**JAVA8 Best Practices**

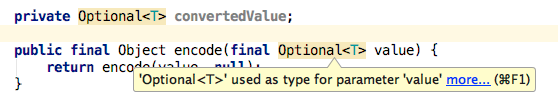
**Optional**

[Optional](http://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html) is a highly underrated feature, and one that has the potential to remove a lot of those NullPointerExceptions that can plague us. It’s particularly useful at the boundaries of code (either APIs you’re using or APIs you’re exposing) as it allows you and your calling code to reason about what to expect.

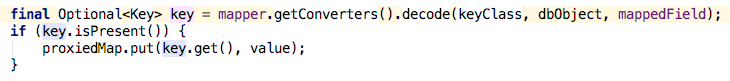
However, applying it without some thought and design can lead to one small change impacting a large number of classes, and can lead to poorer readability. Here are some tips on how to use Optional effectively.

**Optional Should Only Be Used for Return Types**

…not parameters and not fields. [Read this blog post](http://blog.joda.org/2015/08/java-se-8-optional-pragmatic-approach.html) to see a pragmatic approach to using Optional. Fortunately, IntelliJ IDEA lets you turn on an [inspection](https://www.jetbrains.com/help/idea/2016.2/code-inspection.html) to check you’re following these recommendations

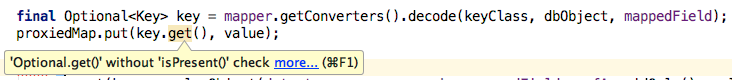
[](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/OptionalParamWarning.png)

Optional values should be dealt with in the place they’re encountered.  IntelliJ IDEA’s suggestions will prevent Optional leaking all over your code, so remember you should deal with Optional where you found it and move swiftly on.

[](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/OptionalUseImmediately.png)

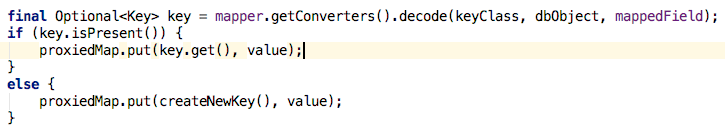
**You Should Not Simply Call get()**

The strength of Optional is to express that this value might be empty, and allow you to cater for that case. Therefore, it’s important to check if there is a value before doing anything with it.  Simply calling get() without checking isPresent() first is likely to lead to a null pointer at some point. Fortunately, once again IntelliJ IDEA has an inspection to warn you to stick to this.

[](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/OptionalGetWithoutIsPresent.png)

**There is Probably a More Elegant Way**

The isPresent() combined with get() certainly does the trick…

[](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/OptionalSimple.png)

…but there are more elegant solutions. You can use orElse to give an alternative value in the case that it’s null

[Using Optional orElse](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/OptionalOrElse.png)

…or you can use orElseGet to tell it which method to call in the case that the value is null. This might seem the same as the above example, but the supplier method will be called only if needed, so if it’s an expensive method, using the lambda will give you better performance.

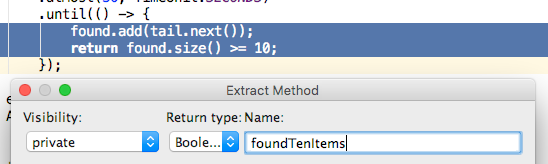
[Using Optional orElseGet](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/OptionalOrElseGet.png)

**Using Lambda Expressions**

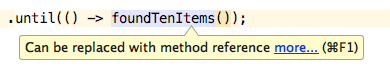
[Lambda expressions](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) were one of the main selling points of Java 8.  Even if you’re not using Java 8 yet, you’ve probably got basic understanding of them by now. But they are a new way of programming within Java, and it’s not yet obvious what is “best practice”. Here are some guidelines I like to follow.

**Keep It Short**

Functional programmers may be happy with longer lambda expressions, but those of us who’ve been using only Java for many years might find it easier to keep lambda expressions to a small number of lines. You might even prefer to limit them to just a single line, and you can easily refactor longer expressions into a method.

[](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/LambdaExtractMethod.png)

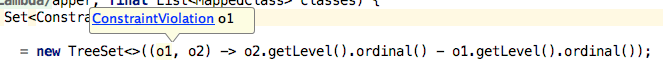
These may even collapse down into method references. Method references may seem a bit alien at first, but it’s worth persevering with them as they can actually aid readability in some cases, which I’ll talk about later.

