32-Java 8: Optional

Author: Vincent Lau

Learning Objectives

> Identify issues with *null* as a return value

> Write code that uses *Optional* and its common methods

> Understand when NOT to use *Optionals*

Overview

What is *Optionals*?

- Introduced in **Java 8**. Provides a type-level solution for representing optional values instead of using *null* references

- Similar to checking whether a collection is empty, *Optional* allows us to check whether a single object is empty.

Why *Optionals*?

- *NullPointerException* is the most common type of exceptions thrown in Java code.
- *null* can be assigned to any reference variables.
- Methods cannot communicate that they might return *null* except in documentation.
- Before Java 8 (The world without *Optional*), we had to either catch *NullPointerException* explicitly or perform null checks all over the place - both are unsustainable and error-prone.

How does *Optional* work?

- Represents a value that *might* be present.
- Uses generics - `Optional<T>` can represent any object type.
- An ***Optional*** communicates that the method **might not return a value** - it **forces you to check and extract an object from *Optional* before performing operations on the object, thus avoiding *NullPointerException***.

Creating Optionals

- The *Optionals* class provides three *static factory methods* to construct an *Optional*:
  - *Optional.empty()*
  - *Optional.of()*
  - *Optional.ofNullable()*

```
// Approach #1
public static<T> Optional<T> empty() {
                // returns an empty Optional instance which contains no value
}

// Approach #2
public static <T> Optional<T> of(T value) {
                // returns an Optional which contains a non-null value
                // throws NullPointerException if value is null
}

// Approach #3
public static <T> Optional<T> ofNullable(T value) {
    // returns an Optional which contains the given value if non-null
                // returns an empty optional if value is null
}
```

Example

```
Optional<String> empty = Optional.empty();
System.out.println(empty.isPresent());  // prints false

String name = "John Doe";
Optional<String> opt = Optional.of(name);
System.out.println(opt.isPresent());  // prints true

String name = null;
Optional.of(name);  // runtime exception, throws NullPointerException

String name = "John Doe";
Optional<String> opt = Optional.ofNullable(name);
System.out.println(opt.isPresent());  // prints true

String name = null;
Optional<String> opt = Optional.ofNullable(name);
System.out.println(opt.isPresent());  // prints false
```

orElse(), orElseGet(), orElseThrow() and *ifPresent()*

- The ***Optionals*** class also provides ways for us to define a *default* value, action or exception to throw in case the ***Optional*** is empty by chaining methods.
  - ***orElse()*** - returns a default value if no value is present
  - ***orElseGet()*** - invokes a *Supplier* implementation to return a default value
  - ***orElseThrow()*** - invokes an *Exception Supplier* implementation to throw an exception
  - ***ifPresent()*** - invokes a *Consumer* implementation to perform some action on the given value, but without returning anything

Example

```
// given a *non-null* value
Optional<String> opt1 = Optional.ofNullable("John");
```

```java

String defaultWithOrElse1 = opt1.orElse("default value");
System.out.println(defaultWithOrElse1);  // prints "John"

// orElseGet takes a Supplier implementation
String defaultWithOrElseGet1 = opt1.orElseGet(() -> "default value");
System.out.println(defaultWithOrElseGet1);  // prints "John"

// orElseThrow takes an Exception Supplier implementation
String defaultWithOrElseThrow1 = opt1.orElseThrow(() -> new SomeException());
System.out.println(defaultWithOrElseThrow1);  // prints "John"

// ifPresent() takes a Consumer implementation
opt1.ifPresent(value -> System.out.println(value));  // prints "John"

// given a null value
Optional<String> opt2 = Optional.ofNullable(null);

String defaultWithOrElse2 = opt2.orElse("default value");
System.out.println(defaultWithOrElse2);  // prints "default value"

// *orElseGet* takes a *Supplier* implementation
String defaultWithOrElseGet2 = opt2.orElseGet(() -> "default value");
System.out.println(defaultWithOrElseGet2);  // prints "default value"

// *orElseThrow* takes an *Exception Supplier* implementation
String defaultWithOrElseThrow2 = opt2.orElseThrow(() -> new SomeException());
System.out.println(defaultWithOrElseThrow2);  // throws SomeException

// *ifPresent()* takes a *Consumer* implementation
opt2.ifPresent(value -> System.out.println(value));  // prints nothing
```

When NOT to Use *Optionals*

- `Optionals` should **ONLY** be used as a return value of a method.
- It is NOT recommended to use *Optionals* as object fields especially when the object is *Serializable*, because ***Optionals*** is currently NOT *Serializable*.
    - Unexpected results may occur when you attempt to serialize *Optionals*.
    - Thus, it is also NOT recommended to use *Optionals* when **an object needs to be converted to JSON** or be **mapped to JPA entities**.

