자바8 Optional 1부: 빠져나올 수 없는 null 처리의 늪

2017-01-01 #FP, #JAVA8, #OPTIONAL, #STREAM

> *Java8이 나오기 이 전에는 얼마나 힘들게 null 처리를 했었는지 살펴봅시다.*

null 창시자가 스스로 인정한 null 탄생의 실수

우선 null이라는 개념은 언제 누구에 의해 만들어졌을까요?
null 참조는 1965년에 Tony Hoare라는 영국의 컴퓨터 과학자에 의해서 처음으로 고안되었습니다.
당시 그는 "존재하지 않는 값"을 표현할 수 있는 가장 편리한 방법이 null 참조라고 생각했다고 합니다.
하지만 나중에 그는 그 당시 자신의 생각이 "10억불 짜리 큰 실수"였고, null 참조를 만든 것을 후회한다고 토로하였습니다.

NPE(NullPointerException)

null 참조로 인해 자바 개발자들이 가장 골치아프게 겪는 문제는 그 악명높은 널 포인터 예외(소위, NPE)일 것입니다.
자바 초보이든 고수이든 객체를 사용하여 모든 것을 표현하는 자바 개발자에게 NPE는 코드베이스 곳곳에 깔려있는 지뢰같은 녀석입니다.
컴파일 타임에서는 조용히 잠복해있다가 런타임 때 펑펑 터지는 NPE의 스택 트레이스에 자바 개발자들은 속수무책으로 당할 수 밖에 없었습니다. :fearful:

```
java.lang.NullPointerException
        at seo.dale.java.practice(OptionalTest.java:26)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces
        at java.lang.reflect.Method.invoke(Method.java:497)
```

null 처리가 취약한 코드

null 처리가 취약한 코드에서는 NPE 발생 확률이 높습니다.
예를 들어 어떤 쇼핑몰에서 다음과 같은 구조의 데이터 모델들이 있다고 가정해보시죠.

```java
/* 주문 */
public class Order {
        private Long id;
        private Date date;
        private Member member;
        // getters & setters
}

/* 회원 */
public class Member {
        private Long id;
        private String name;
        private Address address;
        // getters & setters
}

/* 주소 */
public class Address {
        private String street;
        private String city;
        private String zipcode;
        // getters & setters
}
```

`Order` 클래스는 `Member` 타입의 `member` 필드를 가지며, `Member` 클래스는 다시 `Address` 타입의 `address` 필드를 가집니다.

그리고 "어떤 주문을 한 회원이 어느 도시에 살고 있는지 알아내기"위해서 다음과 같은 메소드가 있다고 가정해봅시다.

```java
/* 주문을 한 회원이 살고 있는 도시를 반환한다 */
public String getCityOfMemberFromOrder(Order order) {
        return order.getMember().getAddress().getCity();
}
```

위 메소드가 얼마나 NPE 위험에 노출된 상태이신지 보이시나요?
(안 보이신다면 평소에 null 처리를 열심히 하시지 않으시는 분으로… :disappointed:)

NPE 발생 시나리오

위 메소드에서 구체적으로 어떤 상황에서 NPE가 발생할까요?
여러 단계로 이뤄진 객체 탐색의 과정을 짚어보면 다음과 같이 NPE 위험 포인트를 도출할 수 있습니다.

1. `order` 파라미터에 null 값이 넘어옴
2. `order.getMember()` 의 결과가 null 임
3. `order.getMember().getAddress()` 의 결과가 null 임
4. `order.getMember().getAddress().getCity()` 의 결과가 null 임

4번째 경우에는 엄밀히 얘기하면 이 메소드 내부에서 NPE가 발생하지 케이스는 아닙니다.
하지만 null을 리턴함으로써 호출부에 NPE 위험을 전파시키는 케이스이므로 포함시켰습니다.

호출부에서 적절히 null 처리를 해주지 않으면, 다음 코드와 같이 호출부에서 NPE를 발생시킬 수 있습니다.

```java
String city = getCityOfMemberFromOrder(order); // returns null
System.out.println(city.length()); // throws NPE!
```

전통적인(?) NPE 방어 패턴

Java8 이전에는 이렇게 NPE의 위험에 노출된 코드를 다음과 같은 코딩 스타일로 회피하였습니다.

