10. Optional

#1.인강/0.자바/7.자바-고급3편

옵셔널이 필요한 이유

NullPointerException(NPE) 문제

- 자바에서 `null`은 "값이 없음"을 표현하는 가장 기본적인 방법이다.
- 하지만 `null`을 잘못 사용하거나 `null` 참조에 대해 메서드를 호출하면 `NullPointerException`(NPE)이 발생하여 프로그램이 예기치 않게 종료될 수 있다.
- 특히 여러 메서드가 연쇄적으로 호출되어 내부에서 `null` 체크가 누락되면, 추적하기 어렵고 디버깅 비용이 증가한다.

가독성 저하

- `null`을 반환하거나 사용하게 되면, 코드를 작성할 때마다 `if (obj != null) { ... } else { ... }` 같은 조건문으로 `null` 여부를 계속 확인해야 한다.
- 이러한 `null` 체크 로직이 누적되면 코드가 복잡해지고 가독성이 떨어진다.

의도가 드러나지 않음

- 메서드 시그니처(`String findNameById(Long id)`)만 보고서는 이 메서드가 `null`을 반환할 수도 있다는 사실을 명확히 알기 어렵다.
- 호출하는 입장에서는 "반드시 값이 존재할 것"이라고 가정했다가, 런타임에 `null`이 나와서 문제가 생길 수 있다.

Optional의 등장

- 이러한 문제를 해결하고자 자바 8부터 `Optional` 클래스를 도입했다.
- `Optional`은 "값이 있을 수도 있고 없을 수도 있음"을 명시적으로 표현해주어, 메서드의 계약(Contract)이나 호출 의도를 좀 더 분명하게 드러낸다.
- `Optional`을 사용하면 "빈 값"을 표현할 때, 더 이상 `null` 자체를 넘겨주지 않고 `Optional.empty()`처럼 의도를 드러내는 객체를 사용할 수 있다.
- 그 결과, `Optional`을 사용하면 `null` 체크 로직을 간결하게 만들고, 특정 경우에 NPE가 발생할 수 있는 부분을 빌드 타임이나 IDE, 코드 리뷰에서 더 쉽게 파악할 수 있게 해준다.

예제 코드를 통해 자세히 살펴보자.

null을 직접 반환하는 경우

```
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");
        // 3L은 넣지 않아서 찾을 수 없는 ID로 활용
    }

    public static void main(String[] args) {
        findAndPrint(1L); // 값이 있는 경우
        findAndPrint(3L); // 값이 없는 경우
    }

    // 이름이 있으면 이름을 대문자로 출력, 없으면 "UNKNOWN"을 출력해라.
    static void findAndPrint(Long id) {
        String name = findNameById(id);
        // 1. NullPointerException 유발
        //System.out.println("name = " + name.toUpperCase());

        // 2. if 문을 활용한 null 체크 필요
        if (name != null) {
```

```
            System.out.println(id + ": " + name.toUpperCase());
        } else {
            System.out.println(id + ": " +  "UNKNOWN");
        }
    }

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

실행 결과

```
1: KIM
3: UNKNOWN
```

- 이 예제에서는 `map.get(id)` 결과가 존재하지 않을 경우 `null`을 반환한다.
- 따라서 `findAndPrint()` 메서드에서 `String name = findNameById(id);` 실행 후, `name`이 `null`인지 아닌지 매번 확인해야 한다.
- `null`을 반환하는 방식을 택하면, 메서드를 호출하는 쪽에서 반드시 `null` 체크 로직을 작성해주어야 하고, 이를 빠뜨리면 `NullPointerException`이 발생한다.
- 반환 타입이 `String`이므로 얼핏 보면 항상 문자열이 반환될 것처럼 보이지만, 실제로는 `null`이 반환될 수도 있어 주의가 필요하다.

Optional을 반환하는 경우

```
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 있는 경우
        findAndPrint(3L); // 값이 없는 경우
    }

    // 이름이 있으면 이름을 대문자로 출력, 없으면 "UNKNOWN"을 출력해라.
    static void findAndPrint(Long id) {
        Optional<String> optName = findNameById(id);
        String name = optName.orElse("UNKNOWN");
        System.out.println(id + ": " + name.toUpperCase());
    }

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

실행 결과

```
1: KIM
3: UNKNOWN
```

- 이번에는 `Optional<String>`을 반환하도록 변경했다.
- `Optional.ofNullable(findId)`를 통해 `null`이 될 수도 있는 값을 `Optional`로 감싼다.
- 메서드 시그니처(`Optional<String> findNameById(Long id)`)만 보고도 "반환 결과가 있을 수도, 없을 수도 있겠구나"라고 명시적으로 인지할 수 있다.
- `Optional.orElse("대체값")` : 옵셔널에 값이 있으면 해당 값을 반환하고, 값이 없다면 대체값을 반환한다.
- `findAndPrint()` 메서드에서는 `Optional<String>`을 받아서, `orElse("UNKNOWN")`로 "빈 값"인 경우 대체 문자열을 지정할 수 있다.
- 이 방식은 "값이 없을 수도 있다"는 점을 호출하는 측에 명확히 전달하므로, 놓치기 쉬운 `null` 체크를 강제하고 코드의 안정성을 높인다.

