HowToDoInJava

Java 8 Optionals : Complete Reference

Last Updated: December 26, 2020 | By: Lokesh Gupta | Java 8 | Java 8, Java Exceptions, Optional

All of us must have encountered `NullPointerException` in our applications. This exception happen when you try to utilize a object reference which has not been initialized, initialized with null or simply does not point to any instance. ***NULL simply means 'absence of a value'***. Most probably, the Romans were only ones, who didn't run into this null problem who started counting at I, II, III.. (no zero). Perhaps, they couldn't model the absence of apples on their markets. [:-)]

"I call it my billion-dollar mistake." – **Sir C. A. R. Hoare**, on his invention of the null reference

In this article, I am going to discuss one of **java 8 features** for this specific usecase i.e. `Optional`. This article has been divided into multiple sections for the sake of clarity and differentiation between multiple concepts.

Discussion Points

1) What is the Type of Null?

In Java, we use a reference type to gain access to an object, and when we don't have a specific object to make our reference point to, then we set such reference to null to imply the absence of a value. Right?

The use of null is so common that we rarely put more thoughts on it. For example, field members of objects are automatically initialized to null and programmers typically initialize reference types to null when they don't have an initial value to give them and, in general, null is used everytime where we don't know or we don't have a value to give to a reference.

> FYI, in Java ***null is actually a type***, a special one. It has no name so we cannot declare variables of its type or cast any variables to it; in fact there is only a single value that can be associated with it (i.e. the literal null). Remember that unlike any other types in Java, a null reference can be safely assigned to any other reference types without any error(See **JLS 3.10.7** and **4.1**).

2) What is wrong with just returning null?

Generally, the API designers put the descriptive java docs in APIs and mention there that API can return a null value, and in which case(s). Now, the problem is that the caller of the API might have missed reading the javadoc for any reason, and **forget about handling the null case**. This is going to be a bug in future for sure.

And believe me, this happens frequently and is one of the main causes of null pointer exceptions, although not the only one. So, take a lesson here that always read the java dcos of an API when you are using it first time (... at least) [:-)].

> Now we know that null is a problem in most of the cases, what's best way to handle them?

A good solution is to always **initialize your object references with some value**, and never with null. In this way, you will never encounter `NullPointerException`. Fair enough. But in practical we always don't have a default value for a reference. So, how those cases should be handled?

Above question is right in many senses. Well, **java 8 `Optionals`** are the answer here.

3) How Java 8 Optionals provide the solution?

Optional is a way of replacing a nullable T reference with a non-null value. An Optional may either contain a non-null T reference (in which case we say the reference is "present"), or it may contain nothing (in which case we say the reference is "absent").

> Remember that it is **never said that optional "contain null"**.

```
Optional<Integer> canBeEmpty1 = Optional.of(5);
canBeEmpty1.isPresent();          // returns true
canBeEmpty1.get();              // returns 5

Optional<Integer> canBeEmpty2 = Optional.empty();
canBeEmpty2.isPresent();          // returns false
```

You can also **view Optional as a single-value container that either contains a value or doesn't**.

It is important to note that the intention of the Optional class is not to replace every single null reference. Instead, its purpose is to **help design more-comprehensible APIs** so that by just reading the signature of a method, you can tell whether you can expect an optional value. This forces you to fetch the value from Optional and work on it, and at the same time handle the case where optional will be empty. Well, this is exactly the

solution of null references/return values which ultimately result into `NullPointerException`.

Below are some examples to learn more about how Optional should be created and used in your application code.

a) Creating Optional objects

There are 3 major ways to create an Optional.

i) Use **Optional.empty()** to create empty optional.

```
Optional<Integer> possible = Optional.empty();
```

ii) Use **Optional.of()** to create optional with default non-null value. If you pass null in of(), then a NullPointerException is thrown immediately.

```
Optional<Integer> possible = Optional.of(5);
```

iii) Use **Optional.ofNullable()** to create an Optional object that may hold a null value. If parameter is null, the resulting Optional object would be empty (remember that value is absent; don't read it null).

```
Optional<Integer> possible = Optional.ofNullable(null);
//or
Optional<Integer> possible = Optional.ofNullable(5);
```

b) Do something If Optional value is present

You got your `Optional` object is first step. Now let's use it after checking whether it holds any value inside it.

```
Optional<Integer> possible = Optional.of(5);
```

```
possible.ifPresent(System.out::println);
```

