**iOptional: Performant Optional Interface Specification and Autogenerated Null Object Extension.**

**Abstract**   
Many object-oriented programming languages define *null* as part of their type systems. Null pointer dereference errors top the list of errors in production environments. They in part account for 40% of errors in Android apps. In this work, we introduce *iOptional*, a solution based on an optional interface specification as well as a metaprogram to proactively handle null pointer dereferences without changing the type system. Additionally, we extend our specification to autogenerate the Null Object. Implementing our specification in Lombok, a well-known open-source code-generation Java tool, we show that *iOptional* performs X times faster than the existing Java *Optional* model, consumes no additional memory and retains the *Optional* model syntax. We finally discuss mainstreaming the solution as part of the Java language.

**Background Information**

The billion dollar mistake…

Modern languages seem to be learning from that mistake

Most object-oriented programming languages support null references. The main issue with such languages is the absence of non-null language constructs at the compiler level in addition to the lack of intention to add any for one reason or another.  
For instance, the Java language follows a backwards compatibility model where introducing.

**What is compile-time generation?**

Consistency

Less error prone

Automation

Still goes thru compile-time checks

**What is the optional pattern?**

Has different names in different languages: Optional, or Option, or Maybe

It’s an encapsulation, container, wrapper…etc

Best practices, wrong usage, limitations, guidelines, consistency in code syntax

Even though java has primitive data types (i.e., value types) which may not be null at all, the type system itself isn’t unified. That is, primitive types do not implement a shared class with object types.  
  
  
**Java Optional**  
Given type X in the codebase and we want the ability to optionally refer to an instance of it. That is, some instances may at times be null. In the Java language, we use the Optional wrapper class as follows:  
 ***public Optional<X> doSomething() {  
 // … do something  
 X x = getX();  
 return Optional.ofNullable(x);  
 }***

**What is wrong with the Optional pattern?**

* A layer of abstractions, stores a new object in the heap
* Equals and hashcode
* An increase in garbage collection
* Less performant
* So many syntactic restrictions and guidelines…
  + Identity sensitive operations are unpredictable…
* When mapping objects from say domain to dto, there is lots of wrapping/unwrapping that will need to happen
* In C++ for example, every optional wrapper adds an additional total of 4 bytes to the actual object.

**How bad is Optional pattern in a distributed environment?**

In a distributed environment where instances/nodes of an application constantly communicate via means of serialization/deserialization or by accessing a distributed shared cache, there would be lots of CPU cycles wasted in wrapping/unwrapping Optional indirections.

Especially that the Optional class in java is NOT serializable…

**What is wrong with the Null Object Pattern?**

The main problem in the Null Object Pattern is the tediousness and effort involved in having to implement all the methods for every class/type where null mitigation is desirable.

**What is the benefit of a null-mitigation pattern for an organization?**

It is definitely infeasible for an organization with hundreds of applications with hundreds of thousands of lines of code suddenly to migrate to a completely new language.

**Related Work**

The concern is not with the “null” literal in and of itself, but rather the lack of compile-time checks when accessing objects in non-null-safe languages such as Java, Groovy, C++, Clojure, Objective C…etc Such shortage causes NPEs.. In other words, in languages that lack the presence of non-null types, it’s the responsibility of the developer to keep in mind where and when to make null checks as opposed to somehow delegating that responsibility to the type-system (the compiler).

Numerous studies in the literature aimed to remediate the problems of null pointer dereferencing at so many different levels.

…

….

However, most of the work attempted to approach the problem reactively. That is, they proposed schemes, algorithms, tools…etc to help detect (and patch) possible null pointers after code had been written employing techniques such as static and dynamic code analysis, code injection…etc. In this work, we propose a rather proactive approach, iOptional. It leverages the type system aiming to help prevent developers from making mistakes that lead to null pointer errors.

The main benefit of a language having non-nullable types is forcing a compile-time check before attempting to access a nullable type

**Proposed Solution**

**The Optional Model**

Given a nullable object instance of type ***Foo***,

The fundamental drawback of the Optional model is the necessity of instantiation. That is, a new Optional wrapper object instance must be created every time it’s unsure whether a given instance of ***Foo*** is null but is in actuality not.

To elaborate, consider the factory method ***ofNullable*** method which aims to create a new Optional instance. This method first checks the ***foo*** argument being passed in, if it’s null then a pre-created static empty Optional instance is returned. Otherwise, a new Optional wrapper instance is created to wrap ***foo***.

***Optional<Foo> optionalFoo = Optional.ofNullable(foo);***

Such instantiation consumes additional compute resources (i.e., memory and CPU cycles) which, in addition to other factors mentioned earlier, limit the applicability and adoption of the Optional model, especially since the state of being null is presumed to be an atypical case where a given instance of type ***Foo*** will -most of the time- not be null.

**The iOptional Model**

Our proposed iOptional specification thoroughly addresses the shortcomings of the *Optional* model and is specified as follows. **Firstly** it elevates an *iOptional* interface preserving all public functional methods of *Optional* and changing the existing wrapper model to a polymorphic model by it introducing two main implementation classes, namely: *Null* and *NotNull* see figure(…).