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

`Optional`을 생성하는 방법은 다음과 같다.

Optional을 생성하는 메서드

- `Optional.of(T value)`
  - 내부 값이 확실히 `null`이 아닐 때 사용. `null`을 전달하면 `NullPointerException` 발생
- `Optional.ofNullable(T value)`
  - 값이 `null`일 수도 있고 아닐 수도 있을 때 사용. `null`이면 `Optional.empty()`를 반환한다.
- `Optional.empty()`
  - 명시적으로 "값이 없음"을 표현할 때 사용

코드로 알아보자.

```
package optional;

import java.util.Optional;

public class OptionalCreationMain {

    public static void main(String[] args) {
        // 1. of() : 값이 null이 아님이 확실할 때 사용, null이면 NullPointerException
발생
        String nonNullValue = "Hello Optional!";
        Optional<String> opt1 = Optional.of(nonNullValue);
        System.out.println("opt1 = " + opt1);  // Optional[Hello Optional!]

        // 2. ofNullable() : 값이 null일 수도 아닐 수도 있을 때
        Optional<String> opt2 = Optional.ofNullable("Hello!");
        Optional<String> opt3 = Optional.ofNullable(null);
        System.out.println("opt2 = " + opt2);  // Optional[Hello!]
        System.out.println("opt3 = " + opt3);  // Optional.empty

        // 3. empty() : 비어있는 Optional을 명시적으로 생성
        Optional<String> opt4 = Optional.empty();
        System.out.println("opt4 = " + opt4);  // Optional.empty
    }
}
```

실행 결과

```
opt1 = Optional[Hello Optional!]
opt2 = Optional[Hello!]
opt3 = Optional.empty
```

```
opt4 = Optional.empty
```

Optional 값 획득

`Optional`에 들어있는 값을 꺼내는 여러 가지 방법을 알아보자.

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 준비
        Optional<String> optValue = Optional.of("Hello");
        Optional<String> optEmpty = Optional.empty();

        // isPresent() : 값이 있으면 true
        System.out.println("=== 1. isPresent() / isEmpty() ===");
        System.out.println("optValue.isPresent() = " +
optValue.isPresent()); // true
        System.out.println("optEmpty.isPresent() = " +
optEmpty.isPresent()); // false
        // isEmpty() : 값이 없으면 true
        System.out.println("optEmpty.isEmpty() = " + optEmpty.isEmpty());

        // get() : 직접 내부 값을 꺼냄, 값이 없으면 예외 (NoSuchElementException)
        System.out.println("=== 2. get() ===");
        String getValue = optValue.get();
        System.out.println("getValue = " + getValue);
        //String getValue2 = optEmpty.get();  // 예외 발생 -> 주석 처리

        // 값이 있으면 그 값, 없으면 지정된 기본값 사용
        System.out.println("=== 3. orElse() ===");
        String value1 = optValue.orElse("기본값");
        String empty1 = optEmpty.orElse("기본값");
        System.out.println("value1 = " + value1);
        System.out.println("empty1 = " + empty1);

        // 값이 없을 때만 람다(Supplier)가 실행되어 기본값 생성
        System.out.println("=== 4. orElseGet() ===");
        String value2 = optValue.orElseGet(() -> {
            System.out.println("람다 호출 - optValue");
            return "New Value";
        });
        String empty2 = optEmpty.orElseGet(() -> {
            System.out.println("람다 호출 - optEmpty");
            return "New Value";
        });
        System.out.println("value2 = " + value2);
        System.out.println("empty2 = " + empty2);

        // 값이 있으면 반환, 없으면 예외 발생
        System.out.println("=== 5. orElseThrow() ===");
```

```
        String value3 = optValue.orElseThrow(() -> new RuntimeException("값이 없습니다!"));
        System.out.println("value3 = " + value3);

        try {
            // optEmpty는 값이 없으므로 예외 발생
            String empty3 = optEmpty.orElseThrow(() -> new RuntimeException("값이 없습니다!"));
            System.out.println("empty3 = " + empty3); // 실행되지 않음
        } catch (Exception e) {
            System.out.println("예외 발생: " + e.getMessage());
        }

        // Optional을 반환
        System.out.println("=== 6. or() ===");
        Optional<String> result1 = optValue.or(() -> Optional.of("Fallback"));
        System.out.println(result1); // Optional[Hello], 값이 이미 존재 -> 원본 그대로

        Optional<String> result2 = optEmpty.or(() -> Optional.of("Fallback"));
        System.out.println(result2); // Optional[Fallback], 비어있으므로 대체 Optional 반환
    }
}
```

실행 결과