[](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/LambdaCollapseToMethodRef.png)

**Be Explicit**

Type information is missing for lambda expressions, so you may find it useful to include the type information for the parameter.

[Explicit parameter types for lambda expressions](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/LambdaExplicitParamTypes.png)As you can see, this can get quite unwieldy.  So I prefer to give my parameters a useful name. Of course, whether you do this or not, IntelliJ IDEA lets you see type information of parameters.

[](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/LambdaParamTypes.png)

and even the Functional Interface the lambda represents:

[See the lambda&apos;s functional interface](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/LambdaFunctionalInterface.png)

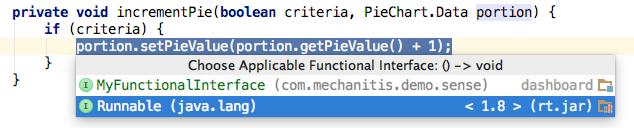
**Designing for Lambda Expressions**

I think of lambda expressions as being a little like [Generics](https://docs.oracle.com/javase/tutorial/extra/generics/index.html)– with Generics, we use them regularly (for example, adding type information to List<> ), but it’s much more rare that we’ll design a method or a class that has a Generic type (like a Person<T>). Similarly, it feels like we’ll pass lambdas around when using things like the Streams API, but it’ll be much more rare for us to create a method that requires a lambda parameter.

However, if you do find yourself in this situation, here are some top tips.

**IntelliJ IDEA Can Help You Introduce a Functional Parameter**

This lets you [create a parameter](https://www.jetbrains.com/help/idea/2016.1/extract-functional-parameter.html) where someone will pass in a lambda rather than an Object. The nice thing about this feature is that it suggests an existing [functional interface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html) that matches the specification.



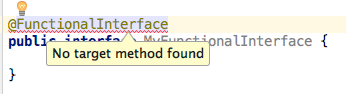
Which leads to…

**Use Existing Functional Interfaces**

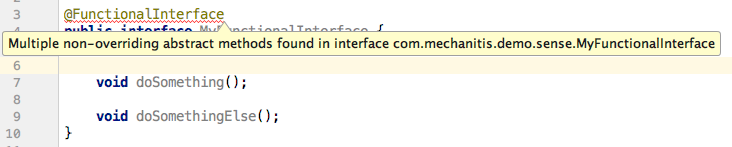
As developers become more familiar with Java 8 code, we’ll know what to expect when using interfaces like [Supplier](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html) and [Consumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html), and creating a home-grown ErrorMessageCreator (for example) could be confusing, and wasteful. Take a look at the [function package](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) to see what’s already available.

**Add @FunctionalInterface to Your Functional Interface**

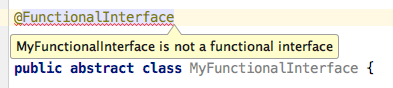
If you really do need to create your own functional interface, then tag it as such with this annotation.  It might not seem to do much, but IntelliJ IDEA will show you if your interface doesn’t match the exceptions for a functional interface. It flags when you haven’t specified a method to override:

[](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/LambdaFunctionalInterfaceNoMethod.png)

It flags when you’ve specified too many:

[](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/LambdaFunctionalInterfaceTooManyMethods.png)

And it warns you if you’ve applied it to a class rather than an interface:

[](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/LambdaFunctionalInterfaceNotInterface.png)

Lambda expressions can be used for any interface with a Single Abstract Method, but they can’t be used for abstract classes that meet the same criteria.  Seems illogical, but it is what it is.

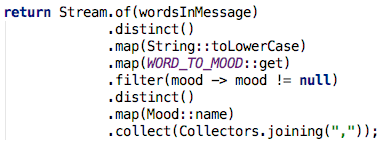
**Streams**

The [Stream API](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html) is another one of the big selling points of Java 8, and I think we still don’t really know how much this is going to change (or not) the way we code. Here’s a mixed bag of things I’ve found useful.

