DZone

Functional Programming in Java 8 (Part 2): Optionals

by Niklas Wuensche MVB · Feb. 08, 17 · Java Zone

Learn how to troubleshoot and diagnose some of the most common performance issues in Java today. Brought to you in partnership with AppDynamics.

Hello everybody,

After we've made our first big steps into functional programming in the last post, we will talk about Optionals in today's part.

Why do we need Optionals?

Hand on heart, you also think that null is annoying, don't you? For every argument which can be null, you have to check whether it is null or not.

```
1  if(argument == null) {
2      throw new NullPointerException();
3  }
```

These lines suck! This is a boilerplate that bloats up your code and can be easily forgotten. So how can we fix that?

Introducing Optionals

In Java 8, java.util.Optional<T> was introduced to handle objects which might not exist better. It is a container object that can hold another object. The Generic *T* is the type of the object you want to contain.

```
1  Integer i = 5;
2  Optional<Integer> optinalI = Optional.of(i);
```

The Optional class doesn't have any public constructor. To create an optional, you have to use `Optional.of(object)` or `Optional.ofNullable(object)`. You use the first one if the object is never, ever null. The second one is used for nullable objects.

How do optionals work?

Options have two states. They either hold an object or they hold null. If they hold an object, optionals are called *present*, if they hold null, they are called *empty*. If they are not empty, you can get the object in the optional by using `Optional.get()`. But be careful, because a *get()* on an empty optional will

in the optional by using `Optional.get()`. But be careful, because a *get()* on an empty optional will cause a `NoSuchElementException`. You can check if a optional is present by calling the method `Optional.isPresent()`

Example: Playing with Optionals

```
public void playingWithOptionals() {
    String s = "Hello World!";
    String nullString = null;

    Optional<String> optionalS1 = Optional.of(s); // Will work
    Optional<String> optionalS2 = Optional.ofNullable(s); // Will work too
    Optional<String> optionalNull1 = Optional.of(nullString); // -> NullPointerException

    Optional<String> optionalNull2 = Optional.ofNullable(nullString); // Will work

    System.out.println(optionalS1.get()); // prints "Hello World!"
    System.out.println(optionalNull2.get()); // -> NoSuchElementException
    if(!optionalNull2.isPresent()) {
        System.out.println("Is empty"); // Will be printed
    }
}
```

Common problems when using Optionals

1. Working with Optional **and** null

```
public void workWithFirstStringInDB() {
    DBConnection dB = new DBConnection();
    Optional<String> first = dB.getFirstString();

    if(first != null) {
        String value = first.get();
        //...
    }
}
```

This is just the wrong use of an Optional! If you get an Optional (In the example you get one from the DB), you don't have to look whether the object is null or not! If there's no string in the DB, it will return `Optional.empty()`, not `null`! If you got an empty Optional from the DB, there would also be a `NoSuchElementException` in this example.

2. Using isPresent() and get()

2.1. Doing something when the value is present

```
public void workWithFirstStringInDB() {
    DBConnection dB = new DBConnection();
    Optional<String> first = dB.getFirstString();
```

```
4
5       if(first.isPresent()) {
6           String value = first.get();
7           //...
8       }
9   }
```

As I already said, you should always be 100% sure if an Optional is present before you use `Optional.get()`. You wouldn't get a `NoSuchElementException` anymore in the updated function. But you shouldn't check a optional with the `isPresent() + get()` combo! Because if you do it like that, you could have used *null* in the first place. You would replace `first.isPresent()` with `first != null` and you have the same result. But how can we replace this annoying if-block with the help of Optionals?

```
1   public void workWithFirstStringInDB() {
2       DBConnection dB = new DBConnection();
3       Optional<String> first = dB.getFirstString();
4
5       first.ifPresent(value -> /*...*/);
6   }
```

The `Optional.ifPresent()` method is our new best friend to replace the if. It takes a Function, so a Lambda or method reference, and applies it only when the value is present. If you don't remember how to use method references or Lambdas, you should read the last part of this series again.

2.2. Returning a Modified Version of the Value

```
1   public Integer doubleValueOrZero(Optional<Integer> value) {
2       if(value.isPresent()) {
3          return value.get() * 2;
4       }
5
6       return 0;
7   }
```

In this method, we want to double the value of an optional, if it is present. Otherwise, we return zero. The given example works, but it isn't the functional way of solving this problem. To make this a lot nicer, we have two function that are coming to help us. The first one's `Optional.map(Function<T, R> mapper)` and the second one's `Optional.orElse(T other)`. `map` takes a function, applies it on the value and returns the result of the function, wrapped in an Optional again. If the Optional is empty, it will return an empty Optional again. `orElse` returns the value of the Optional it is called on. If there is no value, it returns the value you gave `orElse(object)` as a parameter. With that in mind, we can make this function a one liner.

```
1   public Integer doubleValueOrZero(Optional<Integer> value) {
2       return value.map(i -> i * 2).orElse(0);
3   }
```

When Should you use Nullable Objects and when

Optionals

You can find a lot of books, talks and discussions about the question: Should you use null or Optional in some particular case. And both have their right to be used. In the linked talk, you will find a nice rule which you can apply in most of the cases. Use Optionals when *"there is a clear need to represent 'no result' or where null is likely to cause errors."*.

So you shouldn't use Optionals like this:

```
1  public String defaultIfOptional(String string) {
2      return Optional.ofNullable(string).orElse("default");
3  }
```

Because a null check is much easier to read.

```
1  public String defaultIfOptional(String string) {
2      return (string != null) ? string : "default";
3  }
```

You should use Optionals just as a return value from a function. It's not a good idea to create new ones to make a cool method chain like in the example above. Most of the times, null is enough.

Conclusion

That's it for today! We have played with the Optional class. It's a container class for other classes which is either present or not present(*empty*). We have removed some common code smell that comes with Optionals and used functions as objects again. We also discussed when you should use null and when Optionals.

In the next part, we will use *Streams* as a new way to handle a Collection of Objects, like Iterables and Collections.

Thanks for reading and have a nice day,

Niklas

Topics: JAVA , FUNCTIONAL PROGRAMMING , OPTIONALS , NULL CHECKS , TUTORIAL