```
=== 1. isPresent() / isEmpty() ===
optValue.isPresent() = true
optEmpty.isPresent() = false
optEmpty.isEmpty() = true
=== 2. get() ===
getValue = Hello
=== 3. orElse() ===
value1 = Hello
empty1 = 기본값
=== 4. orElseGet() ===
람다 호출 - optEmpty
value2 = Hello
empty2 = New Value
=== 5. orElseThrow() ===
value3 = Hello
```

```
예외 발생: 값이 없습니다!
=== 6. or() ===
Optional[Hello]
Optional[Fallback]
```

- `optValue.isPresent()`가 `true`인 이유는 `Optional.of("Hello")`로 생성했기 때문이다.
- `optEmpty.isPresent()`가 `false`인 이유는 `Optional.empty()`로 생성했기 때문이다.
- `isEmpty()`는 자바 11부터 사용할 수 있는 메서드로, 값이 없으면 `true`를 반환한다.
- `get()` 메서드는 `Optional` 사용 시 가능하면 피해야 한다.
  - 값이 없는 상태(`Optional.empty()`)에서 `get()`을 호출하면 바로 예외가 터지므로, 안전하게 사용하려면 `isPresent()` 같은 사전 체크가 필요하다.
  - `get()` 보다는 `orElse()`, `orElseGet()`, `orElseThrow()` 등의 메서드를 활용하면 좀 더 세련되고 안전하게 값을 처리할 수 있다.

여기서 `orElse()`, `orElseGet()`의 차이가 잘 느껴지지 않을 수 있다. 지금은 값이 없는 경우 기본값을 제공한다 정도로 이해하자. 이 부분은 뒤에서 자세히 설명하겠다.

Optional 값 처리

`Optional`에서는 값이 존재할 때와 존재하지 않을 때를 처리하기 위한 다양한 메서드들을 제공한다.
이를 활용하면, `null` 체크 로직 없이도 안전하고 간결하게 값을 다룰 수 있다.

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 아무 일도 하지 않음
        System.out.println("=== 1. ifPresent() ===");
        optValue.ifPresent(v -> System.out.println("optValue 값: " + v));
        optEmpty.ifPresent(v -> System.out.println("optEmpty 값: " + v)); // 실
행 X

        // 값이 있으면 Consumer 실행, 없으면 Runnable 실행
        System.out.println("=== 2. ifPresentOfElse() ===");
        optValue.ifPresentOrElse(
                v -> System.out.println("optValue 값: " + v),
                () -> System.out.println("optValue는 비어있음")
        );
        optEmpty.ifPresentOrElse(
                v -> System.out.println("optEmpty 값: " + v),
                () -> System.out.println("optEmpty는 비어있음")
        );

        // 값이 있으면 Function 적용 후 Optional로 반환, 없으면 Optional.empty()
        System.out.println("=== 3. map() ===");
        Optional<Integer> lengthOpt1 = optValue.map(String::length);
        System.out.println("optValue.map(String::length) = " + lengthOpt1);

        Optional<Integer> lengthOpt2 = optEmpty.map(String::length);
        System.out.println("optEmpty.map(String::length) = " + lengthOpt2);
```

```
        // map()과 유사하나, 이미 Optional을 반환하는 경우 중첩을 제거
        System.out.println("=== 4. flatMap() ===");
        System.out.println("[map]");
        // Optional[Hello] -> Optional[Optional[HELLO]]
        Optional<Optional<String>> nestedOpt = optValue.map(s ->
Optional.of(s));
        System.out.println("optValue = " + optValue);
        System.out.println("nestedOpt = " + nestedOpt);

        System.out.println("[flatMap]");
        // flatMap을 사용하면 한 번에 평탄화
        // Optional[Hello] -> Optional[HELLO]
        Optional<String> flattenedOpt = optValue.flatMap(s -> Optional.of(s));
        System.out.println("optValue = " + optValue);
        System.out.println("flattenedOpt = " + flattenedOpt);

        System.out.println("=== 5. filter() ===");
        // 값이 있고 조건을 만족하면 그 값을 그대로, 불만족시 Optional.emtpy()
        Optional<String> filtered1 = optValue.filter(s -> s.startsWith("H"));
        Optional<String> filtered2 = optValue.filter(s -> s.startsWith("X"));

        System.out.println("filter(H) = " + filtered1); // Optional[Hello]
        System.out.println("filter(X) = " + filtered2); // Optional.empty

        System.out.println("=== 6. stream() ===");
        // 값이 있으면 단일 요소 스트림, 없으면 빈 스트림
        optValue.stream()
                .forEach(s -> System.out.println("optValue.stream() -> " +
s));

        // 값이 없으므로 실행 안 됨
        optEmpty.stream()
                .forEach(s -> System.out.println("optEmpty.stream() -> " +
s));
    }
}
```

