(/)

Guide To Java 8 Optional

Last modified: January 31, 2020

by baeldung (https://www.baeldung.com/author/baeldung/)

Java (https://www.baeldung.com/category/java/) +

Java 8 (https://www.baeldung.com/tag/java-8/)

1. Overview

In this tutorial, we're going to show the *Optional* class that was introduced in Java 8.

The purpose of the class is to provide a type-level solution for representing optional values instead of *null* references.

To get a deeper understanding of why we should care about the *Optional* class, take a look at the official Oracle's article (http://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html).

2. Creating *Optional* Objects

There are several ways of creating *Optional* objects. To create an empty *Optional* object, we simply need to use its *empty()* static method:

```
@Test
public void whenCreatesEmptyOptional_thenCorrect() {
    Optional<String> empty = Optional.empty();
    assertFalse(empty.isPresent());
}
```

Note that we used the *isPresent()* method to check if there is a value inside the *Optional* object. A value is present only if we have created *Optional* with a non-*null* value. We'll look at the *isPresent()* method in the next section.

We can also create an *Optional* object with the static method *of()*:

```
@Test
public void givenNonNull_whenCreatesNonNullable_thenCorrect() {
    String name = "baeldung";
    Optional<String> opt = Optional.of(name);
    assertTrue(opt.isPresent());
}
```

However, the argument passed to the *of()* method can't be *null*. Otherwise, we'll get a *NullPointerException*:

```
@Test(expected = NullPointerException.class)
public void givenNull_whenThrowsErrorOnCreate_thenCorrect() {
    String name = null;
    Optional.of(name);
}
```

But, in case we expect some *null* values, we can use the *ofNullable()* method:

```
@Test
public void givenNonNull_whenCreatesNullable_thenCorrect() {
    String name = "baeldung";
    Optional<String> opt = Optional.ofNullable(name);
    assertTrue(opt.isPresent());
}
```

By doing this, if we pass in a *null* reference, it doesn't throw an exception but rather returns an empty *Optional* object:

```
@Test
public void givenNull_whenCreatesNullable_thenCorrect() {
    String name = null;
    Optional<String> opt = Optional.ofNullable(name);
    assertFalse(opt.isPresent());
}
```

3. Checking Value Presence: *isPresent()* and *isEmpty()*

When we have an *Optional* object returned from a method or created by us, we can check if there is a value in it or not with the *isPresent()* method:

```
@Test
public void givenOptional_whenIsPresentWorks_thenCorrect() {
    Optional<String> opt = Optional.of("Baeldung");
    assertTrue(opt.isPresent());

    opt = Optional.ofNullable(null);
    assertFalse(opt.isPresent());
}
```

This method returns *true* if the wrapped value is not *null*.

Also, as of Java 11, we can do the opposite with the *isEmpty* method:

```
@Test
public void givenAnEmptyOptional_thenIsEmptyBehavesAsExpected() {
    Optional<String> opt = Optional.of("Baeldung");
    assertFalse(opt.isEmpty());

    opt = Optional.ofNullable(null);
    assertTrue(opt.isEmpty());
}
```

4. Conditional Action With *ifPresent()*

The *ifPresent()* method enables us to run some code on the wrapped value if it's found to be non-*null*. Before *Optional*, we'd do:

```
if(name != null) {
    System.out.println(name.length());
}
```

This code checks if the name variable is *null* or not before going ahead to execute some code on it. This approach is lengthy and that's not the only problem, it's also prone to error.

Indeed, what guaranty us that, after printing that variable, we won't use it again and then **forget to perform the null check**.

This can result in a *NullPointerException* at runtime if a null value finds its way into that code. When a program fails due to input issues, it's often a result of poor programming practices.

Optional makes us deal with nullable values explicitly as a way of enforcing good programming practices. Let us now look at how the above code could be refactored in Java 8.

