Null object design pattern Vs null object check

Ask Question

Why null object design pattern is better than null object check. If we look at the memory footprint in null object design pattern we create a new dummy object of same type. Which show if we have object of big size and large number of nullable objects in search query, this pattern will create that much number of null object which will occupy more memory than a simple check which for null which my cost ignoreable delay in performance.

Null Object design pattern

java design-patterns condition nullable

edited Aug 26 '15 at 7:01

Mathias Begert
1,547 8 22

asked Aug 26 '15 at 6:37

Muhammad Nasir
915 2 22 40

3 In Java8, you can use Optional to avoid NullPointerExceptions without performing additional checks. oracle.com/technetwork/articles/java/… – Stultuske Aug 26 '15 at 6:38

what about other languages like C# and c++ – Muhammad Nasir Aug 26 '15 at 6:40

2 @MuhammadNasir the more you'll get experienced, the more you'll realize that a super-fast program that doesn't work or is full of bugs is worse than a fast enough program that works well. Everything is not about performance and memory footprint. Avoiding bugs is more important. `null` is the source of many bugs, and checking for null is often forgotten. – JB Nizet Aug 26 '15 at 6:45

@MuhammadNasir I second JB Nizet. It is not like a "A is always better than B" - thing. You'll always have to consider context. Of course in an embedded environment with very limited memory capacity, you'll look for small memory footprint. But on a modern day server that's probably not your main concern. – Fildor Aug 26 '15 at 6:48

3 personally I prefer null instead of working with wrapper objects for values. A null check of returned values should always be the case. The compiler will optimise any extra checking code for nulls anyways. You will always want to seperate a null(wether it be null or Null) into different program flows. I never let an object pass through hoops if it's null. Always check. And then it's easy to seperate program flows, give custom error messages. The overhead, and more complicated programming of working with a Null object is not worth it to me. You're gonna check for both anyways. – Tschallacka Aug 26 '15 at 6:58

4 Answers

The whole problem with `null` is that if you try to access a `null` value the application will throw a `NullPointerException` and abort.

To reduce the number of class `NullXXX` in this *null object design pattern* (its actually just the factory design dattern, not a pattern itself) you could make a `static final NullCustomer` which is always returned.

In Java 8 you can use the Optional approach in order to tell when a function does not always return values. This approach does not force you to create arbitrary null classes which pollute the overall structure (consider may have to refactor those null classes, too).

Eclipse and IntelliJ also offer compile time annotations `@Nullable`, `@NonNull` which give compiler warnings when accessing potential `null` objects. However, many frameworks are not annotated. IntelliJ therefore tries to discover those potential null accesses with static analysis. Beside low adoption of this approach IntelliJ and Eclipse use their own annotations (`org.eclipse.jdt.annotation.NonNull`, `com.intellij.annotations.NotNull`) that those are not compatible. But, you can store the annotations outside of the code which works in IntelliJ. Eclipse want to implement this in the future, too. The problem is that there are many frameworks providing this feature giving you many different annotations doing the very same. There was JSR-305 which is dormant. It'd provide an annotation in `javax`. I don't know the reason why they did not pushed this further.

edited May 23 '17 at 12:24
Community ♦
1 1

answered Aug 26 '15 at 6:49

lschuetze
1,011 7 20

Just to amend this " Eclipse want to implement this in the future, too": support for external annotations has been released via Eclipse Mars (4.5). – Stephan Herrmann Sep 4 '15 at 11:07

Another clarification: "IntelliJ and Eclipse use their own annotations (org.eclipse.jdt.annotation.NonNull, com.intellij.annotations.NotNull) that those are not compatible". There's only one thing within these annotations that can be incompatible: the `@Target`. Eclipse ships two versions, org.eclipse.jdt.annotation_1.x for 1.7 and below ("declaration annotations"), and org.eclipse.jdt.annotation_2.x for 1.8 and above ("type annotations", where the target is `TYPE_USE` for much stronger checking). Other than that, Eclipse can be configured to use any set of properly defined null annotations. – Stephan Herrmann Sep 4 '15 at 11:12

The major advantage of using Null Object rather than `null` is that using `null` you have to *repeat* checks of whether that object is indeed `null`, particularly in all methods that require that object.

In Java 8, one will have to do:

```
Object o = Objects.requireNotNull(o); //Throws NullPointerException if o is
indeed null.
```

So, if you have a method that constantly pass the same object into various method, each method will need to check that the object received is not null before using it.

So, a better approach is to have a Null Object, or `Optional` (Java 8 and higher) so that you don't need to do the null check all the time. Instead one would:

```
Object o = optional.get(); //Throws NullPointerException if internal value is
indeed null.
//Keep using o.
```

No (really) need for null checking. The fact that you have an `Optional` means that you might have a value or none.

Null Objects have no side effects because it usually does *nothing* (usually all methods is an empty method) so there is no need to worry about performance (bottlenecks/optimization/etc).

edited Aug 26 '15 at 6:53

answered Aug 26 '15 at 6:48

Buhake Sindi

67.8k 22 142 201

You still need to check if the Optional is present, and a NullPointerException is still something you want to avoid. It's just that, since it's an Optional, it's now obvious that it might be absent. – JB Nizet Aug 26 '15 at 6:51

@JB Nizet indeed. I was trying to emphasize the lack of doing `null` check everytime. – Buhake Sindi Aug 26 '15 at 6:54

With null annotations you'd specify all those methods as taking a @NonNull argument. Then the caller would perform one null check / assert on the local variable in question and pass this variable into all method calls and all is good. Plus:

this avoids potential confusion about method calls that are expected to perform some work were in effect they may be no-ops. – Stephan Herrmann Sep 4 '15 at 11:18

The main difference (and probably the advantage) of this pattern is distinctness. Think about the following method definition:

```
public static int length(String str);
```

This method calculates length of given string. But could argument be `null` ? What will the method do? Throw exception? Return 0? Return -1? We do not know.

Some partial solution can be achieved by writing good java doc. The next and a little bit better solution is using annotations JSR305 annotattion `@Nullable` or `@NotNullable` that however can be ignored by developer.

If however you are using Null object pattern (e.g. Optional of guava or java 8) your code looks like the following:

```
public static int length(Optional<String> str);
```

So developer must care about wrapping his string into Optional and therefore understands that argument can be null. Attempt to get value from Optional that contains null causes exception that does not always happen when working with regular `null` .

Obviously you are right that using this pattern causes some additional CPU and memory consumption that however are not significant in most cases.

answered Aug 26 '15 at 6:48

AlexR

94.4k 7 92 162

Null Object Pattern (en.wikipedia.org/wiki/Null_Object_pattern) is not the same as Optional. – Rodrigo Ruiz Dec 2 '16 at 19:59

Suppose you have something like this:

```
private SomeClass someField;

void someMethod() {
  // some other code
  someField.method1();
  // some other code
  someField.method2();
  // some other code
  someField.method3();
}
```

Now suppose that there are valid use cases when `someField` can be `null` and you don't want to invoke its methods, but you want to execute the other `some other code` sections of the method. You would need to implement the method as:

```
void someMethod() {
  // do something
  if (someField != null) {
    someField.method1();
  }
  // do something
  if (someField != null) {
    someField.method2();
  }
  // do something
  if (someField != null) {
    someField.method3();
  }
}
```

By using Null object with empty (no-op) methods we avoid boilerplate null checks (and the possibility to forget to add the checks for all of the occurrences).

I often find this useful in situations when something is initialized asynchronously or optionally.

edited Sep 1 '15 at 10:39

answered Aug 26 '15 at 16:45

Dragan Bozanovic

17.5k 2 18 72