실행 결과

```
=== 1. ifPresent() ===
optValue 값: Hello
```

```
=== 2. ifPresentOfElse() ===
optValue 값: Hello
optEmpty는 비어있음
=== 3. map() ===
optValue.map(String::length) = Optional[5]
optEmpty.map(String::length) = Optional.empty
=== 4. flatMap() ===
[map]
optValue = Optional[Hello]
nestedOpt = Optional[Optional[Hello]]
[flatMap]
optValue = Optional[Hello]
flattenedOpt = Optional[Hello]
=== 5. filter() ===
filter(H) = Optional[Hello]
filter(X) = Optional.empty
=== 6. stream() ===
optValue.stream() -> Hello
```

`Optional`의 다양한 메서드를 활용하면, 값이 존재할 때와 존재하지 않을 때의 로직을 명확하고 간결하게 구현할 수 있다.
이러한 기능 덕분에 `null` 체크로 인한 복잡한 코드와 예외 처리를 줄이고, 더 읽기 쉽고 안전한 코드를 작성할 수 있다.

즉시 평가와 지연 평가1

앞서 설명한 `orElse()`와 `orElseGet()`의 차이가 잘 느껴지지 않을 수 있다. 둘의 차이를 제대로 이해하려면 즉시 평가와 지연 평가를 먼저 이해해야 한다.

- **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

여기서 평가라고 하는 것은 쉽게 이야기해서 계산이라고 생각하면 된다.
둘의 차이를 이해하기 위해 간단한 로거 예제를 만들어보자.

로거

```
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 받음
    public void debug(Object message) {
        if (isDebug) {
            System.out.println("[DEBUG] " + message);
        }
    }
}
```

- 이 로거의 사용 목적은 일반적인 상황에서는 로그를 남기지 않다가, 디버깅이 필요한 경우에만 디버깅용 로그를 추가로 출력하는 것이다.
- `debug()`에 전달한 메시지는 `isDebug` 값을 `true`로 설정한 경우에만 메시지를 출력한다.

```
package optional.logger;

public class LogMain1 {

    public static void main(String[] args) {
        Logger logger = new Logger();
        logger.setDebug(true);
        logger.debug(10 + 20);

        System.out.println("=== 디버그 모드 끄기 ===");
```

```
        logger.setDebug(false);
        logger.debug(100 + 200);
    }

}
```

실행 결과

```
[DEBUG] 30
=== 디버그 모드 끄기 ===
```

- `debug(10 + 20)` 은 디버그 모드가 켜져있기 때문에 출력된다.
- `debug(100 + 200)` 은 디버그 모드가 꺼져있기 때문에 출력되지 않는다.

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 평가2

`100 + 200` 연산을 메서드 호출로 변경해서, 실제 호출된 것인지 확인해보자.

```
package optional.logger;

public class LogMain2 {

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

        System.out.println("=== 디버그 모드 끄기 ===");
        logger.setDebug(false);
        logger.debug(value100() + value200());
    }

    static int value100() {
```

```
        System.out.println("value100 호출");
        return 100;
    }

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

}
```

실행 결과

```
value100 호출
value200 호출
[DEBUG] 300
=== 디버그 모드 끄기 ===
value100 호출
value200 호출
```

로그를 보면 디버그 모드를 끈 경우에도 `value100()`, `value200()`이 실행된 것을 확인할 수 있다. 따라서 메서드를 호출하기 전에 괄호 안의 내용이 먼저 평가(계산)되는 것을 확인할 수 있다.

```
// 자바 언어의 연산자 우선순위상 메서드를 호출하기 전에 괄호 안의 내용이 먼저 계산된다.
logger.debug(value100() + value200()); // 1. 여기서는 value100() + value200()이 즉시 평가된다.
logger.debug(100 + value200());        // 2. 왼쪽에 있는 value100()이 먼저 평가되고, 반환된다.
출력: value1 호출
logger.debug(100 + 200);               // 3. 오른쪽에 있는 value200()이 평가되고, 반환된다.
출력: value2 호출
debug(300)                // 4. 100 + 200을 평가하고 메서드를 호출한다. 300의 값이 인자로 전달된다.
```

결과적으로 여기서도 `value100() + value200()` 연산은 미래에 전혀 사용하지 않을 값을 계산해서 아까운 CPU 전기만 낭비한 것이다.

