 값을 반환
   - 값이 없으면 지정한 예외를 던짐.
6. `or(Supplier<? extends Optional<? extends T>> supplier)`
   - 값이 있으면 해당 값의 `Optional`을 그대로 반환
   - 값이 없으면 `supplier`가 제공하는 다른 `Optional` 반환
   - 값 대신 `Optional`을 반환한다는 특징

```
package optional;

import java.util.Optional;

public class OptionalRetrievalMain {
    public static void main(String[] args) {
        // 예제 : 문자열 "Java"가 있는 Optional과 비어있는 Optional
        Optional<String> optValue = Optional.of("Hello");
        Optional<String> optEmpty = Optional.empty();
```

```
        // isPresent : 값이 있으면 true
        System.out.println("optValue.isPresent() = " + optValue.isPresent());
        System.out.println("optEmpty.isPresent() = " + optEmpty.isPresent());
        System.out.println("optEmpty.isEmpty() = " + optEmpty.isEmpty());

        // get : 직접 내부 값을 꺼냄, 없으면 NoSuchElementException
        String getValue = optValue.get();
        System.out.println("getValue = " + getValue);
//        String emptyValue = optEmpty.get(); // 예외 발생
//        System.out.println(emptyValue);

        // orElse : 있으면 그 값, 없으면 대체 값
        String value1 = optValue.orElse("기본값");
        String empty1 = optEmpty.orElse("기본값");
        System.out.println("value1 = " + value1);
        System.out.println("empty1 = " + empty1);

        // orElseGet : 값이 없을 때만 람다(Supplier)가 실행되어 기본 값 생성
        String value2 = optValue.orElseGet(() -> {
            System.out.println("람다 호출");
            return "New Value";
        });

        String empty2 = optEmpty.orElseGet(() -> {
            System.out.println("람다 호출");
            return "New Value";
        });
        System.out.println("value2 = " + value2);
        System.out.println("empty2 = " + empty2);

        // orElseThrow : 값이 있으면 반환, 없으면 예외
        String value3 = optValue.orElseThrow(() -> new RuntimeException("값
        System.out.println("value3 = " + value3);
//        String empty3 = optEmpty.orElseThrow(RuntimeException::new); //
//        System.out.println("empty3 = " + empty3);

        // or : Optional을 반환
        Optional<String> result1 = optValue.or(() -> Optional.of("Hello"));
```

```
        Optional<String> result2 = optEmpty.or(() -> Optional.of("Fallback"));
        System.out.println("result1 = " + result1);
        System.out.println("result2 = " + result2);
    }
}
```

- get() 메서드는 Optional 사용 시 가능하면 피하는게 좋음.
    - 값이 없는 상태에서 get()을 호출 시 예외가 터지기 때문에 isPresent()와 같은 사전 체크가 필요.
    - get() 보다는 orElse, orElseGet, orElseThrow와 같은 메서드를 통하는게 더 안전

Optional 값 처리

- 값이 존재할 때와, 존재하지 않을 때를 처리하기 위한 다양한 메서드 제공.
- null 체크 로직 없이도 안전하고 간결하게 값을 다룰 수 있음.

▼ 예제

Optional 값 처리 메서드

- `ifPresent(Consumer<? super T> action)`
  - 값이 존재하면 action 실행
  - 값이 없으면 아무것도 안 함
- `ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)`
  - 값이 존재하면 action 실행
  - 값이 없으면 emptyAction 실행
- `map(Function<? super T, ? extends U> mapper)`
  - 값이 있으면 `mapper`를 적용한 결과(`Optional<U>`) 반환
  - 값이 없으면 `Optional.empty()` 반환
- `flatMap(Function<? super T, ? extends Optional<? extends U>> mapper)`
  - map과 유사하지만, `Optional`을 반환할 때 중첩되지 않고 평탄화(flat)해서 반환