You can re-write above code as below code as well. However, this is not the recommended use of `Optional` because it's not much of an improvement over nested null checks. They do look completely similar.

```
if(possible.isPresent()){
  System.out.println(possible.get());
}
```

If the `Optional` object were empty, nothing would be printed.

c) Default/absent values and actions

A typical pattern in programming is to return a default value if you determine that the result of an operation is null. In general, you can use the ternary operator; but with Optionals, you can write the code as below:

```
//Assume this value has returned from a method
Optional<Company> companyOptional = Optional.empty();

//Now check optional; if value is present then return it,
//else create a new Company object and retur it
Company company = companyOptional.orElse(new Company());

//OR you can throw an exception as well
Company company = companyOptional.orElseThrow(IllegalStateException::new);
```

d) Rejecting certain values using the filter method

Often you need to call a method on an object and check some property. e.g. in below example code check if company has a 'Finance' department; if it has, then print it.

```
Optional<Company> companyOptional = Optional.empty();
companyOptional.filter(department -> "Finance".equals(department.getName())
                    .ifPresent(() -> System.out.println("Finance is present"));
```

The **filter method** takes a **predicate** as an argument. If a value is present in the `Optional` object and it matches the predicate, the filter method returns that value; otherwise, it returns an empty `Optional` object. You might have seen a similar pattern already if you have used the filter method with the `Stream` interface.

Good, this code is looking closer to the problem statement and there are no verbose null checks getting in our way!

Wow!! We've come a long way from writing painful nested null checks to writing declarative code that is composable, readable, and better protected from null pointer exceptions.

4) What is inside Optional make it work?

If you open the sourcecode of Optional.java, you will find that value that Optional holds is defined as:

```
/**
 * If non-null, the value; if null, indicates no value is present
 */
private final T value;
```

And if you define an empty Optional then it is declared as below. **static keyword ensures that generally only one empty instance should exist per VM**.

```
/**
 * Common instance for {@code empty()}.
 */
private static final Optional<?> EMPTY = new Optional<>();
```

The default **no-args constructor is define private**, so you can't create an instance of Optional except 3 given ways above.

When you create an Optional then below call happen at end and assign the passed value to 'value' attribute.

```
this.value = Objects.requireNonNull(value);
```

When you try to **get a value from an Optional**, value is fetched if present other wise `NoSuchElementException` is thrown:

```
public T get() {
  if (value == null) {
    throw new NoSuchElementException("No value present");
  }
  return value;
}
```

Similarly, other functions defined in Optional class operate around the 'value' attribute only. Browse the sourcecode of Optional.java for more insight.

5) What is Optional trying to solve?

`Optional` is an attempt to reduce the number of null pointer exceptions in Java systems, by adding the possibility to build more expressive APIs considering that sometimes return values are missing. If `Optional` was there since the beginning, today most libraries and applications would likely deal better with missing return values, reducing the number of null pointer exceptions and the overall number of bugs in general.

By using `Optional`, user is forced to think about the exceptional case. Besides the increase in readability that comes from giving null a name, the **biggest advantage of Optional is its idiot-proof-ness**. It forces you to actively think about the absent case if you want your program to compile at all, since you have to actively unwrap the Optional and address that failure cases.

6) What is Optional not trying to solve?

Optional is **not meant to be a mechanism to avoid all types of null pointers**. e.g. The mandatory input parameters of methods and constructors will still have to be tested.

Like when using null, **Optional** does not help with conveying the meaning of an absent value. So the caller of the method will still have to check the javadoc of the API for understanding the meaning of the absent **Optional**, in order to deal with it properly.

Please note that `Optional` is not meant to be used in these below contexts, as possibly it won't buy us anything:

- in the domain model layer (it's not serializable)
- in DTOs (it's not serializable)
- in input parameters of methods
- in constructor parameters

7) How should Optional be used?

Optional should be used almost all the time ***as the return type of functions*** that might not return a value.

This is a quote from OpenJDK mailing list:

> The JSR-335 EG felt fairly strongly that Optional should not be on any more than needed to support the optional-return idiom only.
> Someone suggested maybe even renaming it to "***OptionalReturn***".

This essentially means that `Optional` should be used as the return type of certain service, repository or utility methods only where they truly serve the purpose.

8) Conclusion

In this article, we learned that how you can adopt the new **Java SE 8** `java.util.Optional`. The purpose of **Optional** is not to replace every single null reference in your code base but rather to help you design better APIs in which, just by

reading the signature of a method, users can tell whether to expect an optional value and deal with it appropriately.