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());
}
```

실행 결과

```
value100 호출
value200 호출
[DEBUG] 300
=== 디버그 모드 끄기 ===
value100 호출
value200 호출
=== 디버그 모드 체크 ===
```

실행 결과를 보면 디버그 모드 체크 이후에 아무런 로그가 남지 않았다. 따라서 `debug()` 메서드가 실행되지 않은 것을 확인할 수 있다.

이렇게 하면 코드는 지저분해지겠지만, if문 덕분에 디버그 모드가 꺼져있을 때 필요없는 연산을 계산하지 않아도 된다. 하지만 이렇게 하려면 디버그를 출력할 때 마다 계속 if 문을 사용해야 한다. 코드 한 줄을 작성하는데 코드 2줄이 더 필요하다! (if 문을 닫는 괄호 포함)

```
// 1. if 문을 통한 확실한 체크, 코드는 지저분해지지만, 필요 없는 연산 수행X
if (logger.isDebug()) {
    logger.debug(value100() + value200());
}

// 2. 필요없는 연산이 추가되지만 코드는 깔끔
logger.debug(value100() + value200());
```

2번과 같은 방식으로 깔끔하지만 미래에 사용하지 않을 연산이 미리 수행되지 않도록 하는 방법은 없을까?
2번의 코드의 깔끔함과 1번의 필요 없는 연산은 수행하지 않는 둘의 장점만 가져오는 방법은 없을까?

이렇게 하려면 `100 + 200`, `value100() + value200()` 같은 **연산을 정의하는 시점**과 해당 **연산을 실행하는 시점**을 **분리**해야 한다. 그래서 연산의 실행을 최대한 **지연해서 평가**(계산)해야 한다.

즉시 평가와 지연 평가3

자바 언어에서 연산을 정의하는 시점과 해당 연산을 실행하는 시점을 분리하는 방법은 여러 가지가 있다.

- 익명 클래스를 만들고, 메서드를 나중에 호출
- 람다를 만들고, 해당 람다를 나중에 호출

람다를 사용해서 연산을 정의하는 시점과 실행하는 시점을 분리해서 문제를 해결해보자.
Logger에 람다(`Supplier`)를 받는 debug 메서드를 하나 추가하자

```
package optional.logger;

import java.util.function.Supplier;

public class Logger {

    ...
```

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

- `Supplier`를 통해서 람다를 받도록 했다.
- `Supplier`는 `get()`을 실행할 때 해당 람다를 평가(연산)한다. 그리고 그 결과를 반환한다.

```
package optional.logger;

public class LogMain3 {

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

        System.out.println("=== 디버그 모드 끄기 ===");
        logger.setDebug(false);
        logger.debug(() -> value100() + value200());
    }

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

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

}
```

실행 결과

```
```

```
value100 호출
value200 호출
[DEBUG] 300
=== 디버그 모드 끄기 ===
```

실행 결과를 보면 디버그 모드가 꺼져있을 때 `value100()`, `value200()`이 실행되지 않은 것을 확인할 수 있다.
디버그 모드가 켜져있을 때, 꺼져있을 때 실행 순서를 알아보자.

디버그 모드가 켜져있을 때

```
logger.debug(() -> value100() + value200()) // 1. 람다를 생성한다. 이때 람다가 실행되지는 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 평가(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 실행되지는 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()

우리는 앞서 즉시 평가와 지연 평가에 대해서 알아보았다.
람다를 사용하면 연산을 정의하는 시점과 실행하는 시점을 분리해서, 값이 실제로 필요할 때 까지 계산을 미룰 수 있다.
이제 `orElse()`와 `orElseGet()`의 차이를 이해할 수 있을 것이다.
`orElse()`는 보통 데이터를 받아서 인자가 즉시 평가되고, `orElseGet()`은 람다를 받아서 인자가 지연 평가된다.

다음 코드를 통해 둘의 차이를 자세히 알아보자.

```
package optional;

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

public class OrElseGetMain {

    public static void main(String[] args) {
        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 지정된 기본값 사용
        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 지정된 람다 사용
        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("데이터를 생성합니다...");
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        int createValue = new Random().nextInt(100);
        System.out.println("데이터 생성이 완료되었습니다. 생성 값: " + createValue);
        return createValue;
    }
}
```

실행 결과

```
단순 계산
i1 = 100
```

```
i2 = 30
=== orElse ===
값이 있는 경우
데이터를 생성합니다...
데이터 생성이 완료되었습니다. 생성 값: 14
value1 = 100
값이 없는 경우
데이터를 생성합니다...
데이터 생성이 완료되었습니다. 생성 값: 18
empty1 = 18

=== orElseGet ===
값이 있는 경우
value2 = 100
값이 없는 경우
데이터를 생성합니다...
데이터 생성이 완료되었습니다. 생성 값: 11
empty2 = 11
```

- `orElse(createData())`
  - `Optional`에 값이 있어도 `createData()`가 즉시 호출된다. 호출된 값은 버려진다.
  - 자바 연산 순서상 `createData()`를 호출해야 그 결과를 `orElse()`에 인자로 전달할 수 있다.
- `orElseGet(() -> createData())`
  - `Optional` 값이 있으면 `createData()`가 호출되지 않는다.
  - `orElseGet()`에 람다를 전달한다. 해당 람다는 이후에 `orElseGet()` 안에서 실행될 수 있다.
  - `Optional` 내부에 값이 있다면, 인자로 전달한 람다를 내부에서 실행하지 않는다.
  - `Optional` 내부에 값이 없다면, 인자로 전달한 람다를 내부에서 실행하고, 그 결과를 반환한다.