- `filter(Predicate<? super T> predicate)`
  - 값이 있고 조건을 만족하면 그대로 반환,
  - 조건 불만족이거나 비어있으면 `Optional.empty()` 반환
- `stream()`
  - 값이 있으면 단일 요소를 담은 `Stream<T>` 반환
  - 값이 없으면 빈 스트림 반환

```
package optional;

import java.util.Optional;

public class OptionalProcessingMain {
    public static void main(String[] args) {
        Optional<String> optValue = Optional.of("Hello");
        Optional<String> optEmpty = Optional.empty();

        // 값이 존재하면 Consumer 실행, 없으면 아무것도 실행하지 않음.
        optValue.ifPresent(v -> System.out.println("optValue 값 = " + v));
        optEmpty.ifPresent(v -> System.out.println("optEmpty 값 = " + v));

        // 값이 있으면 consumer 실행, 없으면 runnable 실행
        // 값이 있으면 인자가 있으므로 Consumer, 값이 없으면 인자가 없으므로 Run
```

```
        optValue.ifPresentOrElse(
                v -> System.out.println("optValue 값 = " + v)
                , () -> System.out.println("optValue 비어있음.")
        );
        optEmpty.ifPresentOrElse(
                v -> System.out.println("optEmpty 값 = " + v)
                , () -> System.out.println("optEmpty 비어있음.")
        );

        // 값이 있으면 Function 적용 후 Optional로 변환, 없으면 empty
        Optional<Integer> lengthOpt1 = optValue.map(String::length);
        System.out.println("lengthOpt1 = " + lengthOpt1);
        Optional<Integer> lengthOpt2 = optEmpty.map(String::length);
        System.out.println("lengthOpt2 = " + lengthOpt2);

        // map()과 유사하나, Optional을 반환하는 경우 중첩을 제거
        Optional<Optional<String>> nestedOpt = optValue.map(Optional::of);
        System.out.println("nestedOpt = " + nestedOpt);
        Optional<String> flattenedOpt = optValue.flatMap(Optional::of);
        System.out.println("flattenedOpt = " + flattenedOpt);

        // 값이 있고 조건을 만족하면 그 값 그대로, 불만족시 empty
        Optional<String> filtered1 = optValue.filter(s -> s.length() == 5);
        System.out.println("filtered1 = " + filtered1);
        Optional<String> filtered2 = optValue.filter(s -> s.length() == 6);
        System.out.println("filtered2 = " + filtered2);

        // 값이 있으면 단일 요소 스트림, 없으면 빈 스트림
        optValue.stream()
                .forEach(s -> System.out.println("optValue.stream() -> " + s)); // 
        optEmpty.stream()
                .forEach(s -> System.out.println("optEmpty.stream() -> " + s)); //
    }
}
```

- 값이 존재할 때와, 존재하지 않을 때의 로직을 명확하고 간결하게 구현 가능.
- null 체크로 인한 복잡한 코드와 예외처리를 줄이고, 더 읽기 쉽고 안전한 코드를 작성 가능.

- orElse()와 같이, null일 때 대체 값.
- ifPresentOrElse()와 같이 null일 때 대체 로직 과 같은 처리가 가능.

즉시 평가와 지연 평가 - 1

orElse() / orElseGet()은 즉시 평가와 지연 평가로 다름.

즉시 평가(eager evaluation)

- 값(혹은 객체)을 바로 생성하거나 계산해버리는 것.

- 지연 평가(lazy evaluation)
  - - 값이 실제로 필요할 때 (즉, 사용될 때)까지 계산을 미루는 것.

평가 - 계산.

자바 언어의 연산 순서와 즉시 평가

자바는 연산식을 보면 기본적으로 즉시 평가한다. 이 말을 이해하기 위해

debug(10 + 20) 연산부터 알아보자.

```
// 자바 언어의 연산자 우선순위상 메서드를 호출하기 전에 괄호 안의 내용이 먼저 계산된다.
logger.debug(10 + 20); // 1. 여기서는 10 + 20이 즉시 평가된다.
logger.debug(30);      // 2. 10 + 20 연산의 평가 결과는 30이 된다.
debug(30)              // 3. 메서드를 호출한다. 이때 계산된 30의 값이 인자로 전달된다.
```

자바는 `10 + 20`이라는 연산을 처리할 순서가 되면 그때 바로 즉시 평가(계산) 한다.
우리에게는 너무 자연스러운 방식이기 때문에 아무런 문제가 될 것이 없어 보인다.
그런데 이런 방식이 때로는 문제가 되는 경우가 있다.

debug(100 + 200) 연산을 통해 어떤 문제가 있는지 알아보자.

```
System.out.println("=== 디버그 모드 끄기 ===");
logger.setDebug(false);
logger.debug(100 + 200);
```

이 연산은 debug 모드가 꺼져있기 때문에 출력되지 않는다. 따라서 `100 + 200` 연산은 어디에도 사용되지 않는다.
하지만 이 연산은 계산된 후에 버려진다. 다음 코드를 보자.

```
```

```
// 자바 언어의 연산자 우선순위상 메서드를 호출하기 전에 괄호 안의 내용이 먼저 계산된다.
logger.debug(100 + 200); // 1. 여기서는 100 + 200이 즉시 평가된다.
logger.debug(300);       // 2. 100 + 200 연산의 평가 결과는 300이 된다.
debug(300)               // 3. 메서드를 호출한다. 이때 계산된 300의 값이 인자로 전달된다.
```

```
public void debug(Object message = 300) { // 4. message에 계산된 300이 할당된다.
    if (isDebug) { // 5. debug 모드가 꺼져있으므로 false이다.
        System.out.println("[DEBUG] " + message); // 6. 실행되지 않는다.
    }
}
```

이 연산의 결과 `300`은 debug 모드가 꺼져있기 때문에 출력되지 않는다. 따라서 앞서 계산한 `100 + 200` 연산은 어디에도 사용되지 않는다. 결과적으로 연산은 계산된 후에 버려진다.