1. 중첩 null 체크하기

```java
public String getCityOfMemberFromOrder(Order order) {
        if (order != null) {
                Member member = order.getMember();
                if (member != null) {
                        Address address = member.getAddress();
                        if (address != null) {
                                String city = address.getCity();
                                if (city != null) {
                                        return city;
                                }
                        }
                }
        }
        return "Seoul"; // default
}
```

아! :scream: 정말 끔찍하지만 실무에서 심심치 않게 볼 수 있는 코드입니다.
객체 탐색의 모든 단계마다 null이 반환되지 않을지 의심하면서 null 체크를 합니다.
들여쓰기 때문에 코드를 읽기가 매우 어려우며 핵심 비즈니스 파악이 쉽지 않습니다.

2. 사방에서 return 하기

```java
public String getCityOfMemberFromOrder(Order order) {
        if (order == null) {
                return "Seoul";
        }
        Member member = order.getMember();
        if (member == null) {
                return "Seoul";
        }
        Address address = member.getAddress();
        if (address == null) {
                return "Seoul";
        }
        String city = address.getCity();
        if (city == null) {
                return "Seoul";
        }
        return city;
}
```

첫 번째 코드를 조금 개선해보았습니다. (개인 취향 따라 악화라고 생각하시는 분도 계실 것 같습니다만… :sweat:)
전반적으로 코드 읽기가 조금 쉬워지긴 했지만, 결과를 여러 곳에 리턴하기 때문에 유지 보수하기가 난해해졌습니다.

2가지 방법 모두 기본적으로 객체의 필드나 메소드에 접근하기 전에 null 체크를 함으로써 NPE를 방지하고 있습니다.
하지만 안타깝게도 이로 인해 초기 버전의 메소드보다 코드가 상당히 길어지고 지저분해졌음을 볼 수 있습니다.

이 밖에도 null object 패턴 등 NPE 문제를 해결하기 위한 다양한 시도들이 있었지만 그닥 만족스러운 대안을 찾을 수 없었습니다.

null의 저주

코드가 이 지경에 이르면 과연 문제의 원인이 개발자의 무능함 때문인지 다른 곳에서 근본 원인을 찾아야 하는지 혼란스러워집니다.
애초에 `getCityOfMemberFromOrder()` 메소드에 대한 우리의 요구 사항은 상당히 명확하고 간단했습니다.

"어떤 주문을 한 회원이 어느 도시에 살고 있는지 알려주시오!"

하지만 우리의 코드는 중첩된 if 조건문과 사방에 return 문으로 도배되고 말았습니다.
유지 보수 기간이 길어질수록 비즈니스 로직은 점점 null 체크에 가려지곤 했습니다.
이쯤되면 애초에 우리가 하려던 것이 null 체크인지 비즈니스 로직인지 햇갈리기까지 합니다. :sob:

NPE 때문에 시스템이 다운되서 한 두번 데어보신 분이라면, 위와 같은 코드를 작성하고 있는 자신을 발견하실 것입니다.
장애를 겪을 바엔 코드 가독성과 유지 보수성을 희생하는 게 현실적인 선택이기 때문입니다.

자바 언어는 (대부분의 다른 언어들처럼) "값의 부재"를 나타내기 위해 null을 사용하도록 설계되었습니다.
하지만 null 창시자가 의도 했던 바와 다르게 null은 자바 개발자들에게 NPE 방어라는 끝나지 않는 숙제를 남겼습니다.

Java8의 등장

Java8이 등장하면서 자바 개발자들이 null을 대하는 접근 방식에 커다란 페러다임의 전환을 가져오게 되었습니다.
이 부분에 대해서는 이어지는 포스트에서 알아보도록 하겠습니다.

자바8 Optional 2부: null을 대하는 새로운 방법

2017-01-08 #FP, #JAVA8, #OPTIONAL, #STREAM

> *Java8의 Optional API를 통해 어떻게 null 처리를 할 수 있는지 알아봅시다.*

이전 포스트를 통해 Java8 이 전에는 얼마나 null 처리가 고통스러웠는지 살펴보았습니다.
그리고 문제의 본질이 null 참조를 통해 "값의 부재"를 표현하는 자바 언어의 초기 설계에 기인한다라는 것도 말미에 언급하였습니다.

null 관련 문제 돌아보기

이 전 포스트의 예제 코드를 통해 살펴본 null과 관련된 문제들을 크게 2가지로 요약됩니다.