Example 1

Problem - Optional as a Type in POJO

```java
@Entity
public class UserOptionalField implements Serializable {
    @Id
    private long userId;

    private Optional<String> firstName;

    // ... getters and setters
}

// main logic
UserOptionalField user = new UserOptionalField();
user.setUserId(1l);
user.setFirstName(Optional.of("Baeldung")); // its ok
entityManager.persist(user); // Error Occur
```

Error Occur

```
Caused by: javax.persistence.PersistenceException: [PersistenceUnit: com.baeldung.optionalReturnType] Unable to build Hibernate SessionFactory
        at org.hibernate.jpa.boot.internal.EntityManagerFactoryBuilderImpl.persistenceException(EntityManagerFactoryBuilderImpl.java:1015)
        at org.hibernate.jpa.boot.internal.EntityManagerFactoryBuilderImpl.build(EntityManagerFactoryBuilderImpl.java:941)
        at org.hibernate.jpa.HibernatePersistenceProvider.createEntityManagerFactory(HibernatePersistenceProvider.java:56)
        at javax.persistence.Persistence.createEntityManagerFactory(Persistence.java:79)
        at javax.persistence.Persistence.createEntityManagerFactory(Persistence.java:54)
        at com.baeldung.optionalReturnType.PersistOptionalTypeExample.<clinit>(PersistOptionalTypeExample.java:11)
Caused by: org.hibernate.MappingException: Could not determine type for: java.util.Optional, at table: UserOptionalField, for columns: [org.hibernate.mapping.Column(firstName)]
```

Solution

- Keep the class field nullable, but we can provide a method to return the nullable value as Optional data type. It's a good practice.

```
@Column(nullable = true)
private String firstName;

public Optional<String> getFirstName() {
    return Optional.ofNullable(firstName);
}
```

Example 2

Problem - Optional as method input parameter

- A method with an input parameter of Optional Type

```
public static List<Person> search(List<Person> people,
                                  String name, Optional<Integer> age) {
    // Null checks for people and name
    return people.stream()
            .filter(p -> p.getName().equals(name))
            .filter(p -> p.getAge().get() >= age.orElse(0)) // Null Pointer Exception
            .collect(Collectors.toList());
}
```

- What if .... A developer calls the method by putting null as parameter. *NullPointerException will be thrown out during runtime.*

```
// this call will cause search() NullPointerException
someObject.search(people, "Peter", null);
```

Solution

```
public static List<Person> search(List<Person> people,
                                                 String name, Integer age) {

    final Integer ageFilter = age != null ? age : 0;

    return people.stream()
            .filter(p -> p.getName().equals(name))
            .filter(p -> p.getAge().get() >= ageFilter)
            .collect(Collectors.toList());
}
```

Real Example - enum

```
public enum OrderStatus {
  CONFIRMED(1, "Ordered"), //
  PAID(2, "Paid"), //
  READY_TO_SHIP(3, "Ready To Ship"), //
  DELIVERED(4, "Delivered"), //
  UNKNOWN(99, "Unknown"),
  ;

  private final int code;
  private final String value;

  private OrderStatus(int code, String value) {
    this.code = code;
    this.value = value;
  }

  public int getCode() {
    return this.code;
  }

  public String getValue() {
    return this.value;
  }

  /**
   * @return the Enum representation for the given string.
   * @throws IllegalArgumentException
   *            if unknown string.
   */
  public OrderStatus fromCode(int code) {
    return Arrays.stream(OrderStatus.values()) //
        .filter(e -> e.getCode() == code) // return Stream<OrderStatus>
        .findFirst() // Optional<Orderstatus>
        // .orElse(OrderStatus.UNKNOWN);
        .orElseThrow(() -> new IllegalArgumentException()); // runtime exception
  }
}
```

Questions

- Why should we use *Optionals?* What are the problems with returning *null* as a value?
- Under what circumstances should we NOT use *Optionals*? *Why?*
- Write a program that returns *Optionals* instead of *null* when values are not present.