In typical functional programming style, we can execute perform an action on an object that is actually present:

```
@Test
public void givenOptional_whenIfPresentWorks_thenCorrect() {
    Optional<String> opt = Optional.of("baeldung");
    opt.ifPresent(name -> System.out.println(name.length()));
}
```

In the above example, we use only two lines of code to replace the five that worked in the first example. One line to wrap the object into an *Optional* object and the next to perform implicit validation as well as execute the code.

5. Default Value With *orElse()*

The *orElse()* method is used to retrieve the value wrapped inside an *Optional* instance. It takes one parameter which acts as a default value. The *orElse()* method returns the wrapped value if it's present and its argument otherwise:

```
@Test
public void whenOrElseWorks_thenCorrect() {
    String nullName = null;
    String name = Optional.ofNullable(nullName).orElse("john");
    assertEquals("john", name);
}
```

6. Default Value With *orElseGet()*

The *orElseGet()* method is similar to *orElse()*. However, instead of taking a value to return if the *Optional* value is not present, it takes a supplier functional interface which is invoked and returns the value of the invocation:

```
@Test
public void whenOrElseGetWorks_thenCorrect() {
    String nullName = null;
    String name = Optional.ofNullable(nullName).orElseGet(() -> "john'
    assertEquals("john", name);
}
```

7. Difference Between *orElse* and *orElseGet()*

To a lot of programmers who are new to *Optional* or Java 8, the difference between *orElse()* and *orElseGet()* is not clear. As a matter of fact, these two methods give the impression that they overlap each other in functionality.

However, there's a subtle but very important difference between the two which can affect the performance of our code drastically if not well understood.

Let's create a method called *getMyDefault()* in the test class which takes no arguments and returns a default value:

```
public String getMyDefault() {
    System.out.println("Getting Default Value");
    return "Default Value";
}
```

Let's see two tests and observe their side effects to establish both where *orElse()* and *orElseGet()* overlap and where they differ:

```
@Test
public void whenOrElseGetAndOrElseOverlap_thenCorrect() {
    String text = null;

    String defaultText = Optional.ofNullable(text).orElseGet(this::ge
    assertEquals("Default Value", defaultText);

    defaultText = Optional.ofNullable(text).orElse(getMyDefault());
    assertEquals("Default Value", defaultText);
}
```

In the above example, we wrap a null text inside an *Optional* object and we attempt to get the wrapped value using each of the two approaches. The side effect is as below:

```
Getting default value...
Getting default value...
```

The *getMyDefault()* method is called in each case. It so happens that **when the wrapped value is not present, then both *orElse()* and *orElseGet()* work exactly the same way**.

Now let's run another test where the value is present and ideally, the default value should not even be created:

```
@Test
public void whenOrElseGetAndOrElseDiffer_thenCorrect() {
    String text = "Text present";

    System.out.println("Using orElseGet:");
    String defaultText
      = Optional.ofNullable(text).orElseGet(this::getMyDefault);
    assertEquals("Text present", defaultText);

    System.out.println("Using orElse:");
    defaultText = Optional.ofNullable(text).orElse(getMyDefault());
    assertEquals("Text present", defaultText);
}
```

In the above example, we are no longer wrapping a *null* value and the rest of the code remains the same. Now let's take a look at the side effect of running this code:

```
Using orElseGet:
Using orElse:
Getting default value...
```

Notice that when using *orElseGet()* to retrieve the wrapped value, the *getMyDefault()* method is not even invoked since the contained value is present.

However, when using *orElse()*, whether the wrapped value is present or not, the default object is created. So in this case, we have just created one redundant object that is never used.