- 런타임에 NPE(NullPointerException)라는 예외를 발생시킬 수 있습니다.
- NPE 방어를 위해서 들어간 null 체크 로직 때문에 코드 가독성과 유지 보수성이 떨어집니다.

그냥 두자니 곳곳에 숨어서 일으켜 장애를 유발하고, 조치를 하자니 코드를 엉망으로 만드는 null, 어떻하면 좀 더 현명하게 다룰 수 있을까요?

함수형 언어에서 그 해법을 찾다

스칼라나 히스켈과 같은 소위 함수형 언어들은 전혀 다른 방법으로 이 문제을 해결합니다.
자바가 "존재하지 않는 값"을 표현하기 위해서 null을 사용했다면, 이 함수형 언어들은 "존재할지 안 할지 모르는 값"을 표현할 수 있는 별개의 타입을 가지고 있습니다.
그리고 이 타입은 이 존재할지 안 할지도 모르는 값을 제어할 수 있는 여러가지 API를 제공하기 때문에 개발자들 해당 API를 통해서 간접적으로 그 값에 접근하게 됩니다.
Java8은 이러한 함수형 언어의 접근 방식에서 영감을 받아 `java.util.Optional<T>` 라는 새로운 클래스를 도입하였습니다! :tada:

Optional이란?

`Optional` 는 "존재할 수도 있지만 안 할 수도 있는 객체", 즉, "null이 될 수도 있는 객체"을 감싸고 있는 일종의 래퍼 클래스입니다.
원소가 없거나 최대 하나 밖에 없는 `Collection` 이나 `Stream` 으로 생각하셔도 좋습니다.
직접 다루기에 위험하고 까다로운 null을 담을 수 있는 특수한 그릇으로 생각하시면 이해가 쉬우실 것 같습니다.

Optional의 효과

Optional로 객체를 감싸서 사용하시게 되면...

- NPE를 유발할 수 있는 null을 직접 다루지 않아도 됩니다.
- 수고롭게 null 체크를 직접 하지 않아도 됩니다.
- 명시적으로 해당 변수가 null일 수도 있다는 가능성을 표현할 수 있습니다. (따라서 불필요한 방어 로직을 줄일 수 있습니다.)

Optional 기본 사용법

자, 그럼 각설하고 `java.util.Optional<T>` 클래스를 어떻게 사용하는지 좀 더 구체적으로 살펴볼까요?

Optional 변수 선언하기

제네릭을 제공하기 때문에, 변수를 선언할 때 명기한 타입 파라미터에 따라서 감쌀 수 있는 객체의 타입이 결정됩니다.

```java
Optional<Order> maybeOrder; // Order 타입의 객체를 감쌀 수 있는 Optional 타입의 변수
Optional<Member> optMember; // Member 타입의 객체를 감쌀 수 있는 Optional 타입의 변수
Optional<Address> address; // Address 타입의 객체를 감쌀 수 있는 Optional 타입의 변수
```

변수명은 그냥 클래스 이름을 사용하기도 하지만 "maybe"나 "opt"와 같은 접두어를 붙여서 Optional 타입의 변수라는 것을 좀 더 명확히 나타내기도 합니다.

Optional 객체 생성하기

Optional 클래스는 간편하게 객체 생성을 할 수 있도록 3가지 정적 팩토리 메소드를 제공합니다.

- `Optional.empty()`

null을 담고 있는, 한 마디로 비어있는 Optional 객체를 얻어옵니다.
이 비어있는 객체는 Optional 내부적으로 미리 생성해놓은 싱글턴 인스턴스입니다.

```java
Optional<Member> maybeMember = Optional.empty();
```

- `Optional.of(value)`

null이 아닌 객체를 담고 있는 Optional 객체를 생성합니다.
null이 넘어올 경우, NPE를 던지기 때문에 **주의해서 사용해야 합니다**.

```java
Optional<Member> maybeMember = Optional.of(aMember);
```