결과적으로 `100 + 200` 연산은 미래에 전혀 사용하지 않을 값을 계산해서 아까운 CPU 전기만 낭비한 것이다.
그런데 정말 사용하지도 않을 `100 + 200` 연산을 처리한 것일까? 눈으로 확인할 수 없으니 믿을 수가 없다!

즉시 평가와 지연 평가 - 2

```
package optional.logger;

public class LogMain2 {
    public static void main(String[] args) {
        Logger logger = new Logger();
        logger.setDebug(true);
        logger.debug(value100() + value200());

        logger.setDebug(false);
        logger.debug(value100() + value200());

    }
    static int value100() {
        System.out.println("value 100 호출");
        return 100;
    }

    static int value200() {
        System.out.println("value 200 호출");
        return 200;
    }
}
```

debug모드를 끈 후에도 value100(), value200()이 실행됨 .

그렇다면 debug 모드가 켜져있을 때는 해당 연산을 처리하고, debug 모드가 꺼져있을 때는 해당 연산을 처리하지 않으려면 어떻게 해야 할까?
가장 간단한 방법은 디버그 모드를 출력할 때 마다 매번 if 문을 사용해서 체크하는 방법이 있다.

기존 코드

```
logger.debug(value100() + value200());
```

if문으로 debug 메서드 실행 여부를 체크하는 코드

```
if (logger.isDebug()) {
    logger.debug(value100() + value200());
}
```

확인을 위해 코드 마지막에 다음 코드를 추가해보자.

디버그 모드 체크시 추가 코드

```
//코드 마지막에 추가
System.out.println("=== 디버그 모드 체크 ===");
if (logger.isDebug()) {
    logger.debug(value100() + value200());
}
```

- 코드가 지저분해지나, 필요없는 연산을 계산하지 않아도 됨.
  - 체크를 할 때 마다 if문이 들어가야 함.
- 미래에 사용하지 않을 연산이 미리 수행되지 않도록 하는 방법 ?
  - 코드의 깔끔함 + 필요없는 연산은 수행하지 않는 장점을 가져가는 방법
- 연산을 정의하는 시점과, 연산을 실행하는 시점을 분리해야함.
  - 연산의 실행을 최대한 지연해서 평가해야 함.

즉시 평가와 지연 평가 - 3

자바에서 연산을 정의하는 시점과, 연산을 실행하는 시점을 분리하는 방법은 여러가지가 있음.

1. 익명 클래스를 만들고, 메서드를 나중에 호출

2. 람다를 만들고, 해당 람다를 나중에 호출.

▼ 예제

```
package optional.logger;

import java.util.function.Supplier;

public class Logger {
    private boolean isDebug = false;

    public boolean isDebug() {
        return isDebug;
    }

    public void setDebug(boolean debug) {
        isDebug = debug;
    }

    // DEBUG로 설정한 경우만 출력 - 데이터를 바등ㅁ
    public void debug(Object message) {
        if (isDebug) {
            System.out.println("[DEBUG] " + message);
        }
    }

    // 추가
    // Debug로 설정한 경우만 출력 - 람다를 받아서 실행
    public void debug(Supplier<?> supplier) {
        if (isDebug) {
            System.out.println("[DEBUG] " + supplier.get());
        }
    }
}



package optional.logger;

public class LogMain3 {
```

```
    public static void main(String[] args) {
        Logger logger = new Logger();
        logger.setDebug(true);
        logger.debug(() -> value100() + value200());

        logger.setDebug(false);
        logger.debug(() -> value100() + value200());

    }
    static int value100() {
        System.out.println("value 100 호출");
        return 100;
    }

    static int value200() {
        System.out.println("value 200 호출");
        return 200;
    }
}
```

