Application Development Framework
Application Express
Big Data
Business Intelligence
Cloud Computing
Communications
Database Performance & Availability
Data Warehousing
Database
.NET
Dynamic Scripting Languages
Embedded
Digital Experience
Enterprise Architecture
Enterprise Management
Identity & Security
Java
Linux
Mobile
Service-Oriented Architecture
Solaris
SQL & PL/SQL
Systems - All Articles
Virtualization

Tired of Null Pointer Exceptions? Consider Using Java SE 8's `Optional`!

by Raoul-Gabriel Urma

Make your code more readable and protect it against null pointer exceptions.

Published March 2014

A wise man once said you are not a real Java programmer until you've dealt with a null pointer exception. Joking aside, the null reference is the source of many problems because it is often used to denote the absence of a value. Java SE 8 introduces a new class called `java.util.Optional` that can alleviate some of these problems.

Let's start with an example to see the dangers of null. Let's consider a nested object structure for a `Computer`, as illustrated in Figure 1.

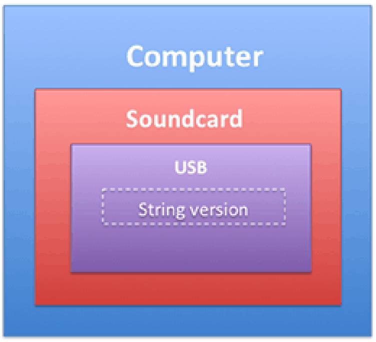

Figure 1: A nested structure for representing a `Computer`

What's possibly problematic with the following code?

```
String version = computer.getSoundcard().getUSB().getVersion();
```

This code looks pretty reasonable. However, many computers (for example, the Raspberry Pi) don't actually ship with a sound card. So what is the result of `getSoundcard()`?

A common (bad) practice is to return the null reference to indicate the absence of a sound card. Unfortunately, this means the call to `getUSB()` will try to return the USB port of a null reference, which will result in a `NullPointerException` at runtime and stop your program from running further. Imagine if your program was running on a customer's machine; what would your customer say if the program suddenly failed?

To give some historical context, Tony Hoare—one of the giants of computer science—wrote, "I call it my billion-dollar mistake. It was the invention of the null reference in 1965. I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement."

What can you do to prevent unintended null pointer exceptions? You can be defensive and add checks to prevent null dereferences, as shown in Listing 1:

```
String version = "UNKNOWN";
if(computer != null){
  Soundcard soundcard = computer.getSoundcard();
  if(soundcard != null){
    USB usb = soundcard.getUSB();
    if(usb != null){
      version = usb.getVersion();
    }
  }
}
```
Listing 1

However, you can see that the code in Listing 1 quickly becomes very ugly due to the nested checks. Unfortunately, we need a lot of boilerplate code to make sure we don't get a `NullPointerException`. In addition, it's just annoying that these checks get in the way of the business logic. In fact, they are decreasing the overall readability of our program.

Furthermore, it is an error-prone process; what if you forget to check that one property could be null? I will argue in this article that using null to represent the absence of a value is a wrong approach. What we need is a better way to model the absence and presence of a value.

To give some context, let's briefly look at what other programming languages have to offer.

What Alternatives to Null Are There?

Languages such as Groovy have a *safe navigation operator* represented by "`?.`" to safely navigate through potential null references. (Note that it is soon to be included in C#, too, and it was proposed for Java SE 7 but didn't make it into that release.) It works as follows:

```
String version = computer?.getSoundcard()?.getUSB()?.getVersion();
```

In this case, the variable `version` will be assigned to null if `computer` is null, or `getSoundcard()` returns null, or `getUSB()` returns null. You don't need to write complex nested conditions to check for null.

In addition, Groovy also includes the *Elvis operator* "`?:`" (if you look at it sideways, you'll recognize Elvis' famous hair), which can be used for simple cases when a default value is needed. In the following, if the expression that uses the safe navigation operator returns null, the default value `"UNKNOWN"` is returned; otherwise, the available version tag is returned.

```
String version =
    computer?.getSoundcard()?.getUSB()?.getVersion() ?: "UNKNOWN";
```

Other functional languages, such as Haskell and Scala, take a different view. Haskell includes a `Maybe` type, which essentially encapsulates an optional value. A value of type `Maybe` can contain either a value of a given type or nothing. There is no concept of a null reference. Scala has a similar construct called `Option[T]` to encapsulate the presence or absence of a value of type `T`. You then have to explicitly check whether a value is present or not using operations available on the `Option` type, which enforces the idea of "null checking." You can no longer "forget to do it" because it is enforced by the type system.