- `Optional.ofNullable(value)`

null인지 아닌지 확신할 수 없는 객체를 담고 있는 Optional 객체를 생성합니다.
`Optional.empty()` 와 `Optional.ofNullable(value)` 를 합쳐놓은 메소드라고 생각하시면 됩니다.
null이 넘어올 경우, NPE를 던지지 않고 `Optional.empty()` 와 동일하게 비어 있는 Optional 객체를 얻어옵니다.
해당 객체가 null인지 아닌지 자신이 없는 상황에서는 이 메소드를 사용하셔야 합니다.

```java
Optional<Member> maybeMember = Optional.ofNullable(aMember);
Optional<Member> maybeNotMember = Optional.ofNullable(null);
```

Optional이 담고 있는 객체 접근하기

Optional 클래스는 담고 있는 객체를 꺼내오기 위해서 다양한 인스턴스 메소드를 제공합니다.
아래 메소드들은 모두 Optional이 담고 있는 객체가 존재할 경우 동일하게 해당 값을 반환합니다.
반면에 Optional이 비어있는 경우(즉, null을 담고 있는 경우), 다르게 작동합니다.
따라서 비어있는 Optional에 대해서 다르게 작동하는 부분만 설명드리겠습니다.

- `get()`

비어있는 Optional 객체에 대해서, `NoSuchElementException` 을 던집니다.

- `orElse(T other)`

비어있는 Optional 객체에 대해서, 넘어온 인자를 반환합니다.

- `orElseGet(Supplier<? extends T> other)`

비어있는 Optional 객체에 대해서, 넘어온 함수형 인자를 통해 생성된 객체를 반환합니다.
`orElse(T other)` 의 게으른 버전이라고 보시면 됩니다.
비어있는 경우에만 함수가 호출되기 때문에 `orElse(T other)` 대비 성능상 이점을 기대할 수 있습니다.

- `orElseThrow(Supplier<? extends X> exceptionSupplier)`

비어있는 Optional 객체에 대해서, 넘어온 함수형 인자를 통해 생성된 예외를 던집니다.

지금까지 Optional에서 제공하는 주요 메소드들에 대해서 퀵하게 알아보았습니다.
이제부터 이 메소드들을 어떻게 활용하는지에 대해서 얘기해보도록 하겠습니다.

Optional의 잘못된 사용

위에서 설명드린 것 처럼 `get()` 메소드는 비어있는 Optional 객체를 대상으로 호출할 경우, 예외를 발생시키므로 다음과 같이 객재 존재 여부를 bool 타입으로 반환하는 `isPresent()` 라는 메소드를 통해 null 체크가 필요합니다.

```java
String text = getText();
Optional<String> maybeText = Optional.ofNullable(text);
int length;
if (maybeText.isPresent()) {
        length = maybeText.get().length();
} else {
        length = 0;
}
```

같은 코드를 다시 Optional 없이 작성해보겠습니다.

```java
String text = getText();
int length;
if (text != null) {
        length = maybeText.get().length();
} else {
        length = 0;
}
```

위 코드를 보시고 이렇게 말씀하시는 분들이 많으실 겁니다. "이럴꺼면 뭐하러 Optional을 사용하는 걸까요? Optional을 사용해서 도대체 뭐가 좋아진거죠?
사실 이렇게 코딩하실 거라면 차라리 Optional을 사용하지 않는 편이 나을 것 같습니다."

안타깝게도 Optional 관련해서 개발자들이 제일 많이 하는 질문 중 하나가 "Optional 적용 후 어떻게 null 체크를 해야하나요?" 입니다.
사실 이 질문에 대한 답변은 "null 체크를 하실 필요가 없으시니 하시면 안 됩니다." 입니다.

제 말이 무슨 말인지 혼란스러우신 분도 있으실 겁니다. :dizzy_face: 저도 처음 Optional을 접했을 때 그랬으니까요.
우리가 Optional을 사용하려는 이유는 앞에서 설명드렸던 것 처럼 **고통스러운 null 처리를 직접하지 않고 Optional 클래스에 위임하기** 위함입니다.

따라서 위와 같은 방식으로 Optional을 사용하게 되면 Java8 이 전에 직접 null 체크를 하던 코딩 수준에서 크게 벗어나지 못하게 됩니다.

