12 recipes for using the Optional class as it’s meant to be used

Follow these dozen best practices to protect your applications against ugly null pointer exceptions—and make your code more readable and concise.

by Mohamed Taman

June 22, 2020

Every serious Java developer or architect has heard about or experienced the nuisance of NullPointerException exceptions.

What can you do? Often programmers use the null reference to denote the absence of a value when they return values from methods, but that is a significant source of many problems.

To have good insight into the null reference problem, consider reading Raoul-Gabriel Urma’s article, “Tired of Null Pointer Exceptions? Consider Using Java SE 8’s ‘Optional’!” That’ll bring you up to speed and introduce you to the Optional class.

Let’s build on Urma’s work by seeing how to use Optional the way it should be used. From the experience and hands-on point of view I gained when I was reviewing developers’ code, I realized developers are using the Optional class in their day-to-day code. That led me to come up with these 12 best practices that will help you improve your skills—and avoid antipatterns.

This article and a follow-up article that will appear in Java Magazine soon will go through all the Optional class methods released through Java 14. Since Java 15 is nearing completion, that will be covered too.

The origin of the Optional class
Java engineers have been working on the null problem for a long time and they tried to ship a solution with Java 7, but that solution wasn't added to the release. Let's think back, though, and imagine the language designers' thoughts about the Stream API. There is a logical return type (such as 0) to some methods such as count() and sum() when the value is absent. A zero makes sense there.

But what about the findFirst() and findAny() methods? It doesn't make sense to return null values if there aren't any inputs. There should be a value type that represents the presence (or absence) of those inputs.

Therefore, in Java 8, a new type was added called Optional<T>, which indicates the presence or absence of a value of type T. Optional was intended to be a return type and for use when it is combined with streams (or methods that return Optional) to build fluent APIs. Additionally, it was intended to help developers deal with null references properly.

By the way, here is how Optional is described in the Java SE 11 documentation: "Optional is primarily intended for use as a method return type where there is a clear need to represent 'no result,' and where using null is likely to cause errors. A variable whose type is Optional should never itself be null; it should always point to an Optional instance."

My own definition, as you are going to see in my code recipes, is this: The Optional class is a container type for a value that may be absent.

Moreover, several corner cases and temptations can be considered traps that could downgrade the quality of your code or even cause unexpected behaviors. I am going to show those in this series of articles.

You might now wonder, "Where is the code?" So, let's jump in. My approach is to answer typical developer questions that categorize all the uses of the Optional class. In this article, I will cover the following three big categories with 12 recipes:

- Why am I getting null even when I use Optional?
- What should I return or set when no value is present?
- How do I consume Optional values effectively?

In an upcoming article, I will cover two more categories:

- How do I avoid Optional antipatterns?
- I like Optional; what can I do more professionally?

Why am I getting null even when I use Optional?

Usually, this question applies to the creation of an Optional class and how to get the data.
Recipe 1: Never assign null to an optional variable.
Sometimes when developers are dealing with a database to search for an employee, they design a method to return Optional<Employee>. I have found developers still returning null if no result is returned from the database, for example:

```java
public Optional<Employee> getEmployee(int i
    // perform a search for employee
    Optional<Employee> employee = null; // i
    return employee;
}
```

The code above is not correct, and you should avoid it completely. To correct it, you should replace line 3 with the following line, which initializes Optional with an empty Optional:

```java
Optional<Employee> employee = Optional.empty();
```

Optional is a container that may hold a value, and it is useless to initialize it with null.

**API note:** The empty() method has existed since Java 8.

Recipe 2: Don’t call get() directly. Consider the following piece of code. What is wrong with it?

```java
Optional<Employee> employee = HRService.getEm
Employee myEmployee = employee.get();
```

Did you guess that the “employee” Optional is prone to being empty, so calling get() directly throws a java.util.NoSuchElementException? If so, you are right. Otherwise, if you think calling get() is going to make your day, you are mistaken. You should always check first for the presence of a value by using the isPresent() method, as in the following:

```java
if (employee.isPresent()) {
    Employee myEmployee = employee.get();
    ... // do something with "myEmployee"
} else {
    ... // do something that doesn’t call emp
}
```

Note that the code above is boilerplate and is not preferable. Next, you are going to see a lot of elegant alternatives to calling isPresent()/get() pairs.

**API note:** The isPresent() and get() methods have existed since Java 8.
Recipe 3: Don’t use null directly to get a null reference when you have an Optional. In some cases, you need to have a null reference. If you have an Optional, don’t use null directly. Instead, use `orElse(null)`.