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 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 활용1 - 주소 찾기

실전 활용1 - User와 Address

시나리오

- `User`라는 클래스가 있고, 그 안에 `Address`라는 주소 정보가 있을 수 있다.
- 주소가 없을 수도 있으므로, 우리가 클래스를 설계할 때 `address` 필드는 `null` 일 수도 있다고 가정한다.

먼저, `User`와 `Address` 클래스를 살펴보자.

```
package optional.model;

public class User {
    private String name;
    private Address address;

    public User(String name, Address address) {
        this.name = name;
        this.address = address;
    }

    public String getName() {
        return name;
    }

    public Address getAddress() {
        return address;
```

```
    }

}
```

```
package optional.model;

public class Address {
    private String street;

    public Address(String street) {
        this.street = street;
    }

    public String getStreet() {
        return street;
    }

}
```

1. null 체크 방식으로 구현

다음 코드는 전통적인 방식으로 `null`을 체크해서 처리하는 방법이다. 주소가 없으면 `"Unknown"`을 출력한다.

```
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) {
            return null;
        }
        Address address = user.getAddress();
        if (address == null) {
            return null;
        }
        return address.getStreet();
    }
}
```

실행 결과

```
Unknown
hello street
```

- `null` 체크가 여러 번 등장하고, `getUserStreet()` 메서드도 언제든 `null`을 반환할 수 있음을 고려해야 한다.
- 따라서 `getUserStreet()`를 호출하는 `printStreet()` 메서드도 `null` 체크 로직이 필요하다.
- 작은 예제에서는 괜찮아 보이지만, 점점 `User` 내부 구조가 복잡해지면 `null` 체크 구문도 늘어난다.

2. Optional로 개선

이번에는 `Optional`을 활용해 동일한 로직을 구현해보자.

```
package optional;
```

```java
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);
    }

    private 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 null일 수 있으므로 ofNullable 사
용
                .map(User::getAddress) // user.getAddress() 처리
                .map(Address::getStreet); // address.getStreet() 처리
        // 여기서 map 체이닝 중간에 null이면 Optional.empty()를 반환
    }
}
```

실행 결과

```
Unknown
hello street
```

- `getUserStreet()` 메서드는 이제 `Optional<String>`을 반환하므로, 호출 측에서 `ifPresentOrElse()`, `orElse()`, `orElseGet()` 등을 통해 안전하게 처리할 수 있다.
- 여러 `map()` 체이닝을 통해 내부에서 `null`이 발생하면 자동으로 `Optional.empty()`로 전환된다.
- `null` 체크 구문이 사라지고, 의도가 더욱 명확해진다.

- `Optional.ofNullable(user)`를 사용한 이유는 `user`의 값이 `null`일 수도 있기 때문이다.

실전 활용2 - 배송

실전 활용2 - Order와 Delivery

시나리오

- `Order`라는 주문 클래스가 있고, 내부에 `Delivery`(배송) 정보가 있을 수 있다.
- 각 주문의 배송 상태를 출력한다.
- 배송 정보가 없거나, 배송이 취소된 경우에는 `"배송X"`라고 표시해야 한다.

먼저 `Order`와 `Delivery` 클래스를 살펴보자.

```
package optional.model;

public class Order {
    private Long id;
    private Delivery delivery;

    public Order(Long id, Delivery delivery) {
        this.id = id;
        this.delivery = delivery;
    }

    public Long getId() {
        return id;
    }

    public Delivery getDelivery() {
        return delivery;
    }
}
```

```
package optional.model;

public class Delivery {
```

```
    private String status;
    private boolean canceled;

    public Delivery(String status, boolean canceled) {
        this.status = status;
        this.canceled = canceled;
    }

    public String getStatus() {
        return status;
    }

    public boolean isCanceled() {
        return canceled;
    }
}
```

다음 예제는 `orderRepository`라는 맵에서 주문 정보를 찾은 다음, 배송 정보를 조회하여 출력하는 코드이다.

- 배송 정보가 `null`이거나, `canceled == true`인 경우에는 `"배송X"`를 출력한다.

```
package optional;

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

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

// Delivery가 없거나, 배송이 취소된 경우 배송X
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));
    }

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

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

실행 결과

```
1 = 배송완료
2 = 배송중
3 = 배송X
4 = 배송X
```

- 주문 정보가 `null` 이면 `Optional.empty()`
- 배송이 없거나 취소된 경우에도 `Optional.empty()` 체이닝
- 최종적으로 `.orElse("배송X")` 처리

이렇게 `Optional`을 활용하면, 중첩된 `null` 체크 없이도 의미 있는 로직을 간결하게 작성할 수 있다.

옵셔널 - 베스트 프랙티스

- `Optional`이 좋아보여도 무분별하게 사용하면 오히려 코드 가독성과 유지보수에 도움이 되지 않을 수 있다.
- `Optional`은 주로 **메서드의 반환값**에 대해 값이 없을 수도 있음을 표현하기 위해 도입되었다.

- 여기서 핵심은 메서드의 반환값에 `Optional`을 사용하라는 것이다.

`Optional`을 사용할 때 자주 이야기되는 **베스트 프랙티스(모범 사례)**를 정리했다.
실무 환경에서 `Optional`을 사용할 때 다음 내용들을 알고 사용하면, `Optional`을 더욱 잘 활용할 수 있을 것이다.

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 `Optional`을 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.

권장 예시

- **오버로드**된 메서드를 만들거나,
- **명시적으로** `null` **허용 여부**를 문서화하는 방식을 택합니다.

