Java 8 Optional Use Cases

By **Daniel Olszewski**, dzone.com
March 22nd, 2016

It's been two years since Java 8 was officially released and many excellent articles about new enhancements and related best practices have been written through that time. **Surprisingly, one of the more controversial topics amongst all the added features is the Optional class.** The type is a container, which can be either empty or contain a non-null value. Such construction reminds the user of an Optional object that the situation when there's nothing inside must be handled appropriately. Although the definition of the type on Javadoc is quite descriptive, when it comes to identifying valid use cases it's getting more problematic.

Method Result

The first possible use case is actually a no-brainer. Brian Goetz, who is working on the Java language at Oracle, stated this purpose in his answer on Stack Overflow as **the main motivator to add the type to the standard library**.

> Our intention was to provide a limited mechanism for library method return types where there needed to be a clear way to represent "no result", and using null for such was overwhelmingly likely to cause errors.

Before Java 8, receiving a null value from a method was ambiguous. It could mean there's nothing to return or that an error occurred during execution. Besides, developers tended to forget verifying if the result was null, which led to nasty NullPointerException at runtime. Optional solves both problems by providing a convenient way to force the user of the method to check its self-explanatory output.

Collection Wrapper - Bad Idea

While putting a nullable method result inside Optional is advisable, the rule doesn't apply when the output is a collection or an array. In a case when there's no element to return, **an empty instance is superior to empty Optional and null** as it conveys all necessary information. Optional doesn't provide any additional value and only complicates client code, hence it should be avoided. Here's a bad practice sample:

```
Optional<List<Item>> itemsOptional = getItems();
if (itemsOptional.isPresent()) { // do we really need this?
    itemsOptional.get().forEach(item -> {
        // process item
    });
} else {
    // the result is empty
}
```

As long as the *getItems()* method returned the unwrapped list instance, client code could get rid of one condition check and simply iterate over the collection. If we want to verify the absence of the result, each collection has the *isEmpty()* method and in the case of arrays, the length property can be used. On the whole, Optional adds unnecessary complexity.

Constructor and Method Parameters

Although it might be tempting to consider Optional for not mandatory method parameters, such a solution pale in comparison with other possible alternatives. To illustrate the problem, examine the following constructor declaration:

```
public SystemMessage(String title, String content,
Optional<Attachment> attachment) {
    // assigning field values
}
```

At first glance, it may look as a right design decision. After all, we explicitly marked the

attachment parameter as optional. However, as for calling the constructor, client code can become a little bit clumsy.

```
SystemMessage withoutAttachment = new SystemMessage("title",
"content", Optional.empty());
Attachment attachment = new Attachment();
SystemMessage withAttachment = new SystemMessage("title",
"content", Optional.ofNullable(attachment));
```

Instead of providing clarity, the factory methods of the Optional class only distract the reader. Note there's only one optional parameter, but imagine having two or three. Uncle Bob definitely wouldn't be proud of such code.

When a method can accept optional parameters, it's preferable to adopt the well-proven approach and design such case using method overloading. In the example of the *SystemMessage* class, declaring two separate constructors are superior to using Optional.

```
public SystemMessage(String title, String content) {
    this(title, content, null);
}

public SystemMessage(String title, String content,
Attachment attachment) {
    // assigning field values
}
```

That change makes client code much simpler and easier to read.

```
SystemMessage withoutAttachment = new SystemMessage("title",
"content");
Attachment attachment = new Attachment();
SystemMessage withAttachment = new SystemMessage("title",
"content", attachment);
```

POJOs - Debatable Case

POJO fields or getters are probably the most controversial candidates for Optional usage and judging by different blog posts, articles, and comments we all can agree only on one thing: another holy war has already been started. On the one hand, in the aforementioned Stack Overflow post, **Brian Goetz left no doubt the type isn't suitable for accessors**.

> I think routinely using it as a return value for getters would definitely be over-use.

What is more, Optional deliberately doesn't implement the Serializable interface, which essentially disqualifies the type as a member of any class that relies on the mechanism. For a number of developers, these two justifications are sufficient to reject the idea of Optional POJO fields.

On the other side of the coin, others started to question the previously mentioned arguments. As a matter of fact, a nullable model field isn't a rare case and considering Optional is reasonable. Stephen Colebourne, mostly known as a principal contributor of Joda-Time and the JSR-310 specification, proposed on his blog post to **keep nullable fields inside of a class but wrap them up with Optional when they leave the private scope through public getters**. Just like in the case of the method result, we want to aware other developers of the possible lack of value.

Besides memory overhead, **the main obstacle which prevents using Optional as a POJO field is support of libraries and frameworks**. Reflection is widely used to read and manipulate objects and Optional requires special treatment. For instance, the

Jackson development team has already provided an additional module that handles Optional fields while converting a POJO into JSON format. Hibernate validator also works with Optional entity fields, but in many cases you don't get support out of the box and some additional work might be inevitable.

If you choose the dark side and ignore the advice from Brian Goetz, you have to **make sure all libraries and frameworks that you use can fully deal with the Optional class**. No matter what your team decides when starting a new application, the best advice is to keep it consistent across the whole project.

Optional Class Dependency

Sometimes features or some part of business logic can be toggled on and off based on an application configuration. When such code is externalized into a separate class, it becomes an optional runtime dependency. Stateless classes don't implement the Serializable interface, so there are no technical obstacles to using Optional as a class field. Consider the following example:

```
public class OrderProcessor {

    private Optional<SmsNotifier> smsNotifier =
Optional.empty();

    public void process(final Order order) {
        // some processing logic here
        smsNotifier.ifPresent(n ->
n.sendConfirmation(order));
    }

    public void setSmsNotifier(SmsNotifier smsNotifier) {
        this.smsNotifier = Optional.ofNullable(smsNotifier);
    }

}
```

If a nullable field were used, we'd risk someone while extending the class, would unknowingly use the field without verifying its presence. **In the world dominated by dependency injection frameworks, a multitude of developers automatically assume if there's a field in a business logic class then it must be set by a container**. The Optional type is highly expressive in this situation and prevents the automatic behavior from occurring.

As usual in programming, there's no one best way to tackle the problem. Another possible solution for optional dependencies is the Null Object pattern, which in some cases may be preferable.

Not a Silver Bullet

Although in many situations the temptation to use Optional may be strong, the original idea behind the type wasn't to create an all-purpose replacement for every nullable value. Before applying the type, all possible alternatives should be considered as

overusing of Optional may lead to introducing new burdensome code smells. If you have practical experience with Optional, especially as POJO fields or getter outputs, I'd be glad to read about what you've learned. Every comment is highly appreciated, so don't hesitate to share your observations.