디버그 모드가 켜져있을 때

```
logger.debug(() -> value100() + value200()) // 1. 람다를 생성한다. 이때 람다가 실행되지
는 않는다.
logger.debug(() -> value100() + value200()) // 2. debug()를 호출하면서 인자로 람다를
전달한다.
```

```
// 3. supplier에 람다가 전달된다. (람다는 아직 실행되지 않았다.)
public void debug(Supplier<?> supplier = () -> value100() + value200()) {
    if (isDebug) { // 4. 디버그 모드이므로 if 문이 수행된다.
        // 5. supplier.get()을 실행하는 시점에 람다에 있는 value100() + value200()이 평
가(계산)된다.
        // 6. 평가 결과인 300을 반환하고 출력한다.
        System.out.println("[DEBUG] " + supplier.get());
    }
}
```

디버그 모드가 꺼져있을 때

```
logger.debug(() -> value100() + value200()) // 1. 람다를 생성한다. 이때 람다가 실행되지
는 않는다.
logger.debug(() -> value100() + value200()) // 2. debug()를 호출하면서 인자로 람다를
전달한다.
```

```
// 3. supplier에 람다가 전달된다. (람다는 아직 실행되지 않았다.)
public void debug(Supplier<?> supplier = () -> value100() + value200()) {
    if (isDebug) { // 4. 디버그 모드가 아니므로 if 문이 수행되지 않는다.
```

```
        // 5. 다음 코드는 수행되지 않고, 람다도 실행되지 않는다.
        System.out.println("[DEBUG] " + supplier.get());
    }
}
```

정리

람다를 사용해서 **연산을 정의하는 시점과 실행(평가)하는 시점을 분리**했다. 따라서 값이 실제로 필요할 때 까지 계산을 미룰 수 있었다. 람다를 활용한 지연 평가 덕분에 꼭 필요한 계산만 처리할 수 있었다.

- **즉시 평가(eager evaluation)**:
  - 값(혹은 객체)을 바로 생성하거나 계산해 버리는 것
- **지연 평가(lazy evaluation)**:
  - 값이 실제로 필요할 때(즉, 사용될 때)까지 계산을 미루는 것

orElse() vs orElseGet()

orElse()는 보통 데이터를 받아서 인자가 즉시 평가되고,

orElseGet()은 람다를 받아서 인자가 지연 평가됨.

▼ 예제

```
package optional;

import java.util.Optional;
import java.util.Random;

public class OrElseGetMain {
    public static void main(String[] args) throws Exception{
        Optional<Integer> optValue = Optional.of(100);
        Optional<Integer> optEmpty = Optional.empty();

        System.out.println("단순 계산");
        Integer i1 = optValue.orElse(10 + 20); // 10 + 20 계산 후 버림
        Integer i2 = optEmpty.orElse(10 + 20);// 10 + 20 계산 후 사용
        System.out.println("i1 = " + i1);
        System.out.println("i2 = " + i2);

        // 값이 있으면 그 값, 없으면 대체 값
        System.out.println("=== orElse ===");
        System.out.println("값이 있는 경우");
        Integer value1 = optValue.orElse(createData());
        System.out.println("value1 = " + value1);

        System.out.println("값이 없는 경우");
        Integer empty1 = optEmpty.orElse(createData());
        System.out.println("empty1 = " + empty1);

        // 값이 있으면 그 값, 없으면 대체 값
        System.out.println("=== orElseGet ===");
        System.out.println("값이 있는 경우");
        Integer value2 = optValue.orElseGet(() -> createData());
        System.out.println("value2 = " + value2);
```

```
        System.out.println("값이 없는 경우");
        Integer empty2 = optEmpty.orElseGet(() -> createData());
        System.out.println("empty2 = " + empty2);
    }
    public static int createData() {
        System.out.println("데이터 생성");
        try {
            Thread.sleep(3000);
        }catch (Exception e){
            e.printStackTrace();
        }
        int createValue = new Random().nextInt(100);
        System.out.println("생성 된 값 " + createValue);
        return createValue;
    }
}
```

- orElse(createData())
  - Optional에 값이 있어도, createData()가 즉시 호출됨. 호출된 값은 버려짐.
  - 자바 연산 순서상 createData()를 호출해야, orElse()에 인자로 전달할 수 있음.
- orElseGet(() → createData())
  - Optional에 값이 있으면, createData()가 호출되지 않음.
  - orElseGet()에 람다를 전달한다, 해당 람다는 이후에 orElseGet() 안에서 실행될 수 있음.

두 메서드의 차이

- `orElse(T other)`는 "빈 값이면 `other`를 반환"하는데, `other`를 "항상" 미리 계산한다.
  - 따라서 `other`를 생성하는 비용이 큰 경우, 실제로 값이 있을 때도 쓸데없이 생성 로직이 실행될 수 있다.
  - `orElse()`에 넘기는 표현식은 **호출 즉시 평가**하므로 **즉시 평가(eager evaluation)**가 적용된다.
- `orElseGet(Supplier supplier)`은 빈 값이면 `supplier`를 통해 값을 생성하기 때문에, **값이 있을 때는 `supplier`가 호출되지 않는다.**
  - 생성 비용이 높은 객체를 다룰 때는 `orElseGet()`이 더 효율적이다.
  - `orElseGet()`에 넘기는 표현식은 **필요할 때만 평가**하므로 **지연 평가(lazy evaluation)**가 적용된다.