```
// 오버로드 예시
public void processOrder(long orderId) {
    // 이 메서드는 orderId가 항상 있어야 하는 경우
    System.out.println("Order ID: " + orderId);
```

```
}

public void processOrder() {
    // 이 메서드는 orderId가 없을 때 호출할 경우
    System.out.println("Order ID is empty!");
}
```

```
// 방어적 코드(여기서는 null 허용, 내부에서 처리)
public void processOrder(Long orderId) {
    if (orderId == null) {
        System.out.println("Order ID is empty!");
        return;
    }
    System.out.println("Order ID: " + orderId);
}
```

어떤 방식이든 `Optional`을 매개변수로 받는 것은 지양하고, **오히려 반환 타입**을 `Optional`로 두는 것이 더 자연스러운 활용 방법이다.

3. 컬렉션(Collection)이나 배열 타입을 `Optional`로 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.

권장 예시

```
public List<String> getUserRoles(String userId) {
    // ...
    if (!foundUser) {
        // 권장: 빈 리스트 반환
        return Collections.emptyList();
    }
    return userRolesList;
}
```

- 빈 컬렉션을 반환하면, 호출 측에서는 단순히 `optList.isEmpty()`로 처리하면 된다.

4. `isPresent()`와 `get()` 조합을 직접 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");
    }
}
```

권장 예시

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

    // 1) orElse
    System.out.println(optStr.orElse("Nothing"));

    // 2) ifPresentOrElse
    optStr.ifPresentOrElse(
            System.out::println,
            () -> System.out.println("Nothing")
    );

    // 3) map
    int length = optStr.map(String::length).orElse(0);
    System.out.println("Length: " + length);
}
```

- 각 메서드(`map`, `filter`, `ifPresentOrElse`, `orElse`, `orElseThrow`, `orElseGet` 등)를 잘 조합하면, `get()` 없이도 대부분의 로직을 처리할 수 있다.
- `get()` 메서드는 가급적 사용하지 말고, 예제 코드나, 간단한 테스트에서만 사용하는 것을 권장한다.
- 그럼에도 불구하고 `get()` 메서드를 사용해야 하는 상황이라면, 이럴 때는 반드시 `isPresent()`와 함께 사용하는 것을 권장한다.

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 `Optional`이 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)

예제 코드

```
// 1. 항상 값이 있는 경우: 차라리 Optional 사용 X
public String findConfigValue() {
    // 이 로직은 무조건 "NotNull" 반환
    // null이 나오면 프로그래밍적 오류
    return "ConfigValue";
}

// 2. 값이 없으면 예외가 맞는 경우
```

```java
public String findRequiredEntity(Long id) {
    // DB나 Repository에서 무조건 존재해야 하는 엔티티
    Entity entity = repository.find(id);
    if (entity == null) {
        throw new IllegalStateException("Required Entity not found!");
    }
    return entity.getName();
}