**Line Up the Dots**

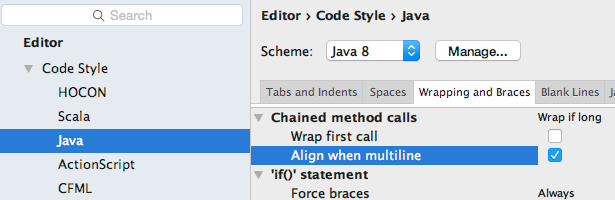
I personally prefer to line up my stream operations. You don’t have to, of course, but I’ve found this helps me:

* See which operations I have, in order, at a glance
* Debug more easily (although IntelliJ IDEA does offer the ability to [set breakpoints on any of multiple lambda expressions on a line](https://youtu.be/rimzOolGguo?t=3s), splitting them onto separate lines does make it simpler)
* Comment out an operation while I’m testing things
* Easily insert a peek() for debugging or testing



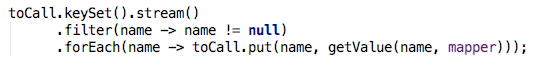
Also it’s neater, in my opinion. But we don’t gain a lot in terms of reducing lines of code if we follow this pattern.

You may need to adjust your formatting settings to get the dots lined up.

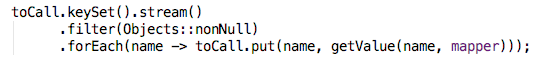


**Use Method References**

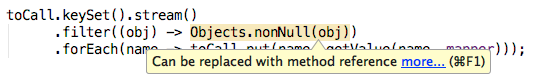
Yes, it does take a while to get used to this weird syntax. But when used correctly, it really adds to the readability. Consider:

[](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/StreamSimpleFilter.png)

Versus the use of the helper methods on the (fairly) new [Objects](https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html) class:

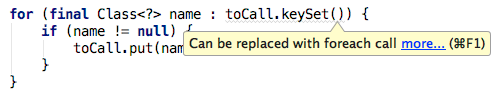
[](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/StreamMethodRefFilter.png)

This latter code is much more explicit about which values it wants to keep.  IntelliJ IDEA will usually let you know when a lambda can be collapsed to a method reference.

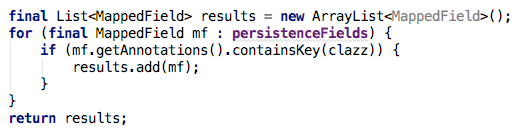


**When Iterating Over a Collection, Use the Streams API Where Possible**

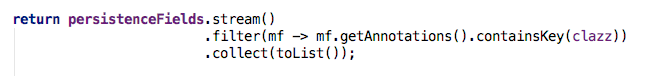
…or the new collections methods like forEach.  IntelliJ IDEA can suggest this to you:

[](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/StreamRepalceForWithForEach.png)

Generally using the Streams API is more explicit than a combination of for loops and if statements.  For example:

[](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/StreamBefore.png)

IntelliJ IDEA suggests this can be refactored to:

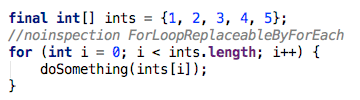
[](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/StreamAfter.png)

The performance tests I’ve done show that this sort of refactoring can be surprising – it’s not always predictable whether performance will stay the same, improve, or get worse. As always, if performance is critical in your application, measure it before committing to one style over another.

**Use for Loops When Looping Over Arrays**

However, using Java 8 doesn’t necessarily mean you have to use streams and new collections methods everywhere. IntelliJ IDEA will suggest things that can be converted to streams, but that doesn’t mean you have to say “yes” (remember [inspections can be suppressed](https://www.jetbrains.com/help/idea/2016.2/suppressing-inspections.html) or [turned off](https://www.jetbrains.com/help/idea/2016.2/disabling-and-enabling-inspections.html)).

In particular, iterating over a small array of primitive types is almost definitely going to be better performance with a for loop, and probably (at least while Java developers are new to streams) more readable.

[](https://d3nmt5vlzunoa1.cloudfront.net/idea/files/2016/07/StreamPreferArrayLoop.png)

As with any of the tips, this rule is not set in stone, but you should decide whether you are going to prefer using the Streams API where you can, or whether you’re still going to use loops for some operations.  Above all, be consistent.

**Finally**

I’m discovering new things every day, and sometimes my preferences change – method references, for example, I used to hate, and avoided having them in my code. I’d love to hear your top tips!