OK, we diverged a bit and all this sounds fairly abstract. You might now wonder, "so, what about Java SE 8?"

`Optional` in a Nutshell
Java SE 8 introduces a new class called `java.util.Optional<T>` that is inspired from the ideas of Haskell and Scala. It is a class that encapsulates an optional value, as illustrated in Listing 2 below and in Figure 1. You can view `Optional` as a single-value container that either contains a value or doesn't (it is then said to be "empty"), as illustrated in Figure 2.

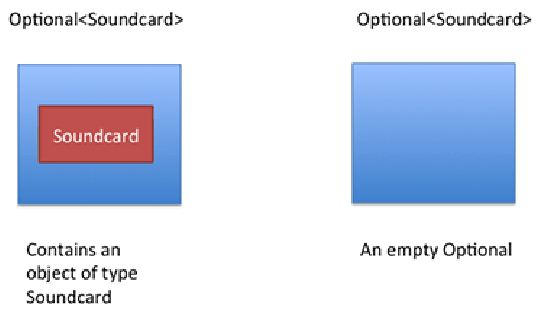

Figure 2: An optional sound card

We can update our model to make use of `Optional`, as shown in Listing 2:

```
public class Computer {
  private Optional<Soundcard> soundcard;
  public Optional<Soundcard> getSoundcard() { ... }
  ...
}

public class Soundcard {
  private Optional<USB> usb;
  public Optional<USB> getUSB() { ... }

}

public class USB{
  public String getVersion(){ ... }
}
```
Listing 2

The code in Listing 2 immediately shows that a computer might or might not have a sound card (the sound card is optional). In addition, a sound card can optionally have a USB port. This is an improvement, because this new model can now reflect clearly whether a given value is allowed to be missing. Note that similar ideas have been available in libraries such as Guava.

But what can you actually do with an `Optional<Soundcard>` object? After all, you want to get to the USB port's version number. In a nutshell, the `Optional` class includes methods to explicitly deal with the cases where a value is present or absent. However, the advantage compared to null references is that the `Optional` class forces you to think about the case when the value is not present. As a consequence, you can prevent unintended null pointer exceptions.

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 actively unwrap an `Optional` to deal with the absence of a value.

Patterns for Adopting `Optional`
Enough talking; let's see some code! We will first explore how typical null-check patterns can be rewritten using `Optional`. By the end of this article, you will understand how to use `Optional`, as shown below, to rewrite the code in Listing 1 that was doing several nested null checks:

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

Note: Make sure you brush up on the Java SE 8 lambdas and method references syntax (see "Java 8: Lambdas") as well as its stream pipelining concepts (see "Processing Data with Java SE 8 Streams").

Creating `Optional` objects
First, how do you create `Optional` objects? There are several ways:

Here is an empty `Optional`:

```
Optional<Soundcard> sc = Optional.empty();
```

And here is an `Optional` with a non-null value:

```
SoundCard soundcard = new Soundcard();
Optional<Soundcard> sc = Optional.of(soundcard);
```

If `soundcard` were null, a `NullPointerException` would be immediately thrown (rather than getting a latent error once you try to access properties of the `soundcard`).

Also, by using `ofNullable`, you can create an `Optional` object that may hold a null value:

```
Optional<Soundcard> sc = Optional.ofNullable(soundcard);
```

If soundcard were null, the resulting `Optional` object would be empty.

Do Something If a Value Is Present
Now that you have an `Optional` object, you can access the methods available to explicitly deal with the presence or absence of values. Instead of having to remember to do a null check, as follows:

```
SoundCard soundcard = ...;
if(soundcard != null){
  System.out.println(soundcard);
}
```

You can use the `ifPresent()` method, as follows:

```
Optional<Soundcard> soundcard = ...;
soundcard.ifPresent(System.out::println);
```

You no longer need to do an explicit null check; it is enforced by the type system. If the `Optional` object were empty, nothing would be printed.

You can also use the `isPresent()` method to find out whether a value is present in an `Optional` object. In addition, there's a `get()` method that returns the value contained in the `Optional` object, if it is present. Otherwise, it throws a `NoSuchElementException`. The two methods can be combined, as follows, to prevent exceptions:

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

However, this is not the recommended use of `Optional` (it's not much of an improvement over nested null checks), and there are more idiomatic alternatives, which we explore below.