Consider the following example of calling the Reflection API’s `invoke()` method of the `Method` class. It invokes the method at runtime. The first argument is `null` if the called method is `static`; otherwise, it passes the method containing the class instance.

```java
public void callDynamicMethod(MyClass clazz) {
    Optional<MyClass> myClass = clazz.getIns;
    Method = MyClass.class.getDeclaredMethod
    if (myClass.isPresent()) {
        method.invoke(myClass.get(), "Test")
    } else {
        method.invoke(null, "Test");
    }
}
```

Generally, you should avoid using `orElse(null)`, although in such a case, using `orElse(null)` is preferable to using the code above. So, you can replace lines 4 through 8 with the following concise line of code:

```java
4    method.invoke(myClass.orElse(null), "Test")
```

**API note:** The `orElse()` method has existed since Java 8.

**What should I return or set when no value is present?**

The previous section covered how to avoid null reference problems even when you have Optional. Now it’s time to explore different ways of setting and returning data using Optional.

**Recipe 4: Avoid using an isPresent() and get() pair for setting and returning a value.** Consider the following code. What can you change to make it more elegant and effective?

```java
public static final String DEFAULT_STATUS = "...
public String getEmployeeStatus(long id) {
    Optional<String> empStatus = ...;
    if (empStatus.isPresent()) {
        return empStatus.get();
    } else {
        return DEFAULT_STATUS;
    }
}
```
Similar to recipe #3, just replace the `isPresent()` and `get()` pair with `orElse()`, as in the following:

```java
  public String getEmployeeStatus(long id) {
      Optional<String> empStatus = ...;
      return empStatus.orElse(DEFAULT_STATUS);
  }
```

A very important note to consider here is a probable performance penalty: The value returned by `orElse()` is always evaluated regardless of the optional value’s presence. So the rule here is to use `orElse()` when you have already preconstructed values and you don’t use an expensive computed value.

**API note:** The `orElse()` method has existed since Java 8.

**Recipe 5: Don’t use `orElse()` for returning a computed value.**

As mentioned in recipe #4, avoid using `orElse()` to return a computed value because there is a performance penalty. Consider the following code snippet:

```java
  Optional<Employee> getFromCache(int id) {
      System.out.println("search in cache with Id: "+id);
      // get value from cache
  }

  Optional<Employee> getFromDB(int id) {
      System.out.println("search in Database with Id: "+id);
      // get value from database
  }

  public Employee findEmployee(int id) {
      return getFromCache(id)
          .orElse(getFromDB(id)
                      .orElseThrow(() -> new NotFoundException));
  }
```

First, the code tries to get an employee with a given ID from the cache, and if it is not in the cache, it tries to get it from the database. Then, if the employee is not in the cache or the database, the code throws a `NotFoundException`. If you run this code and the employee is in the cache, the following is printed:

```
Search in cache with Id: 1
Search in Database with Id: 1
```

Even though the employee will be returned from the cache, the database query is still called. That is very expensive, right? Instead, I will use `orElseGet(Supplier<? extends T> supplier)`, which is like `orElse()` but with one difference: If the `Optional` is empty, `orElse()` returns a default value directly, whereas...
orElseGet() allows you to pass a Supplier function that is invoked only when the Optional is empty. That's great for performance.

Now, consider that you rerun the previous code with the same assumption that the employee is in the cache but with the following change to orElseGet():

```java
public Employee findEmployee(int id) {
    return getFromCache(id)
        .orElseGet(() -> getFromDB(id)
            .orElseThrow(() -> {
                return new NotFoundException();
            }));
}
```

This time, you will get what you want and a performance improvement: The code will print only the following:

```
Search in cache with Id: 1
```

Finally, don’t even think of using.isPresent() and get() pairs because they are not elegant.

**API note:** The orElseGet() method has existed since Java 8.

**Recipe 6: Throw an exception in the absence of a value.**

There are cases when you want to throw an exception to indicate a value doesn’t exist. Usually this happens when you develop a service interacting with a database or other resources. With Optional, it is easy to do this. Consider the following example:

```java
public Employee findEmployee(int id) {
    var employee = p.getFromDB(id);
    if(employee.isPresent())
        return employee.get();
    else
        throw new NoSuchElementException();
}
```

The code above could be avoided and the task could be expressed elegantly, as follows:

```java
public Employee findEmployee(int id) {
    return getFromDB(id).orElseThrow();
}
```

The code above will throw java.util.NoSuchElementException to the caller method.
API note: The `orElseThrow()` method has existed since Java 10. If you’re still stuck on Java 8 or 9, consider recipe #7.

**Recipe 7: How can I throw explicit exceptions when no value is present?** In recipe #6, you were limited to throwing one kind of implicit exception: `NoSuchElementException`. Such an exception is not enough to report the problem with more-descriptive and relevant information to the client.

If you recall, recipe #5 used the method

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

which is an elegant alternative to `isPresent()` and `get()` pairs. Therefore, try to avoid this:

```java
if (foundEmployee.isPresent())
    return foundEmployee.get();
else
    throw new NotFoundException("Employee");
```

The `orElseThrow()` method simply throws the explicit exception you pass to it if there is no value present with the `Optional`. So, let’s change the previous method to be more elegant, like this:

```java
@GetMapping("/employees/{id}"ements/{id}"
public Employee getEmployee(@PathVariable("id" id)
    Optional<Employee> foundEmployee = HrRepo
    if (foundEmployee.isPresent())
        return foundEmployee.get();
    else
        throw new NotFoundException("Employee");
}
```

Additionally, if you are only interested in throwing an empty exception, here it is:

```java
return status.orElseThrow(NotFoundException::
```

API note: If you pass a `null` to the `orElseThrow()` method, it will throw a `NullPointerException` if no value is present. The

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

method has existed since Java 8.

