ABOUT DANI BUIZA

Daniel Gutierrez Diez holds a Master in Computer Science Engineering from the University of Oviedo (Spain) and a Post Grade as Specialist in Foreign Trade from the UNED (Spain). Daniel has been working for different clients and companies in several Java projects as programmer, designer, trainer, consultant and technical lead.

Java 8 Optional Example

Posted by: Dani Buiza in Optional August 7th, 2014

In this article we are going to show how to use the new

```
java.util.Optional
```

class.

The null reference is a very common problem in Java, everybody got once a

```
NullPointerException
```

because some variable or input parameter was not properly validated. In Java, null, can have a logical value and a meaning; so it is important to take it into consideration and do not ignore it.

With the introduction of

```
java.util.Optional
```

in Java 8, several new possibilities for handling this problematic are available.

The new class is based on the analog functionalities in Haskell and Scala. It contains a value that can be present or not; if the value is not present, the

```
Optional
```

instance is said to be empty.

All examples and code snippets listed have been done using Eclipse Luna version 4.4 and Java version 8 update 5 and can be downloaded at the end of this article.

NullPointerException

For the ones that do not know what a

```
NullPointerException
```

is, just try:

```
1 String strNull0 = null;
2 System.out.println( strNull0.contains( "something" ) );
```

The code above would compile but we would get a warning like:

```
1 Null pointer access: The variable strNull can only be null at this location
```

In order to handle this, we can check and validate for null, or we can surround the block with a

```
try catch
```

. In the following chapters we are going to see how to handle this issue using the new

```
Optional
```

class.

Optionals

In order to create an

```
Optional
```

we have to indicate what type of value is going to contain:

```
1 Optional emptyOptional = Optional.empty();
```

The code above shows the creation of an empty optional, that is, its value is null or not initialized. In order to access the value of an optional we can use the method

```
get()
```

; if we try to access the value of the example shown above, we would get the following exception.

```
01 try
02 {
03     /* empty */
04     Optional emptyOptional = Optional.empty();
05     System.out.println( emptyOptional.get() );
06 }
07 catch( NoSuchElementException ex )
08 {
09     System.out.println( "expected NoSuchElementException" ); //this is executed
10 }
```

The exception thrown is of the type

```
java.util.NoSuchElementException
```

and means that the

```
Optional
```

value is not initialized or null.

In order to create an

```
Optional
```

we can use an existing object and pass it to the

```
Optional
```

using the static method

```
of()
```

:

```
1 Optional nonEmptyOptional = Optional.of( str );
```

The object passed to the method

```
of()
```

has to be different to null. In case we want to offer the possibility of using potential null values, we can use

```
ofNullable()
```

:

```
1 String strNull = null;
2 Optional nullableOptional = Optional.ofNullable( strNull );
```

If we try to pass a null object to an

```
Optional
```

using the method

```
of()
```

we will get a

```
NullPointerException
```

. At the end of this article there is a file to download with examples of all these possible combinations.

Get, orElse, ifElse, orElseThrow…

In order to get the value of an

```
Optional
```

you can use the

```
get()
```

method as shown above. The problem of this method is that you can get a

```
NullPointerException
```

if the value is not initialized. So, although it has some benefits is not solving all our problems.

There are several methods that can be used to retrieve the value of an

```
Optional
```

and handle at the same time the possibility that it is a null reference. We are going to see some of them:

orElse

We can use the method

```
orElse()
```

:

```
1 Optional optionalCar = Optional.empty();
2 price = optionalCar.orElse( defaultCar ).getPrice();
```

In the code shown above we are trying to access to the price of a Car object, if the Car object is not initialize (or it is null) we will retrieve the price for the default car that we have defined before.

orElseThrow

We can indicate the

```
Optional
```

to throw an exception in case its value is null:

```
1 Optional optionalCarNull = Optional.ofNullable( carNull );
2 optionalCarNull.orElseThrow( IllegalStateException::new );
```

In this case, an

```
IllegalStateException
```

will be thrown.

isPresent

There is the possiblity to check directly if the value is initialized and not null:

```
1 Optional stringToUse = Optional.of( "optional is there" );
2 if( stringToUse.isPresent() )
3 {
4     System.out.println( stringToUse.get() );
5 }
```

ifPresent

And also the option to execute actions directly when the value is present, in combination with Lambdas:

```
1 Optional stringToUse = Optional.of( "optional is there" );
2 stringToUse.ifPresent( System.out::println );
```

The code shown above and the one before are doing exactly the same. I would prefer to use the second one.

Filtering and mapping in combination with Lambdas

So these are some of the available methods to retrieve the values of an

```
Optional
```

and to handle the null references. Now we are going to see the options that are offered in combination with

```
Lambdas
```

:

Filter (Lambdas)

The Optional class contains a

```
filter()
```

method that expects a

```
Predicate
```

and returns an

```
Optional
```

back if the

```
Predicate
```

is true. Here are some examples:

```
01 // if the value is not present
02 Optional carOptionalEmpty = Optional.empty();
03 carOptionalEmpty.filter( x -> "250".equals( x.getPrice() ) ).ifPresent( x -> System.out.println(
   x.getPrice() + " is ok!" ) );
04
05 // if the value does not pass the filter
06 Optional carOptionalExpensive = Optional.of( new Car( "3333" ) );
07 carOptionalExpensive.filter( x -> "250".equals( x.getPrice() ) ).ifPresent( x -> System.out.println(
   x.getPrice() + " is ok!" ) );
08
09 // if the value is present and does pass the filter
10 Optional carOptionalOk = Optional.of( new Car( "250" ) );
11 carOptionalOk.filter( x -> "250".equals( x.getPrice() ) ).ifPresent( x -> System.out.println( x.getPrice()
   + " is ok!" ) );
```

As we can see in the snippet above we do not have to take care of the null reference of the value, we can just apply our filters directly and the

```
Optional
```

takes care of all the rest.

Map (Lambdas)

It is also very interesting the method

```
map()
```

. This method "maps" or converts an Optional to another Optional using a Function as parameter. The mapping is only executed, if the result of the past Function is not null. Here are some examples:

```
1 // non empty string map to its length -> we get the lenght as output (18)
2 Optional stringOptional = Optional.of( "looooooooong string" );
3 Optional sizeOptional = stringOptional.map( String::length ); //map from Optional to Optional
4 System.out.println( "size of string " + sizeOptional.orElse( 0 ) );
5
6 // empty string map to its length -> we get 0 as lenght
7 Optional stringOptionalNull = Optional.ofNullable( null );
8 Optional sizeOptionalNull = stringOptionalNull.map( x -> x.length()  ); // we can use Lambdas as we want
9 System.out.println( "size of string " + sizeOptionalNull.orElse( 0 ) );
```

Summary

And that's it!
In this article we saw several examples about how to use the new

```
Optional
```

class coming out in Java 8. This class allows us to manage null references in a clear and concise way and to handle the famous

```
NullPointerException
```

more effectively. It is interesting to mention that there are also typified "optionals" for the types

```
double
```

,

```
int
```