Última modificación: 31 de agosto de 2017

por baeldung (http://www.baeldung.com/author/baeldung/) (http://www.baeldung.com/author/baeldung/)

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

1. Información general

En este artículo, vamos a mostrar la nueva clase *opcional* que se introdujo en Java 8.

El objetivo de la clase es proporcionar una solución de nivel de tipo para representar valores opcionales en lugar de utilizar referencias *nulas* .

Para obtener una comprensión más profunda de por qué debería preocuparse por la clase Opcional, eche un vistazo al artículo (http://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html) oficial de Oracle .

Opcional es parte del *paquete java.util* , así que necesitamos hacer esta importación:

```java
import java.util.Optional;
```

2. Creando Objetos *Opcionales*

Hay varias formas de crear objetos *opcionales* . Para crear un objeto *opcional* vacío :

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

La API *isPresent* se usa para verificar si hay un valor dentro del objeto *Opcional* . Un valor está presente si y solo si hemos creado *Opcional* con un valor no nulo. Veremos la API *isPresent* en la siguiente sección.

También podemos crear un objeto opcional con la estática *de la* API:

[illegible]

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

Pero, en caso de que esperemos algunos valores nulos para el argumento pasado, podemos usar la API *ofNullable* :

```java
@Test
public void givenNonNull_whenCreatesNullable_thenCorrect() {
    String name = "baeldung";
    Optional<String> opt = Optional.ofNullable(name);
    assertEquals("Optional[baeldung]", opt.toString());
}
```

De esta forma, si pasamos una referencia nula, no lanza una excepción, sino que devuelve un objeto *opcional* vacío como si lo hubiéramos creado con la API *Optional.empty* :

```java
@Test
public void givenNull_whenCreatesNullable_thenCorrect() {
    String name = null;
    Optional<String> opt = Optional.ofNullable(name);
    assertEquals("Optional.empty", opt.toString());
}
```

Esta API devuelve verdadero si y sólo si el valor envuelto no es nulo.

4. Acción condicional con *ifPresent ()*

La API *ifPresent* nos permite ejecutar algún código en el valor empaquetado si se determina que no es nulo. Antes de *Opcional* , haríamos algo como esto:

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

```
    Optional<String> opt = Optional.of("baeldung");

    opt.ifPresent(name -> System.out.println(name.length()));
}
```

En el ejemplo anterior, usamos solo dos líneas de código para reemplazar las cinco que funcionaron en el primer ejemplo. Una línea para envolver el objeto en un objeto *opcional* y el siguiente para realizar una validación implícita y ejecutar el código.

5. Valor predeterminado con *orElse*

La API *orElse* se usa para recuperar el valor envuelto dentro de una instancia *opcional* . Toma un parámetro que actúa como un valor predeterminado. Con *orElse* , el valor envuelto se devuelve si está presente y el argumento dado a *orElse* se devuelve si el valor envuelto está ausente:

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

6. Valor predeterminado con *orElseGet*

La API *orElseGet* es similar a *orElse* . Sin embargo, en lugar de tomar un valor para devolver si el valor *opcional* no está presente, toma una interfaz funcional del proveedor que se invoca y devuelve el valor de la invocación:

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

7. Diferencia entre *orElse* y *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 if fact, these two APIs give the impression that they overlap each other in functionality.

However, there is a subtle but very important difference between the two which can affect the performance of your 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";
}
```

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:

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

The *getMyDefault* API is called in each case. It so happens that **when the wrapped value is not present, then both *orElse* and *orElseGet* APIs 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* API 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* API follows from *orElse* and *orElseGet* in 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:

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

    assertEquals("baeldung", name);
}
```

However, unlike the above three approaches, the get API 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* API. 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()*

The *filter* API is used to run an inline test on the wrapped value. 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 an empty *Optional* is returned:

```
@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* API is normally used this way to reject wrapped values based on a predefined rule. You 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
}
```

```
@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* API:

```
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 API in detail in the next section.

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

Secondly, the only logic we write inside its body is exactly what the API 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 API promises to check price range but has to do more than that to defend against its inherent fragility. Therefore, the *filter* API can be used to replace unnecessary *if* statements to 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. A similar syntax can be used to transform the *Optional* value with the *map* API:

Notice that the filter API simply performs a check on the value and returns a boolean. On the other hand, the map API 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 you 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* API, we also have the *flatMap* API 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:

```
}
```

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 API 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* API in the third statement and then notice how we do the same with *flatMap* API afterward.

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 result is wrapped in an *Optional* object leads to a nested *Optional*.

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

13. Conclusion

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

También hemos explorado brevemente algunas razones por las que elegiríamos utilizar la validación de entrada y verificación nula *opcional en* lugar de explícita.

Finalmente, abordamos la diferencia sutil pero significativa entre *orElse* y *orElseGet* ; aquí hay algunas buenas lecturas adicionales (/java-filter-stream-of-optional) sobre el tema.

El código fuente para todos los ejemplos en el artículo está disponible en GitHub. (https://github.com/eugenp/tutorials/tree/master/core-java-8)

CATEGORÍAS

PRIMAVERA (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
DESCANSO (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SEGURIDAD (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCIA (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)

ACERCA DE

ACERCA DE BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)
LOS CURSOS (HTTP://COURSES.BAELDUNG.COM)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
EL ARCHIVO COMPLETO (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
ESCRIBIR PARA BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)
POLÍTICA DE PRIVACIDAD (HTTP://WWW.BAELDUNG.COM/PRIVACY-POLICY)
TÉRMINOS DE SERVICIO (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)
CONTACTO (HTTP://WWW.BAELDUNG.COM/CONTACT)
INFORMACIÓN DE LA COMPAÑÍA (HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
PUBLICIDAD EN JAVA WEEKLY (HTTP://WWW.BAELDUNG.COM/JAVA-WEEKLY-SPONSORSHIP)
TRABAJO DE CONSULTORÍA (HTTP://WWW.BAELDUNG.COM/CONSULTING)