The latter implementation classes fulfill the *iOptional* interface contract in a mutual exclusive fashion and as we’ll show later work as the blueprint for subsequent user-defined types. For instance, Table (..) presents pseudocode for ***ifPresent*** method for Optional and IOptional implementations. We observe two main things.

|  |  |  |  |
| --- | --- | --- | --- |
| **Optional** | **IOptional** | **IOptional - NotNull** | **IOptional - Null** |
| **void** ifPresent(Consumer c){  **if** (**value** != **null**)  c.accept(**value**); } | **void** ifPresent(Consumer c); | **void** ifPresent(Consumer c){  c.accept(**this**); } | **void** ifPresent(Consumer c){ } |

The latter classes work at object *references* as opposed their Optional counterparts which work at object *instances*.. Secondly, and most importantly is the implementation of the ***ofNullable*** method

Thirdly, is replacing the getter methods with IOptiona.ofNullable(…)

|  |  |
| --- | --- |
| **Optional** | **IOptional** |
| **static** Optional<T> ofNullable(T value) {  **return** value == **null**? *empty*(): *new Optional*(value); } | **static** IOptional<T> ofNullable(IOptional<T> value) {  **return** value == **null**? *empty*(): value; } |

Diagram

Description automatically generatedDiagram

Description automatically generated

We introduce iOptional, to force null pointer checks leveraging the type system…  
*iOptional* excels at proactively performing null checks via autogenerated abstract syntax trees (ASTs).

Given a user-defined class Foo, here we show the steps to be taken in order to make it a proper IOptional implementation class:

1- Make the Foo class extend the Value class as follows:  
 ***public final class Foo extends NotNull<Foo> {  
 }***  
  
2- In class Customer Provide an ofNullable static factory method as follows:  
 ***public class Foo extends NotNull<Foo> {  
 public static IOptional<Foo> ofNullable(Foo foo) {  
 return foo == null ? Empty.EMPTY : foo;  
 }  
 }***  
  
3- If class ***Foo*** does not have any further implementations, denote it as ***final***  
  
4- Wherever in the code a Customer instance is being returned and it’s unsure whether such instance is null or not, change the return type to ***IOptional<Foo>*** and change the method’s return statement to return ***Foo.ofNullable(customer);***  
  
The next step would naturally be to automate the above steps to eliminate the tediousness of having to rewrite such boilerplate code for every single class. To setup such automation, we develop a Java Annotation Processor to scan for classes annotated with @OptionalType. Such annotation marks classes as ones for which instances maybe optional; and thus,  
1- Annotate a given java class with @OptionalType  
2-

To promote backwards compatibility, we approach the solutions as follows:

1- Code generated is source-code for two reasons:

* + Apply compile-time checks.
  + Be as less intrusive (to the original code) as possible in case of refactoring to use the newly proposed Nullable pattern.

However, it worth mentioning that it is possible to edit byte-code directly such as the stuff that Lombok does.

2- Since these languages allow null by default, all getters in the generated sources are assumed nullable unless otherwise stated. In java there is an @NonNull annotation to do so.

3- Code generation is highly customizable. For instance, one can choose to generate one or a set of getters i.e., plain old getters, optional getters and/or nullable getter…

4- Allowing any pre/post-fix of choice for the generated classes. In our case, we use Customer$ and Nullable<Customer$>.

In the proposed solution, we can also bridge to Optional

**What’s the benefit of the generated classes being marked as ”final”?**

We create the classes as “final” such that the JVM uses static dispatch for better performance

**Syntactic Sugar is possible**

We create “NullableXXX” interfaces for syntactic sugar and better readability…

**Where is it possible employ this proposed solution?**

To start off, use it in the domain model, simply due to two reasons

1- It is assumed that in real applications, most of the time, it is possible to employ Inversion of control (IoC) in almost all

**Results**

**What is the benefit of compile-time generation in light of the proposed solution?**

Most importantly, because the domain-model is generated, it’s WILL be null-safe…

**How can this solution address built-in types, or third party types which may not be modified?**

* + Create new types that copy the exact same code from the existing types, but make it inherit from Nullable<T extends Nullable<T>>
  + Provide wrappers that delegate things
  + Leave them as is, and use the primitives which don’t allow nulls, except for String
  + Don’t address the for now
  + Provide an additional getter method with a defaulted value if result is null when the field is not primitive…
  + Use Optional<String> in the getter

In any case, the worst case scenario would be equal to the original optional pattern which in this case becomes a sub-solution of the proposed one.

**What is the benefit of the “is-a” relationship over the Optional pattern’s “has-a” counterpart?**

Unlike the Optional pattern which denotes a “has-a” relationship, the proposed one denotes an “is-a” relationship. This has the following benefits:

* Because the relationship becomes a “has-a” relationship, there is no problem with returning Nullable<T> in the getters, unlike the Optional<T> pattern where there are some guidelines to prevent the “overuse”…
* Unnecessary additional memory used for optional pattern
* Object creation is cheap, but when there are several layers of nested fields, performance may worsen.
* As the depth of field nesting increases, the “overuse” of the Optional pattern gets more prevalent due to the desire to continue to wrap nullable values within an Optional (For consistency purposes)

**Conclusion**

* This newly proposed pattern does not completely solve for NPEs, but it helps stretch the null-free zone while not incurring additional cost.
* The proposed solution can be adopted as part of these languages to eliminate the need for compile-time code generation…
* The proposed work does not discuss how to refactor existing code-bases to migrate to use Nullable…
* The proposed work is only proposed for domain-models at this point.
* The proposed solution does not address lists, maps and collections in general…