In this simple example, there is no significant cost to creating a default object, as the JVM knows how to deal with such. **However, when a method such as *getMyDefault()* has to make a web service call or even query a database, then the cost becomes very obvious.**

8. Exceptions with *orElseThrow()*

The *orElseThrow()* method follows from *orElse()* and *orElseGet()* and adds a new approach for handling an absent value. Instead of returning a default value when the wrapped value is not present, it throws an exception:

```
@Test(expected = IllegalArgumentException.class)
public void whenOrElseThrowWorks_thenCorrect() {
    String nullName = null;
    String name = Optional.ofNullable(nullName).orElseThrow(
      IllegalArgumentException::new);
}
```

Method references in Java 8 come in handy here, to pass in the exception constructor.

9. Returning Value with *get()*

The final approach for retrieving the wrapped value is the *get()* method:

```
@Test
public void givenOptional_whenGetsValue_thenCorrect() {
    Optional<String> opt = Optional.of("baeldung");
    String name = opt.get();
    assertEquals("baeldung", name);
}
```

However, unlike the above three approaches, *get()* can only return a value if the wrapped object is not *null*, otherwise, it throws a no such element exception:

```
@Test(expected = NoSuchElementException.class)
public void givenOptionalWithNull_whenGetThrowsException_thenCorrect()
    Optional<String> opt = Optional.ofNullable(null);
    String name = opt.get();
}
```

This is the major flaw of the *get()* method. Ideally, *Optional* should help us to avoid such unforeseen exceptions. Therefore, this approach works against the objectives of *Optional* and will probably be deprecated in a future release.

It is, therefore, advisable to use the other variants which enable us to prepare for and explicitly handle the *null* case.

10. Conditional Return with *filter()*

We can run an inline test on our wrapped value with the *filter* method. It takes a predicate as an argument and returns an *Optional* object. If the wrapped value passes testing by the predicate, then the *Optional* is returned as-is.

However, if the predicate returns *false*, then it will return an empty *Optional*:

```
@Test
public void whenOptionalFilterWorks_thenCorrect() {
    Integer year = 2016;
    Optional<Integer> yearOptional = Optional.of(year);
    boolean is2016 = yearOptional.filter(y -> y == 2016).isPresent();
    assertTrue(is2016);
    boolean is2017 = yearOptional.filter(y -> y == 2017).isPresent();
    assertFalse(is2017);
}
```

The *filter* method is normally used this way to reject wrapped values based on a predefined rule. We could use it to reject a wrong email format or a password that is not strong enough.

Let's look at another meaningful example. Let's say we want to buy a modem and we only care about its price. We receive push notifications on modem prices from a certain site and store these in objects:

```
public class Modem {
    private Double price;

    public Modem(Double price) {
        this.price = price;
    }
    // standard getters and setters
}
```

We then feed these objects to some code whose sole purpose is to check if the modem price is within our budget range for it.

Let's now take a look at the code without *Optional*:

```
public boolean priceIsInRange1(Modem modem) {
    boolean isInRange = false;

    if (modem != null && modem.getPrice() != null
      && (modem.getPrice() >= 10
        && modem.getPrice() <= 15)) {

        isInRange = true;
    }
    return isInRange;
}
```

Pay attention to how much code we have to write to achieve this, especially in the *if* condition. The only part of the if the condition that is critical to the application is the last price-range check; the rest of the checks are defensive:

```
@Test
public void whenFiltersWithoutOptional_thenCorrect() {
    assertTrue(priceIsInRange1(new Modem(10.0)));
    assertFalse(priceIsInRange1(new Modem(9.9)));
    assertFalse(priceIsInRange1(new Modem(null)));
    assertFalse(priceIsInRange1(new Modem(15.5)));
    assertFalse(priceIsInRange1(null));
}
```

Apart from that, it's possible to forget about the null checks on a long day without getting any compile-time errors.

Now let us look at a variant with *Optional#filter*:

```
public boolean priceIsInRange2(Modem modem2) {
     return Optional.ofNullable(modem2)
       .map(Modem::getPrice)
       .filter(p -> p >= 10)
       .filter(p -> p <= 15)
       .isPresent();
 }
```

The *map* call is simply used to transform a value to some other value. Keep in mind that this operation does not modify the original value.