다른 잘못된 예제로 이 전 포스트에서 보았던 `getCityOfMemberFromOrder()` 메소드를 같은 스타일로 작성하면 다음과 같습니다.
이 전 포스트에서 보았던 코드와 별반 다르지 않은 수준의, 사실 오히려 살짝 더 복잡해 보이는 끔찍한 코드가 탄생하였습니다.
무엇이 어디서부터 어떻게 잘못된 걸까요? :sob:

```java
/* 주문을 한 회원이 살고 있는 도시를 반환한다 */
public String getCityOfMemberFromOrder(Order order) {
        Optional<Order> maybeOrder = Optional.ofNullable(order);
        if (maybeOrder.isPresent()) {
                Optional<Member> maybeMember = Optional.ofNullable(maybeOrder.g
                if (maybeMember.isPresent()) {
                        Optional<Address> maybeAddress = Optional.ofNullable(ma
                        if (maybeAddress.isPresent()) {
                                Address address = maybeAddress.get();
                                Optinal<String> maybeCity = Optional.ofNullable
                                if (maybeCity.isPresent()) {
                                        return maybeCity.get();
                                }
                        }
                }
        }
        return "Seoul";
}
```

Optional을 정확히 이해하고 제대로 사용하실 수 있는 개발자라면 첫번째 예제의 코드는 다음과 같이 한 줄의 코드로 작성할 수 있어야 합니다.
다시 말해서, 기존에 조건문으로 null을 대하던 생각을 **함수형 사고**로 완전히 새롭게 바꿔야 합니다.

```java
int length = Optional.ofNullable(getText()).map(String::length).orElse(0);
```

아직은 코드가 어떻게 작동하는 건지 이해가 되지 않으실 수도 있지만 괜찮습니다.
다음 포스트에서 Optional을 좀 더 Optional답게 사용하는 방법에 대해서 다뤄보도록 하겠습니다.

자바8 Optional 3부: Optional을 Optional답게

2017-01-15 #FP, #JAVA8, #OPTIONAL, #STREAM

> *Optional을 좀 더 Java8 API 설계자의 의도에 맞게 쓰는 방법에 대해서 알아봅시다.*

이전 포스트를 통해서 Optional에 대해서 소개드렸습니다.
또한 Optional을 Optional답지 않게 사용할 경우, 어떤 부작용이 발생하는지도 살펴보았는데요.
이 번 포스트에서는 어떻게 코드를 작성해야 Optional을 Optional답게 쓸 수 있는지 알아보도록 하겠습니다.

Stream처럼 사용하기

Optional을 제대로 사용하려면, Optional을 **최대 1개의 원소를 가지고 있는** 특별한 Stream이라고 생각하시면 좋습니다.
Optional 클래스와 Stream 클래스 간에 직접적인 구현이나 상속관계는 없지만 사용 방법이나 기본 사상이 매우 유사하기 때문입니다.
Stream 클래스가 가지고 있는 `map()` 이나 `flatMap()` , `filter()` 와 같은 메소드를 Optional도 가지고 있습니다.
따라서 Stream을 능숙하게 다루시는 분이시라면 Optional도 어렵지 않게 다루실 수 있으실 겁니다.

`map()` 으로 변신하기

그럼 `Stream` API를 나루듯이 `Optional` API를 사용하여 첫 번째 포스트의 `getCityOfMemberFromOrder()` 메소드를 리펙토링 해보겠습니다.

```java
/* 주문을 한 회원이 살고 있는 도시를 반환한다 */
public String getCityOfMemberFromOrder(Order order) {
        return Optional.ofNullable(order)
                        .map(Order::getMember)
                        .map(Member::getAddress)
                        .map(Address::getCity)
                        .orElse("Seoul");
}
```

첫 번째 포스트에서 다루었던 2가지 전통적인 NPE 방어 패턴에 비해 훨씬 간결하고 명확해진 코드를 볼 수 있습니다.
우선 기존에 존재하던 조건문들이 모두 사라지고 Optional의 수려한(fluent) API에 의해서 단순한 메소드 체이닝으로 모두 대체되었습니다.
메소드 체이닝의 간 단계 별로 좀 더 상세히 짚어보겠습니다.