// 3. null이 날 가능성이 희박하고, 주요 흐름에서 필수로 존재해야 하는 경우
public String getValue(Data data) {
    // 비즈니스상 data.getValue()가 null이면 안 되는 상황이라면?
    // Optional보다 null 체크 후 예외가 더 직관적일 수 있음
    if (data.getValue() == null) {
        throw new IllegalArgumentException("Value is missing, cannot
proceed!");
    }
    return data.getValue();
}
```

정리

1. **필드는 지양**, 메서드 **반환값**에 `Optional` 사용
2. **메서드 파라미터**로 `Optional` 받지 말 것
3. **컬렉션**은 굳이 `Optional`로 감싸지 말고, **빈 컬렉션**을 반환
4. `isPresent() + get()` 대신 **다양한 메서드**(`orElse`, `orElseGet`, `ifPresentOrElse`, `map`, `filter`, `flatMap`, ...) 활용
5. `orElseGet()` **vs** `orElse()`: 즉시 평가 vs 지연 평가
6. **무조건** `Optional`**이 정답은 아니다**: 값이 항상 있어야 하거나, 예외가 더 적절한 곳에는 굳이 쓰지 않는다

`Optional`은 **반환 타입**, 그리고 **지역 변수** 정도에 사용하는 것은 괜찮다.
이처럼 `Optional`을 적극적으로 사용하면 `null`을 직접 다루는 코드보다 가독성이 좋아지고, 런타임 오류(`NullPointerException`)를 줄이는 데 도움이 된다. 그러나 **"모든 상황에 마구 적용하면 오히려 복잡도를 높일 수 있다"**는 점을 고려하자.

클라이언트 메서드 vs 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`, `OptionalLong`, `OptionalDouble`과 같은 기본형 타입의 `Optional`도 있지만 다음과 같은 이유로 잘 사용되지는 않는다.

- `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

자바에서 `Optional`은 `null`로 인해 발생하는 혼동과 예외(`NullPointerException`)를 줄이고, "값이 있을 수도 없을 수도 있음"을 명시적으로 표현하기 위해 도입된 클래스이다. 주요 포인트는 다음과 같다.

1. **왜 Optional이 필요한가?**
   - `null`을 잘못 다루면 `NullPointerException`으로 이어질 수 있고, 여러 곳에서 `null` 체크가 필요해 코드 가독성이 떨어진다.
   - `Optional`은 "값이 비어있을 수도 있음을 명시적으로 표현"해주어, 호출 측에서 쉽게 확인하고 안전하게 처리할 수 있다.

2. **Optional 생성과 값 획득**
   - `Optional.of(value)`, `Optional.ofNullable(value)`, `Optional.empty()` 세 가지로 생성할 수 있으며, 내부 값을 꺼낼 때는 `get()`보다는 `orElse()`, `orElseGet()`, `orElseThrow()` 등의 메서드를 사용하는 것을 권장한다.
   - `orElse()`는 인자를 즉시 평가(**즉시 평가**), `orElseGet()`은 필요할 때만(**지연 평가**) 람다를 실행해 비용을 절약할 수 있다.

3. **Optional 값 처리**
   - `ifPresent()`, `ifPresentOrElse()`, `map()`, `flatMap()`, `filter()` 등 다양한 메서드를 제공해, 존재하는 값만 편리하게 연산하거나 조건을 걸 수 있다.
   - 이 메서드들을 적절히 조합하면, 복잡한 `null` 체크 없이도 직관적이고 안전한 코드를 작성할 수 있다.

4. **즉시 평가와 지연 평가**
   - 자바는 메서드 호출 전 괄호 안의 표현식을 먼저 계산하는(**즉시 평가**) 특징이 있어, `orElse()`에 전달된 값이 필요하지 않아도 미리 생성될 수 있다.
   - `orElseGet()`에 람다(`Supplier`)를 넘기면, 실제로 대체 값이 필요할 때만(**지연 평가**) 람다가 실행되어 불필요한 연산을 방지할 수 있다.

5. **실전 활용 예**
   - **주소 조회**나 **배송 상태**처럼, 중첩된 객체에서 필드가 없거나(`null`), 원하는 값을 찾지 못하는 상황에서 `Optional.map()`, `filter()`, `orElse()` 등을 연쇄적으로 사용하면, `null` 체크 없이도 안전하고 간결하게 코드를 작성할 수 있다.

6. **베스트 프랙티스**
   1. **반환값으로 사용**하되, 클래스 **필드에는** `Optional`**을 쓰지 말기**: 필드에 `Optional`을 두면 `null`과 `empty()`가 중복 표기되어 혼란을 야기할 수 있다.
   2. **메서드 파라미터로** `Optional`**을 받지 말기**: 호출부에서 `Optional.empty()`를 만들어 넘기는 것은 `null`과 큰 차이가 없어 오히려 복잡해진다.
   3. **컬렉션을** `Optional`**로 감싸지 말기**: 이미 리스트, 세트 등의 컬렉션은 빈 상태로 반환할 수 있으므로 `Optional.empty()`와 겹치는 중복 표현이 생긴다.

4. `isPresent()` **와** `get()` **을 직접 조합하지 않기**: `null` 체크와 크게 다를 바 없으며, 예외 발생 위험이 커진다. `ifPresentOrElse()`, `orElse()`, `orElseThrow()` 등 다양한 메서드를 활용하자.
5. `orElse()` **vs** `orElseGet()`: 단순 상수나 작은 비용일 때는 `orElse()`, 객체 생성 비용이 클 때는 `orElseGet()`으로 **지연 평가**를 활용하는 편이 낫다.
6. **무조건** `Optional`**이 정답은 아님**: 비즈니스상 항상 값이 있어야 하는 경우나 예외가 더 자연스러운 상황("존재해야 하는 엔티티" 등)에서는 `Optional` 대신 바로 예외 처리가 더 나은 선택일 수 있다.

정리하면, `Optional`은 "값이 없을 수 있음"을 명확히 표현하고 `null` 체크를 간결히 해주는 좋은 도구이지만, **반환 타입** 외에 과도하게 사용하면 오히려 복잡도를 높일 수 있다. 비용이 큰 객체를 대체 값으로 생성해야 할 때는 `orElseGet()`로 최적화하는 등 상황에 맞게 적절히 활용하는 것이 중요하다.