사용 용도

`orElse(T other)`

- **값이 이미 존재할 존재하지 않을 가능성이 높거나**, 혹은 **orElse()에 넘기는 객체(또는 메서드)가 생성 비용이 크지 않은 경우** 사용해도 괜찮다.
- 연산이 없는 상수나 변수의 경우 사용해도 괜찮다.

`orElseGet(Supplier supplier)`

- 주로 **orElse()에 넘길 값의 생성 비용이 큰 경우**, 혹은 **값이 들어있을 확률이 높아 굳이 매번 대체 값을 계산할 필요가 없는 경우**에 사용한다.

정리하면, **단순한 대체 값**을 전달하거나 코드가 매우 간단하다면 `orElse()`를 사용하고, **객체 생성 비용이 큰 로직**이 들어있고, **Optional에 값이 이미 존재할 가능성이 높다면** `orElseGet()`을 고려해볼 수 있다.

실전 활용 1 - 주소 찾기

실전 활용 1 - User와 Address

- User라는 클래스가 있고, 그 안에 Address라는 주소 정보가 있을 수 있음.
- 없을 수도 있으므로, 클래스 설계시 address 필드 null일 수 있다고 가정

```
package optional;

import optional.model.Address;
import optional.model.User;

public class AddressMain1 {
    public static void main(String[] args) {
```

```
        User user1 = new User("user1", null);
        User user2 = new User("user2", new Address("hello street"));

        printStreet(user1);
        printStreet(user2);
    }

    static void printStreet(User user) {
        String userStreet = getUserStreet(user);
        if (userStreet != null) {
            System.out.println(userStreet);
        }else{
            System.out.println("UNKNOWN");
        }
    }
    static String getUserStreet(User user) {
        if (user == null) { // null check1
            return null;
        }
        Address address = user.getAddress();
        if(address == null) { // null check2
            return null;
        }

        return address.getStreet();
    }
}
```

- null 체크가 여러 번 등장하고, getUserStreet()메서드도 언제든 null을 반환할 수 있음.
- null 체크 구문으로 인해 불필요한 코드가 많아짐.

Optional로 개선

```
package optional;

import optional.model.Address;
import optional.model.User;
```

```
import java.util.Optional;

public class AddressMain2 {
    public static void main(String[] args) {
        User user1 = new User("user1", null);
        User user2 = new User("user2", new Address("hello street"));

        printStreet(user1);
        printStreet(user2);
    }

    static void printStreet(User user) {
        getUserStreet(user).ifPresentOrElse(
                System.out::println, // 값이 있을 때,
                () -> System.out.println("UNKNOWN") // 값이 없을 때
        );

    }
    static Optional<String> getUserStreet(User user) {
        return Optional.ofNullable(user) // user가 null 일 수 있으므로 ofNullable
                .map(User::getAddress)
                .map(Address::getStreet);

        // map 체이닝 중간에 null이면, Optional.empty()를 반환
    }
}
```

- ifPresentOrElse()등을 통해 코드가 간결해지고, 의도가 명확해짐

실전 활용 2 - 배송

Order와 Delivery

- Order라는 주문 클래스가 있고, 내부에 Delivery(배송) 정보가 있을 수 있음.
- 각 주문의 배송 상태를 출력
- 배송 정보가 없거나, 배송이 취소된 경우 배송X라고 표시.

```
package optional;

import optional.model.Delivery;
import optional.model.Order;

import java.util.HashMap;
import java.util.Map;
import java.util.Optional;

public class DeliveryMain {
    static Map<Long, Order> orderRepository = new HashMap<>();
    static {
        orderRepository.put(1L, new Order(1L, new Delivery("배송완료", false)));
        orderRepository.put(2L, new Order(2L, new Delivery("배송중", false)));
        orderRepository.put(3L, new Order(3L, new Delivery("배송중", true)));
        orderRepository.put(4L, new Order(4L, null));
    }

    public static void main(String[] args) {
        System.out.println("1 = " + getDeliveryStatus(1L));
        System.out.println("2 = " + getDeliveryStatus(2L));
        System.out.println("3 = " + getDeliveryStatus(3L));
        System.out.println("4 = " + getDeliveryStatus(4L));

    }

    private static String getDeliveryStatus(Long orderId) {
        return findOrder(orderId)
                .map(Order::getDelivery)
                .filter(delivery -> !delivery.isCanceled())
                .map(Delivery::getStatus)
                .orElse("배송X");
    }

    static Optional<Order> findOrder(Long orderId) {
        return Optional.ofNullable(orderRepository.get(orderId));
```

```
    }
}
```

- Optional을 활용하여, 중첩된 null체크 없이도 의미 있는 로직을 간결하게 작성 가능.