Default Values and Actions

A typical pattern 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, as follows, to achieve this:

```
Soundcard soundcard =
  maybeSoundcard != null ? maybeSoundcard
            : new Soundcard("basic_sound_card");
```

Using an `Optional` object, you can rewrite this code by using the `orElse()` method, which provides a default value if `Optional` is empty:

```
Soundcard soundcard = maybeSoundcard.orElse(new Soundcard("defaut"));
```

Similarly, you can use the `orElseThrow()` method, which instead of providing a default value if `Optional` is empty, throws an exception:

```
Soundcard soundcard =
  maybeSoundCard.orElseThrow(IllegalStateException::new);
```

Rejecting Certain Values Using the `filter` Method

Often you need to call a method on an object and check some property. For example, you might need to check whether the USB port is a particular version. To do this in a safe way, you first need to check whether the reference pointing to a USB object is null and then call the `getVersion()` method, as follows:

```
USB usb = ...;
if(usb != null && "3.0".equals(usb.getVersion())){
  System.out.println("ok");
}
```

This pattern can be rewritten using the `filter` method on an `Optional` object, as follows:

```
Optional<USB> maybeUSB = ...;
maybeUSB.filter(usb -> "3.0".equals(usb.getVersion())
                    .ifPresent(() -> System.out.println("ok"));
```

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.

Extracting and Transforming Values Using the `map` Method

Another common pattern is to extract information from an object. For example, from a `Soundcard` object, you might want to extract the `USB` object and then further check whether it is of the correct version. You would typically write the following code:

```
if(soundcard != null){
  USB usb = soundcard.getUSB();
  if(usb != null && "3.0".equals(usb.getVersion()){
    System.out.println("ok");
  }
}
```

We can rewrite this pattern of "checking for null and extracting" (here, the `Soundcard` object) using the `map` method.

```
Optional<USB> usb = maybeSoundcard.map(Soundcard::getUSB);
```

There's a direct parallel to the `map` method used with streams. There, you pass a function to the `map` method, which applies this function to each element of a stream. However, nothing happens if the stream is empty.

The `map` method of the `Optional` class does exactly the same: the value contained inside `Optional` is "transformed" by the function passed as an argument (here, a method reference to extract the USB port), while nothing happens if `Optional` is empty.

Finally, we can combine the `map` method with the `filter` method to reject a USB port whose version is different than 3.0:

```
maybeSoundcard.map(Soundcard::getUSB)
      .filter(usb -> "3.0".equals(usb.getVersion())
      .ifPresent(() -> System.out.println("ok"));
```

Awesome; our code is starting to look closer to the problem statement and there are no verbose null checks getting in our way!

Cascading `Optional` Objects Using the `flatMap` Method

You've seen a few patterns that can be refactored to use `Optional`. So how can we write the following code in a safe way?

```
String version = computer.getSoundcard().getUSB().getVersion();
```

Notice that all this code does is extract one object from another one, which is exactly what the `map` method is for. Earlier in the article, we changed our model so a `Computer` has an `Optional<Soundcard>` and a `Soundcard` has an `Optional<USB>`, so we should be able to write the following:

```
String version = computer.map(Computer::getSoundcard)
                  .map(Soundcard::getUSB)
                  .map(USB::getVersion)
                  .orElse("UNKNOWN");
```

Unfortunately, this code doesn't compile. Why? The variable computer is of type `Optional<Computer>`, so it is perfectly correct to call the `map` method. However, `getSoundcard()` returns an object of type `Optional<Soundcard>`. This means the result of the map operation is an object of type `Optional<Optional<Soundcard>>`. As a result, the call to `getUSB()` is invalid because the outermost `Optional` contains as its value another `Optional`, which of course doesn't support the `getUSB()` method. Figure 3 illustrates the nested `Optional` structure you would get.

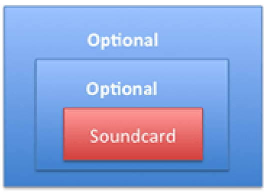

Figure 3: A two-level `Optional`

So how can we solve this problem? Again, we can look at a pattern you might have used previously with streams: the `flatMap` method. With streams, the `flatMap` method takes a function as an argument, which returns another stream. This function is applied to each element of a stream, which would result in a stream of streams. However, `flatMap` has the effect of replacing each generated stream by the contents of that stream. In other words, all the separate streams that are generated by the function get amalgamated or "flattened" into one single stream. What we want here is something similar, but we want to "flatten" a two-level `Optional` into one.