In our case, we are obtaining a price object from the *Model* class. We will look at the *map()* method in detail in the next section.

First of all, if a *null* object is passed to this method, we don't expect any problem.

Secondly, the only logic we write inside its body is exactly what the method name describes, price range check. *Optional* takes care of the rest:

```
@Test
public void whenFiltersWithOptional_thenCorrect() {
    assertTrue(priceIsInRange2(new Modem(10.0)));
    assertFalse(priceIsInRange2(new Modem(9.9)));
    assertFalse(priceIsInRange2(new Modem(null)));
    assertFalse(priceIsInRange2(new Modem(15.5)));
    assertFalse(priceIsInRange2(null));
}
```

The previous approach promises to check price range but has to do more than that to defend against its inherent fragility. Therefore, we can use the *filter* method to replace unnecessary *if* statements and reject unwanted values.

11. Transforming Value with *map()*

In the previous section, we looked at how to reject or accept a value based on a filter. We can use a similar syntax to transform the *Optional* value with the *map()* method:

```
@Test
public void givenOptional_whenMapWorks_thenCorrect() {
    List<String> companyNames = Arrays.asList(
      "paypal", "oracle", "", "microsoft", "", "apple");
    Optional<List<String>> listOptional = Optional.of(companyNames);

    int size = listOptional
      .map(List::size)
      .orElse(0);
    assertEquals(6, size);
}
```

In this example, we wrap a list of strings inside an *Optional* object and use its *map* method to perform an action on the contained list. The action we perform is to retrieve the size of the list.

The *map* method returns the result of the computation wrapped inside *Optional*. We then have to call an appropriate method on the returned *Optional* to retrieve its value.

Notice that the *filter* method simply performs a check on the value and returns a *boolean*. On the other hand, the *map* method takes the existing value, performs a computation using this value and returns the result of the computation wrapped in an *Optional* object:

```
@Test
public void givenOptional_whenMapWorks_thenCorrect2() {
    String name = "baeldung";
    Optional<String> nameOptional = Optional.of(name);

    int len = nameOptional
     .map(String::length)
     .orElse(0);
    assertEquals(8, len);
}
```

We can chain *map* and *filter* together to do something more powerful.

Let's assume we want to check the correctness of a password input by a user; we can clean the password using a *map* transformation and check it's correctness using a *filter*:

```
@Test
public void givenOptional_whenMapWorksWithFilter_thenCorrect() {
    String password = " password ";
    Optional<String> passOpt = Optional.of(password);
    boolean correctPassword = passOpt.filter(
      pass -> pass.equals("password")).isPresent();
    assertFalse(correctPassword);

    correctPassword = passOpt
      .map(String::trim)
      .filter(pass -> pass.equals("password"))
      .isPresent();
    assertTrue(correctPassword);
}
```

As we can see, without first cleaning the input, it will be filtered out – yet users may take for granted that leading and trailing spaces all constitute input. So we transform dirty password into a clean one with a *map* before filtering out incorrect ones.

12. Transforming Value with *flatMap()*

Just like the *map()* method, we also have the *flatMap()* method as an alternative for transforming values. The difference is that *map* transforms values only when they are unwrapped whereas *flatMap* takes a wrapped value and unwraps it before transforming it.

Previously, we've created simple *String* and *Integer* objects for wrapping in an *Optional* instance. However, frequently, we will receive these objects from an accessor of a complex object.