베스트 프랙티스

- Optional이 좋아보여도, 무분별하게 사용하면 오히려 코드 가독성과 유지보수에 도움이 되지 않음.
- 주로 메서드의 반환값에 대해 값이 없을 수도 있음을 표현하기 위해 도입.
- 메서드 반환 값에 Optional을 사용하라는 것.

1. 반환 타입으로만 사용하고, 필드에는 가급적 쓰지 말기.

원칙

- `Optional`은 주로 **메서드의 반환값**에 대해 "값이 없을 수도 있음"을 표현하기 위해 도입되었다.
- 클래스의 필드(멤버 변수)에 `Optional`을 직접 두는 것은 권장하지 않는다.

잘못된 예시

```
public class Product {
    // 안티 패턴: 필드를 Optional로 선언
    private Optional<String> name;

    // ... constructor, getter, etc.
}
```

- 이렇게 되면 다음과 같은 3가지 상황이 발생한다.
  1. `name = null`
  2. `name = Optional.empty()`
  3. `name = Optional.of(value)`
- `Optional` 자체도 참조 타입이기 때문에, 혹시라도 개발자가 부주의로 `Optional` 필드에 `null`을 할당하면, 그 자체가 `NullPointerException`을 발생시킬 여지를 남긴다.
- 값이 없음을 명시하기 위해 사용하는 것이 `Optional`인데, 정작 필드 자체가 `null`이면 혼란이 가중된다.

권장 예시

```
public class Product {
    // 필드는 원시 타입(혹은 일반 참조 타입) 그대로 둔다.
    private String name;

    // ... constructor, getter, etc.
}
```

```
// name 값을 가져올 때, "필드가 null일 수도 있음"을 고려해야 한다면
// 다음 메서드에서 Optional로 변환해서 반환할 수 있다.
public Optional<String> getNameAsOptional() {
    return Optional.ofNullable(name);
}
```

- 만약 `Optional`로 `name` 값을 받고 싶다면, 필드는 `Optional`을 사용하지 않고, **반환하는 시점**에 `Optional`로 감싸주는 것이 일반적으로 더 나은 방법이다.

2. 메서드 매개변수로 Optional을 사용하지 말기.

원칙

- 자바 공식 문서에 `Optional`은 메서드의 **반환값**으로 사용하기를 권장하며, **매개변수**로 사용하지 말라고 명시되어 있다.
- 호출하는 측에서는 단순히 `null` 전달 대신 `Optional.empty()`를 전달해야 하는 부담이 생기며, 결국 `null`을 사용하든 `Optional.empty()`를 사용하든 큰 차이가 없어 가독성만 떨어진다.

잘못된 예시

```
public void processOrder(Optional<Long> orderId) {
    if (orderId.isPresent()) {
        System.out.println("Order ID: " + orderId.get());
    } else {
        System.out.println("Order ID is empty!");
    }
}
```

- 호출하는 입장에서는 `processOrder(Optional.empty())`처럼 호출해야 하는데, 사실 `processOrder(null)`과 큰 차이가 없고, 오히려 `Optional.empty()`를 만드는 비용이 추가된다.

3. 컬렉션(Collection)이나 배열 타입을 Optional로 감싸지 말기.

원칙

- `List<T>`, `Set<T>`, `Map<K,V>` 등 **컬렉션(Collection)** 자체는 **비어있는 상태(empty)를 표현**할 수 있다.
- 따라서 `Optional<List<T>>`처럼 다시 감싸면 `Optional.empty()`와 "빈 리스트"(`Collections.emptyList()`)가 이중 표현이 되고, 혼란을 야기한다.

잘못된 예시

```
public Optional<List<String>> getUserRoles(String userId) {
    List<String> userRolesList ...;
    if (foundUser) {
        return Optional.of(userRolesList);
    } else {
        return Optional.empty();
    }
}
```

반환 받은 쪽에서는 다음 코드와 같이 사용해야 한다.

```
Optional<List<String>> optList = getUserRoles("someUser");
if (optList.isPresent()) {
    // ...
}
```

하지만 정작 내부의 리스트가 `empty`일 수도 있으므로, 한 번 더 체크해야 하는 모호함이 생긴다.