- `ofNullable()` 정적 팩토리 메소드를 호출하여 Order 객체를 Optional로 감싸주었습니다. 혹시 Order 객체가 null인 경우를 대비하여 `of()` 대신에 `ofNullable()` 을 사용했습니다.
- 3번의 `map()` 메소드의 연쇄 호출을 통해서 Optional 객체를 3번 변환하였습니다. 매 번 다른 메소드 레퍼런스를 인자로 넘겨서 Optional에 담긴 객체의 타입을 바꿔주었습니다. (`Optional<Order>` -> `Optional<Member>` -> `Optional<Address>` -> `Optional<String>`)
- 마무리 작업으로 `orElse()` 메소드를 호출하여 이 전 과정을 통해 얻은 Optional이 비어있을 경우, 디폴트로 사용할 도시 이름을 세팅해주고 있습니다.

어떠신가요? Optional를 제대로 활용하여 처음으로 null-safe한 코드를 작성해보았습니다. 이전에 Stream을 사용하여 이런 식으로 코딩을 해보신 적이 없으시다면 많이 낯설 수도 있습니다. Java8의 람다식과 메소드 레퍼런스를 좋아하신다면 이 코드가 마음에 드실 겁니다. :smile:

`filter()` 로 레벨업

Java8 이 전에 NPE 방지를 위해서 다음과 같이 null 체크로 시작하는 if 조건문 패턴을 자주 보셨을 겁니다.

```java
if (obj != null && obj.do() ...)
```

예를 들어, 주어진 시간(분) 내에 생성된 주문을 한 경우에만 해당 회원 정보를 구하는 메소드를 위 패턴으로 작성해보았습니다.

```java
public Member getMemberIfOrderWithin(Order order, int min) {
        if (order != null && order.getDate().getTime() > System.currentTimeMilli
                return order.getMember();
        }
}
```

위 코드는 if 조건문 내에 null 체크와 비지니스 로직이 혼재되어 있어서 가독성이 떨어집니다.
게다가 null을 리턴할 수 있기 때문에 메소드 호출부에 NPE 위험을 전파하고 있습니다.

반면에 `filter()` 메소드를 사용하면 if 조건문 없이 메소드 연쇄 호출만으로도 좀 더 읽기 편한 코드를 작성할 수 있습니다.
뿐만 아니라, 메소드의 리턴 타입을 Optional로 사용함으로써 호출자에게 해당 메소드가 null을 담고 있는 Optional을 반환할 수도 있다는 것을 명시적으로 알려주고 있습니다.

```java
public Optional<Member> getMemberIfOrderWithin(Order order, int min) {
        return Optional.ofNullable(order)
                        .filter(o -> o.getDate().getTime() > System.currentTimeM
                        .map(Order::getMember);
}
```

`filter()` 메소드는 넘어온 함수형 인자의 리턴 값이 `false` 인 경우, Optional을 비워버리므로 그 이후 메소드 호출은 의미가 없어지게 됩니다. `Stream` 클래스의 `filter()` 메소드와 동작 방식이 동일하지만, `Optional` 의 경우 원소가 하나 밖에 없기 때문에 이런 효과가 나타나게 됩니다.

Java8 이 전에 개발된 코드를 Optional하게 바꾸기

Java8 이 전에 개발된 API들은 안타깝게도 당시에 `Optional` 클래스가 없었기 때문에 null-safe하지 않습니다.
심지어 Java 표준 API 조차 하위 호환성을 보장을 위해서 기존 API에 `Optional` 클래스를 적용할 수 없었습니다.
다행이도 우리는 스스로 `Optional` 클래스를 사용하여 기존 코드가 null-safe하도록 바꿔줄 수 있습니다.

메소드의 반환값이 존재하지 않을 때 전통적인 처리 패턴

이전 개발된 메소드들은 반환할 값이 존재하지 않을 경우, 크게 2가지 패턴으로 처리하였습니다.
각 처리 패턴을 어떻게 개선할 수 있는지 예제 코드를 통해 살펴보겠습니다.