That's all for this awesome feature. Let me know of your thoughts in comments section.

Happy Learning !!

Was this post helpful?

Let us know if you liked the post. That's the only way we can improve.

Yes

No

Recommended Reading:

1. Java 8 method reference example
2. Complete Java Annotations Tutorial
3. Complete Java Servlets Tutorial
4. Puzzle – Check if string is complete (contains all alphabets)
5. jQuery Selectors – Complete List
6. Java Pass-by-Value vs. Pass-by-Reference
7. Java Exact Arithmetic Operations Support in Math Class
8. Guide to IntStream in Java
9. Java Stream findFirst()
0. Java Stream findAny()

Join 7000+ Awesome Developers

Get the latest updates from industry, awesome resources, blog updates and much more.

Email Address

Subscribe

** We do not spam !!*

---

3 thoughts on “Java 8 Optionals : Complete Reference”

Herman Bovens

June 24, 2015 at 2:26 pm

Actually Oracle advises to use Optional as class fields as well:

https://www.oracle.com/technical-resources/articles/java/java8-optional.html

so that the following piece of code

```
String version = &amp;quot;UNKNOWN&amp;quot;;
if(computer != null){
  Soundcard soundcard = computer.getSoundcard();
  if(soundcard != null){
    USB usb = soundcard.getUSB();
    if(usb != null){
      version = usb.getVersion();
    }
  }
}
```

can be written as

```
String name = computer.flatMap(Computer::getSoundcard)
                          .flatMap(Soundcard::getUSB)
                          .map(USB::getVersion)
```

```
.orElse(&amp;quot;UNKNOWN&amp;quot;);
```

Reply

Lokesh Gupta

June 24, 2015 at 2:28 pm

Thanks for sharing.

Reply

Ash

November 14, 2019 at 8:45 pm

what about in case of object referance?

Reply

Leave a Comment

Name *

Email *

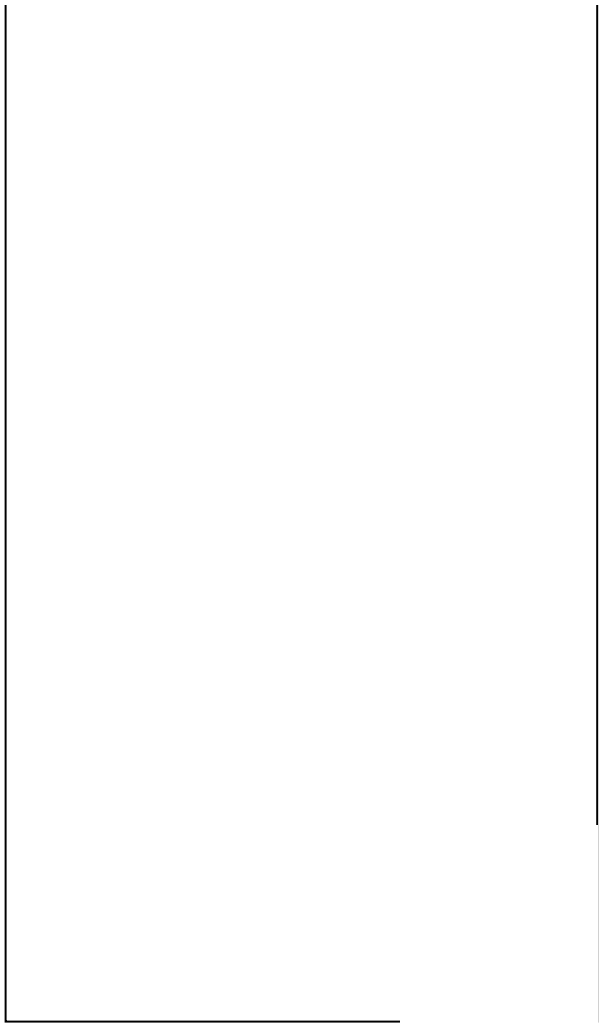
Website

Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search ...


BUY
STOCKS
Your ca
Othe
For mor
visit
tra



HowToDoInJava

A blog about Java and related technologies, the best practices, algorithms, and interview questions.

Meta Links

- About Me
- Contact Us
- Privacy policy
- Advertise
- Guest Posts

Blogs

REST API Tutorial

Copyright © 2022 · Hosted on Cloudways · Sitemap