- `Optional`이 비어있는지 체크해야 하고, `userRolesList`가 비어있는지 추가로 체크해야 한다.

4. isPresent와 get 메서드를 조합해서 사용하지 않기.

원칙

- `Optional`의 `get()` 메서드는 가급적 사용하지 않아야 한다.
- `if (opt.isPresent()) { ... opt.get() ... } else { ... }`는 사실상 `null` 체크와 다를 바 없으며, 깜빡하면 `NoSuchElementException` 같은 예외가 발생할 위험이 있다.
- 대신 `orElse`, `orElseGet`, `orElseThrow`, `ifPresentOrElse`, `map`, `filter` 등의 메서드를 활용하면 간결하고 안전하게 처리할 수 있다.

잘못된 예시

```
public static void main(String[] args) {
    Optional<String> optStr = Optional.ofNullable("Hello");

    if (optStr.isPresent()) {
        System.out.println(optStr.get());
    } else {
        System.out.println("Nothing");
    }
}
```

5. orElseGet() vs orElse() 차이를 분명히 이해하기

원칙

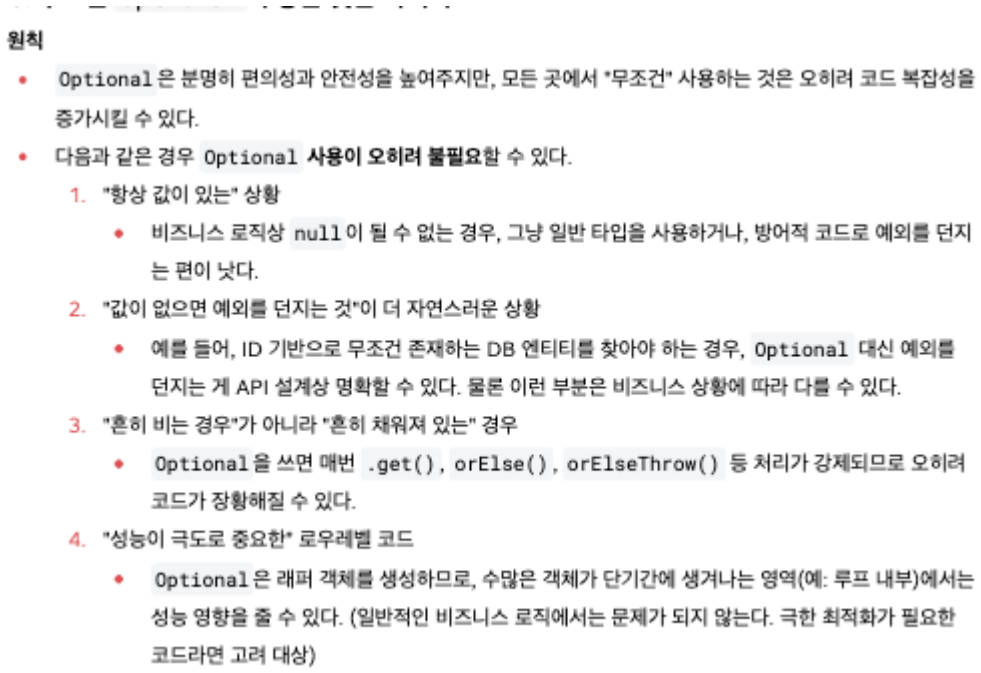
- `orElse(T other)`는 **항상** `other`를 즉시 생성하거나 계산한다.
  - 즉, `Optional` 값이 존재해도 불필요한 연산/객체 생성이 일어날 수 있다. (**즉시 평가**)
- `orElseGet(Supplier<? extends T>)`는 **필요할 때만**(빈 `Optional`일 때만) `Supplier`를 호출한다.
  - 값이 이미 존재하는 경우에는 `Supplier`가 실행되지 않으므로, 비용이 큰 연산을 뒤로 미룰 수 있다(**지연 평가**).

예제 코드는 앞서 알아보았으므로 생략한다.

정리

- **비용이 크지 않은(또는 간단한 상수 정도) 대체값**이라면 간단하게 `orElse()`를 사용하자.
- **복잡하고 비용이 큰 객체 생성**이 필요한 경우, 그리고 **Optional 값이 이미 존재할 가능성이 높다면** `orElseGet()`를 사용하자.

6. 무조건 Optional이 좋은 것은 아니다.