1. null 반환

`Map` 인터페이스의 `get()` 메소드는 주어진 인덱스에 해당하는 값이 없으면 null을 반환합니다.

```java
Map<Integer, String> cities = new HashMap<>();
cities.put(1, "Seoul");
cities.put(2, "Busan");
cities.put(3, "Daejeon");
```

따라서 해당 API를 사용하는 코드를 null-safe하게 만들기 위해서 null 체크를 해줘야 합니다.

```java
String city = cities.get(4); // returns null
int length = city == null ? 0 : city.length(); // null check
System.out.println(length);
```

다음과 같이 `get()` 메소드의 반환 값을 Optional로 감싸주면, 자연스럽게 null-safe한 코드가 됩니다.

```java
Optional<String> maybeCity = Optional.ofNullable(cities.get(4)); // Optional
int length = maybeCity.map(String::length).orElse(0); // null-safe
System.out.println("length: " + length);
```

읽기 쉬운 코드를 작성하기 위해 `map()` 과 `orElse()` 메소드가 어떻게 사용되고 있는지 주의깊게 보시기 바랍니다.
도시 문자열을 길이로 변환하고 디폴트 값을 설정해주는 과정을 한눈에 파악하기 편하지 않으신가요? :smile:

2. 예외 발생

두번째 패턴은 null을 반환하지 않고 예외를 던저버리는 경우입니다.
`List` 인터페이스의 `get()` 메소드는 주어진 인덱스에 해당하는 값이 없으면 ArrayIndexOutOfBoundsException을 던집니다.

```java
List<String> cities = Arrays.asList("Seoul", "Busan", "Daejeon");
```

따라서, 다음과 같이 try-catch 구문을 사용하여 예외 처리를 해줘야 하며, 예외 처리 후에도 null check도 해줘야 하기 때문에 코드가 지저분해집니다.

```java
String city = null;
try {
        city = cities.get(3); // throws exception
} catch (ArrayIndexOutOfBoundsException e) {
        // ignore
}
int length = city == null ? 0 : city.length(); // null check
System.out.println(length);
```

이런 경우, 다음과 같이 예외 처리부를 감싸서 정적 유틸리티 메소드로 분리해주면 좋습니다.
`Optional` 클래스의 정적 팩토리 메소드를 사용하여 정상 처리 시와 예외 처리 시에 반환할 Optional 객체를 각각 지정해주었습니다.
이 경우에는 Optional에 담을 객체가 null인지 아닌지 확실히 알 수 있기 때문에 `Optional.ofNullable()` 대신에 다른 2개의 정적 팩토리 메소드를 쓸 수 있다는 점을 주의깊게 보시기 바랍니다.

```java
public static <T> Optional<T> getAsOptional(List<T> list, int index) {
        try {
                return Optional.of(list.get(index));
        } catch (ArrayIndexOutOfBoundsException e) {
                return Optional.empty();
        }
}
```

아래와 같이 정적 유틸리티 메소드를 통해 Optional 객체를 확보 후에 null-safe하게 코딩할 수 있습니다.

```java
Optional<String> maybeCity = getAsOptional(cities, 3); // Optional
int length = maybeCity.map(String::length).orElse(0); // null-safe
System.out.println("length: " + length);
```

`ifPresent()` 메소드

아, 이 전 포스트에서 미처 소개드리지 않은 조금 특별한 메소드가 하나 있습니다.
바로 `ifPresent()` 메소드인데요. 많은 분들이 `isPresent()` 와 혼동하기도 하는 녀석입니다.

- `ifPresent(Consumer<? super T> consumer)` : 이 메소드는 특정 결과를 반환하는 대신에 Optional 객체가 감싸고 있는 값이 존재할 경우에만 실행될 로직을 함수형 인자로 넘길 수 있습니다.

함수형 인자로 람다식이나 메소드 레퍼런스가 넘어올 수 있는데요. 마치 비동기 메소드의 콜백 함수처럼 작동합니다.
바로 전 예제의 코드를 `ifPresent()` 메소드를 이용해서 재작성하면 다음과 같습니다.

```java
Optional<String> maybeCity = getAsOptional(cities, 3); // Optional
maybeCity.ifPresent(city -> {
        System.out.println("length: " + city.length());
});
```

마치면서

총 3부의 포스트를 통해서 Java8의 Optional에 대해서 탐구해보았습니다.
이 포스트를 통해 더 많은 자바 개발자들이 Optional을 잘 쓰셔서 null 체크의 스트레스에서 벗어나셨으면 좋겠습니다.