**How do I consume Optional values effectively?**
Recipe 8: Don’t use isPresent()-get() if you want to perform an action only when an Optional value is present.

Sometimes you want to perform an action only if an Optional value is present and do nothing if it is not. That is the job of the ifPresent(Consumer<? super T> action) method, which takes a consumer action as an argument. For the record, avoid the following:

```java
1 Optional<String> confName = Optional.of("ConfName");
2 if(confName.isPresent())
3    System.out.println(confName.get().length);
```

It is perfectly fine to use ifPresent(); just replace lines 2 and 3 above with one line, as follows:

```java
confName.ifPresent( s -> System.out.println(s.length));
```

**API note:** The ifPresent() method returns nothing, and it has existed since Java 8.

Recipe 9: Don’t use isPresent()-get() to execute an empty-based action if a value is not present. Developers sometimes write code that does something if an Optional value is present but executes an empty-based action if it is not, like this:

```java
1 Optional<Employee> employee = ... ;
2 if(employee.isPresent()) {
3    log.debug("Found Employee: {}", employee)
4 } else {
5    log.error("Employee not found");
6 }
```

Note that ifPresentOrElse() is like ifPresent() with the only difference being that it covers the else branch as well. Therefore, you can replace lines 2 through 6 with this:

```java
employee.ifPresentOrElse(
    emp -> log.debug("Found Employee: {}", emp.get()),
    () -> log.error("Employee not found");
```

**API note:** The ifPresentOrElse() method has existed since Java 9.

Recipe 10: Return another Optional when no value is present. In some cases, if the value of the Optional at hand is present, return an Optional describing the value; otherwise, return an Optional produced by the supplying function. Avoid doing the following:
Don’t overuse the `orElse()` or `orElseGet()` methods to accomplish this because both methods return an *unwrapped* value. So avoid doing something like this also:

```
Optional<String> defaultJobStatus = Optional.<String>empty();
```  

The perfect and elegant solution is to use the `orElse (Supplier<? extends Optional<? extends T>> supplier)` method, as follows:

```
Optional<String> foundStatus = ... ; // f
return foundStatus.orElseGet(() -> defaultJobStatus);

public Optional<String> fetchJobStatus(int jobID) {
    Optional<String> foundStatus = ... ; // f
    return foundStatus.orElseGet(() -> defaultJobStatus);
}
```

Or, even without defining the `defaultJobStatus` optional at the beginning, you could replace the code in line 3 with this:

```
return foundStatus.or(() -> Optional.of("Not found"));
```

*API note:* The `or()` throws `NullPointerException` if the supplying function is `null` or produces a `null` result. This method has existed since Java 9.

**Recipe 11: Get an Optional’s status regardless of whether it is empty.** Since Java 11, you can directly check if an `Optional` is empty by using the `isEmpty()` method, which returns `true` if the `Optional` is empty. So, instead of this code

```
public boolean isMovieListEmpty(int id){
    Optional<MovieList> movieList = ... ;
    return !movieList.isPresent();
}
```

You can replace line 3 with the following line to make the code more readable:

```
return movieList.isPresent();
```
Recipe 12: Don’t overuse Optional. Sometimes developers tend to overuse things they like and the Optional class is one of them. I have found that developers can see a use case for Optional everywhere, by chaining its methods just for the single purpose of getting a value, and they forget about clarity, memory footprint, and being straightforward. So, avoid this:

```java
public String fetchJobStatus(int jobId) {
    String status = ... ; // fetch declared
    return Optional.ofNullable(status).orElse(null);
}
```

Be straightforward by replacing line 3 with this clearer line of code:

```java
return status == null ? "Not started yet." :
```

Conclusion

Just as with any other Java language feature, Optional can be used correctly or abused. To know the best way to use the Optional class, you need to understand what has been explored in this article and keep the recipes handy to strengthen your toolkit.

I like this saying from Oliver Wendell Holmes, Sr.: “The young man knows the rules, but the old man knows the exceptions.”

I have only scratched the surface of the Optional class here. In a forthcoming article, I will dive into the Optional antipatterns I have observed from developers’ code and even my own code. I’ll then go through more-advanced recipes that deal with the Stream API, transformations, and many more cases to cover the use of all the remaining Optional class methods.

To learn more, see the Java SE 15 API documentation for Optional and see "Java 8 in Action" by Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft (Manning, 2014).
and is a Java Champion, an Oracle Groundbreaker, and a member of the Adopt-a-Spec program for Jakarta EE and Adopt-a-JSR for OpenJDK.