To get a clearer picture of the difference, let's have a look at a *Person* object that takes a person's details such as name, an age and a password:

```
public class Person {
    private String name;
    private int age;
    private String password;

    public Optional<String> getName() {
        return Optional.ofNullable(name);
    }

    public Optional<Integer> getAge() {
        return Optional.ofNullable(age);
    }

    public Optional<String> getPassword() {
        return Optional.ofNullable(password);
    }

    // normal constructors and setters
}
```

We would normally create such an object and wrap it in an *Optional* object just like we did with String. Alternatively, it can be returned to us by another method call:

```
Person person = new Person("john", 26);
Optional<Person> personOptional = Optional.of(person);
```

Notice now that when we wrap a *Person* object, it will contain nested *Optional* instances:

```
@Test
public void givenOptional_whenFlatMapWorks_thenCorrect2() {
    Person person = new Person("john", 26);
    Optional<Person> personOptional = Optional.of(person);

    Optional<Optional<String>> nameOptionalWrapper
      = personOptional.map(Person::getName);
    Optional<String> nameOptional
      = nameOptionalWrapper.orElseThrow(IllegalArgumentException::new
    String name1 = nameOptional.orElse("");
    assertEquals("john", name1);

    String name = personOptional
      .flatMap(Person::getName)
      .orElse("");
    assertEquals("john", name);
}
```

Here, we're trying to retrieve the name attribute of the *Person* object to perform an assertion.

Note how we achieve this with *map()* method in the third statement and then notice how we do the same with *flatMap()* method afterwards.

The *Person::getName* method reference is similar to the *String::trim* call we had in the previous section for cleaning up a password.

The only difference is that *getName()* returns an *Optional* rather than a String as did the *trim()* operation. This, coupled with the fact that a *map* transformation wraps the result in an *Optional* object leads to a nested *Optional*.

While using *map()* method, therefore, we need to add an extra call to retrieve the value before using the transformed value. This way, the *Optional* wrapper will be removed. This operation is performed implicitly when using *flatMap*.

13. Chaining *Optional*s in Java 8

Sometimes, we may need to get the first non-empty *Optional* object from a number of *Optional*s. In such cases, it would be very convenient to use a method like *orElseOptional()*. Unfortunately, such operation is not directly supported in Java 8.

Let's first introduce a few methods that we'll be using throughout this section:

```
1  private Optional<String> getEmpty() {
2      return Optional.empty();
3  }
4
5  private Optional<String> getHello() {
6      return Optional.of("hello");
7  }
8
9  private Optional<String> getBye() {
10     return Optional.of("bye");
11 }
12
13 private Optional<String> createOptional(String input) {
14     if (input == null || "".equals(input) || "empty".equals(input)) {
15         return Optional.empty();
16     }
17     return Optional.of(input);
18 }
```

In order to chain several *Optional* objects and get the first non-empty one in Java 8, we can use the *Stream* API:

```
@Test
public void givenThreeOptionals_whenChaining_thenFirstNonEmptyIsReturn
    Optional<String> found = Stream.of(getEmpty(), getHello(), getBye
      .filter(Optional::isPresent)
      .map(Optional::get)
      .findFirst();

    assertEquals(getHello(), found);
}
```

The downside of this approach is that all of our *get* methods are always executed, regardless of where a non-empty *Optional* appears in the *Stream*.

If we want to lazily evaluate the methods passed to *Stream.of()*, we need to use the method reference and the *Supplier* interface:

```
@Test
public void givenThreeOptionals_whenChaining_thenFirstNonEmptyIsRetur
    Optional<String> found =
      Stream.<Supplier<Optional<String>>>of(this::getEmpty, this::get
        .map(Supplier::get)
        .filter(Optional::isPresent)
        .map(Optional::get)
        .findFirst();

    assertEquals(getHello(), found);
}
```

In case we need to use methods that take arguments, we have to resort to lambda expressions:

```
@Test
public void givenTwoOptionalsReturnedByOneArgMethod_whenChaining_ther
    Optional<String> found = Stream.<Supplier<Optional<String>>>of(
      () -> createOptional("empty"),
      () -> createOptional("hello")
    )
      .map(Supplier::get)
      .filter(Optional::isPresent)
      .map(Optional::get)
      .findFirst();

    assertEquals(createOptional("hello"), found);
}
```