원칙

- `Optional`은 분명히 편의성과 안전성을 높여주지만, 모든 곳에서 "무조건" 사용하는 것은 오히려 코드 복잡성을 증가시킬 수 있다.
- 다음과 같은 경우 `Optional` **사용이 오히려 불필요**할 수 있다.
  1. "항상 값이 있는" 상황
     - 비즈니스 로직상 `null`이 될 수 없는 경우, 그냥 일반 타입을 사용하거나, 방어적 코드로 예외를 던지는 편이 낫다.
  2. "값이 없으면 예외를 던지는 것"이 더 자연스러운 상황
     - 예를 들어, ID 기반으로 무조건 존재하는 DB 엔티티를 찾아야 하는 경우, `Optional` 대신 예외를 던지는 게 API 설계상 명확할 수 있다. 물론 이런 부분은 비즈니스 상황에 따라 다를 수 있다.
  3. "흔히 비는 경우"가 아니라 "흔히 채워져 있는" 경우
     - `Optional`을 쓰면 매번 `.get()`, `orElse()`, `orElseThrow()` 등 처리가 강제되므로 오히려 코드가 장황해질 수 있다.
  4. "성능이 극도로 중요한" 로우레벨 코드
     - `Optional`은 래퍼 객체를 생성하므로, 수많은 객체가 단기간에 생겨나는 영역(예: 루프 내부)에서는 성능 영향을 줄 수 있다. (일반적인 비즈니스 로직에서는 문제가 되지 않는다. 극한 최적화가 필요한 코드라면 고려 대상)

정리

1. 필드는 지양, 메서드 반환 값에 Optional 사용
2. 메서드 파라미터로 Optional을 받지 말 것.
3. 컬렉션은 굳이 Optional로 감싸지 말고, 빈 컬렉션 반환
4. isPresent() + get() 대신 다양한 메서드(orElse, orElseGet, ifPresentOrElse 등) 활용
5. orElseGet() vs orElse() : 지연 평가 vs 즉시 평가

반환 타입, 지역 변수 정도에 사용하는 정도는 괜찮음.

클라이언트 메서드 vs 서버 메서드

호출 하는 메서드 - 클라이언트 메서드 - 가져다 쓰는 곳

호출 당하는 메서드 - 서버 메서드

사실 `Optional`을 고려할 때 가장 중요한 핵심은 `Optional`을 생성하고 반환하는 서버쪽 메서드가 아니라, `Optional`을 반환하는 코드를 호출하는 클라이언트 메서드에 있다. 결과적으로 `Optional`을 반환받는 클라이언트의 입장을 고려해서 하는 선택이, `Optional`을 가장 잘 사용하는 방법이다.

- "이 로직은 `null`을 반환할 수 있는가?"
- "`null`이 가능하다면, 호출하는 사람 입장에서 '값이 없을 수도 있다'는 사실을 명시적으로 인지할 필요가 있는가?"
- "`null`이 적절하지 않고, 예외를 던지는 게 더 맞진 않은가?"

위와 같이 서버 메서드를 작성할 때, 클라이언트 코드를 고려하면서 `Optional`을 적용하면, 더욱 깔끔하고 안전한 코드를 작성할 수 있다.

Optional 기본형 타입 지원

OptionalInt, OptionalLong과 같은 기본형 타입의 Optional도 있지만,

잘 사용하지 않음.

- `Optional<T>`와 달리 `map()`, `flatMap()` 등의 다양한 연산 메서드를 제공하지 않는다. 그래서 범용적으로 활용하기보다는 특정 메서드(`isPresent()`, `getAsInt()` 등)만 사용하게 되어, 일반 `Optional<T>`처럼 메서드 체인을 이어 가며 코드를 간결하게 작성하기 어렵다.
- 기존에 이미 `Optional<T>`를 많이 사용하고 있는 코드베이스에서, 특정 상황만을 위해 `OptionalInt` 등을 섞어 쓰면 오히려 가독성을 떨어뜨린다.

원시 타입 Optional을 고려해볼 만한 경우

- **일반적인 상황**에서는 `Optional<T>` 하나로 통일하는 편이 가독성과 유지보수 면에서 유리하고, 충분히 빠른 성능을 제공한다.
- **예외적으로** 미세한 성능을 극도로 추구하거나, 기본형 타입 스트림을 직접 다루면서 중간에 `OptionalInt`, `OptionalLong`, `OptionalDouble`을 자연스럽게 얻는 상황이라면 이를 사용하는 것도 괜찮다.

기본형 `Optional`의 존재는 박싱/언박싱을 없애고 성능을 조금 더 높일 수 있다라는 선택지를 제공하지만, 실제로는 별도로 쓰는 게 좋을 정도로 성능 문제가 크게 나타나느냐?를 고민해야 한다. 대부분의 경우에는 그렇지 않기 때문에 잘 사용되지는 않는다.

정리