Well, here's good news: `Optional` also supports a `flatMap` method. Its purpose is to apply the transformation function on the value of an `Optional` (just like the map operation does) and then flatten the resulting two-level `Optional` into a single one. Figure 4 illustrates the difference between `map` and `flatMap` when the transformation function returns an `Optional` object.

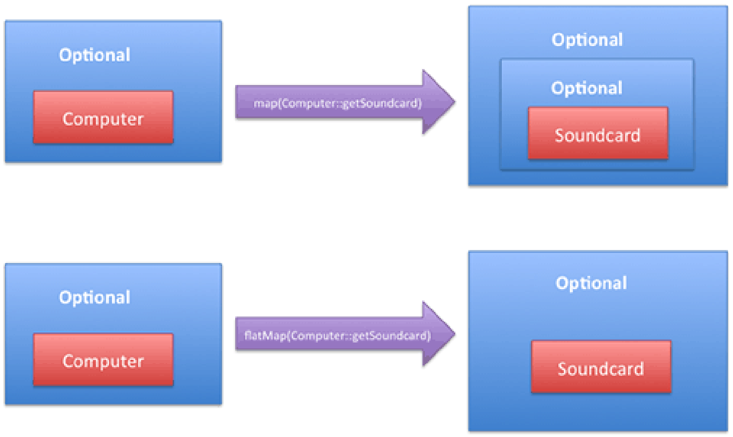

Figure 4: Using `map` versus `flatMap` with `Optional`

So, to make our code correct, we need to rewrite it as follows using `flatMap`:

```
String version = computer.flatMap(Computer::getSoundcard)
                   .flatMap(Soundcard::getUSB)
                   .map(USB::getVersion)
                   .orElse("UNKNOWN");
```

The first `flatMap` ensures that an `Optional<Soundcard>` is returned instead of an `Optional<Optional<Soundcard>>`, and the second `flatMap` achieves the same purpose to return an `Optional<USB>`. Note that the third call just needs to be a `map()` because `getVersion()` returns a `String` rather than an `Optional` object.

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.

Conclusion
In this article, we have seen how you can adopt the new Java SE 8 `java.util.Optional<T>`. The purpose of `Optional` is not to replace every single null reference in your codebase but rather to help design better APIs in which—just by reading the signature of a method—users can tell whether to expect an optional value. In addition, `Optional` forces you to actively unwrap an `Optional` to deal with the absence of a value; as a result, you protect your code against unintended null pointer exceptions.

See Also
Chapter 9, "Optional: a better alternative to null," from *Java 8 in Action: Lambdas, Streams, and Functional-style Programming*
"Monadic Java" by Mario Fusco
"Processing Data with Java SE 8 Streams"

Acknowledgments
Thanks to Alan Mycroft and Mario Fusco for going through the adventure of writing *Java 8 in Action: Lambdas, Streams, and Functional-style Programming* with me.

About the Author
Raoul-Gabriel Urma (@raoulUK) is currently completing a PhD in computer science at the University of Cambridge, where he does research in programming languages. He's a coauthor of the upcoming book *Java 8 in Action: Lambdas, Streams, and Functional-style Programming*, published by Manning. He is also a regular speaker at major Java conferences (for example, Devoxx and Fosdem) and an instructor. In addition, he has worked at several well-known companies—including Google's Python team, Oracle's Java Platform group, eBay, and Goldman Sachs—as well as for several startup projects.

Join the Conversation
Join the Java community conversation on Facebook, Twitter, and the Oracle Java Blog!

E-mail this page Printer View

ORACLE CLOUD
Learn About Oracle Cloud Computing
Get a Free Trial
Learn About DaaS
Learn About SaaS
Learn About PaaS
Learn About IaaS

JAVA
Learn About Java
Download Java for Consumers
Download Java for Developers
Java Resources for Developers
Java Cloud Service
Java Magazine

CUSTOMERS AND EVENTS
Explore and Read Customer Stories
All Oracle Events
Oracle OpenWorld
JavaOne

COMMUNITIES
Blogs
Discussion Forums
Wikis
Oracle ACEs
User Groups
Social Media Channels

SERVICES AND STORE
Log In to My Oracle Support
Training and Certification
Become a Partner
Find a Partner Solution
Purchase from the Oracle Store