Often, we'll want to return a default value in case all of the chained *Optional*s are empty. We can do so just by adding a call to *orElse()* or *orElseGet()* as in the following example:

```
@Test
public void givenTwoEmptyOptionals_whenChaining_thenDefaultIsReturnec
    String found = Stream.<Supplier<Optional<String>>>of(
      () -> createOptional("empty"),
      () -> createOptional("empty")
    )
      .map(Supplier::get)
      .filter(Optional::isPresent)
      .map(Optional::get)
      .findFirst()
      .orElseGet(() -> "default");

    assertEquals("default", found);
}
```

14. JDK 9 *Optional* API

The release of Java 9 added even more new methods to the *Optional* API:

- the *or()* method for providing a supplier that creates an alternative *Optional*
- the *ifPresentOrElse()* method that allows executing an action if the *Optional* is present or another action if not
- *stream()* method for converting an *Optional* to a *Stream*

Here is the complete article for further reading (/java-9-optional).

15. Misusage of *Optionals*

Finally, let's see a tempting, however dangerous, way to use *Optionals*: passing an *Optional* parameter to a method.

Imagine we have a list of *Person* and we want a method to search through that list for people having a given name. Also, we would like that method to match entries with at least a certain age, if it's specified. With this parameter being optional, we come with this method:

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

Then, we release our method and another developer tries to use it:

```
someObject.search(people, "Peter", null);
```

Now, the developer executes its code and gets a *NullPointerException.* **There we are, having to null check our optional parameter, which defeats our initial purpose in wanting to avoid this kind of situation.**

Here are some possibilities we could have done to handle it better:

```
public static List<Person> search(List<Person> people, String name, In
    // Null checks for people and name
    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());
}
```

There, the parameter's still optional, but we handle it in only one check. Another possibility would have been to **create two overloaded methods**:

```
public static List<Person> search(List<Person> people, String name) {
    return doSearch(people, name, 0);
}

public static List<Person> search(List<Person> people, String name, 
    return doSearch(people, name, age);
}

private static List<Person> doSearch(List<Person> people, String name
    // Null checks for people and name
    return people.stream()
            .filter(p -> p.getName().equals(name))
            .filter(p -> p.getAge().get().intValue() >= age)
            .collect(Collectors.toList());
}
```

That way we offer a clear API with two methods doing different things (though they share the implementation).

So, there are solutions to avoid using *Optional*s as method parameters. **The intent of Java when releasing *Optional* was to use it as a return type**, thus indicating that a method could return an empty value. As a matter of fact, the practice of using *Optional* as a method parameter is even discouraged by some code inspectors (https://rules.sonarsource.com/java/RSPEC-3553).

16. *Optional* and Serialization

As discussed above, *Optional* is meant to be used as a return type. Trying to use it as a field type is not recommended.

Additionally, **using *Optional* in a serializable class will result in a *NotSerializableException***. Our article Java *Optional* as Return Type (https://www.baeldung.com/java-optional-return) further addresses the issues with serialization.

And, in Using *Optional* with Jackson (https://www.baeldung.com/jackson-optional), we explain what happens when *Optional* fields are serialized, along with a few workarounds to achieve the desired results.

17. Conclusion

In this article, we covered most of the important features of Java 8 *Optional* class.

We have also briefly explored some reasons why we would choose to use *Optional* instead of explicit null checking and input validation.

We also learned how to get the value of an *Optional*, or a default one if empty, with the *get(), orElse()* and *orElseGet()* methods (and saw the important difference between the two last (/java-filter-stream-of-optional)).

Then, we saw how to transform or filter our *Optional*s with *map(), flatMap()* and *filter()*.

We saw what a fluent *API Optional* offers as it allows us to chain the different methods easily.

Finally, we saw how using *Optionals* as method parameters is a bad idea and how to avoid it.

The source code for all examples in the article is available over on GitHub. (https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-optional)

Learning to "Build your API **with Spring**"?

Enter your email address

>> Get the eBook

Comments are closed on this article!

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)