Community

yegor256.com

44 Comments • Created a year ago

Why NULL is Bad?

Why NULL is Bad?

A simple example of NULL usage in Java: What is wrong with this method? It may return NULL instead of an object - that's what is wrong. NULL is a terrible practice in an object-oriented paradigm and should be avoided at all costs.

(yegor256.com)

44 Comments

Recommend 3 Share Sort by Newest

Join the discussion…

Serhij Pylypenko • 2 months ago

By the way, Null Object pseudo-pattern contradicts 'failing fast' behavior rule, because in this case program tries to figure out some plausible 'good enough' default value thus twisting things and delaying failure as much as possible

https://sourcemaking.com/refac...

• Reply • Share ›

Serhij Pylypenko • 2 months ago

Involving any reference type in method signature like
Employee getByName(String name)
automatically implies use of either object or null.

If novice programmer forgets it, the signature might have been changed to emphasize that fact
SearchResult<employee> getByName(String name)

Here method crams both possible outcomes/states (object and 'not-found' condition) into result object (similar to Optional).

class SearchResult<t>{

```
private boolean found=false; public boolean found() {return found;}
private T result=null; public T getResult() {if(result==null) throw new NoResultException();
else return result;}
public SearchResult(T r,boolean f) {found=f; result=r;}
public SearchResult() {}
}
```

I guess Java language designers skipped this option to free garbage collector of grinding on plenty of not-very-smart-and-useful-result/state-objects.

• Reply • Share ›

Serhij Pylypenko • 2 months ago

There is nothing exceptional about fact that object is not present in collection, it's an ordinary outcome, so no reason to throw exception in 'get' method.

The Null Object is simply dangerous, because it covers and masks problem, but doesn't actually solve it.
For example, older and young people should sometimes have very different medical treatment because of their age, and "average value" Employee.NOBODY may induce deadly result if leaked internal error obstructs program from finding proper person.

By the way, your phone conversation sounds just as unnatural with NOBODY as with NULL. Have you ever met NOBODY in person? Don't think this fake concept agrees with OOA/OOD.

• Reply • Share ›

Leon • 3 months ago

Null Objects sound nice in theory, and may be a more pure object oriented way of doing things, but are not so great in practice.

The nice thing about a null value is that it works with references of any type. The bad thing about null is that it doesn't have type info (an Integer null is indistinguishable from a String null reference), and it generates a runtime exception whenever you try to dereference it.

The main issue with Null Objects is that you have to write a sensible dummy implementation for each class you need it for.
One problem with that is you don't know how your dummy is going to be used, so you don't know what the desired behavior should be.
When you call a method, should it do nothing, or should it fail (exception)? When you call a getter, should it return some dummy value, or should it fail?
It will depend on the code that uses it, which you don't know if you didn't write it, so I wouldn't expose it in public interfaces.

Another problem is that some classes (such as String/Integer/enums!) are final, so you can't even create a null object for them. And having a special value null-object String or Integer also doesn't seem like a good idea, since someone might accidentally use it (and comparing by reference might fail with clustering, serialization or different classloaders.
You might wrap those in an Optional or some similar class, but this will create twice the amount of objects, putting more strain on the GC and memory.

If you just want to throw an exception in all methods/accessors of your null object, It would have been better to do that in the first place instead of returning a null object (fail fast). Exceptions may be an acceptable alternative to returning null, but you have to take into account that exceptions have a large performance cost compared to a simple null check. And if you have to wrap each call in a try-catch block, you could just as easily wrap it in an if-null check.

A naive parser of free-form XML/JSON data could call Integer.parseInt() for example, to check the data type, which would generate a zillion parse exceptions which would cost a lost of extra CPU cycles and time. I'm not saying that's a good way of doing things, just that blindly throwing exceptions may have performance consequences for users of your code in use cases you don't even know of.

• Reply • Share ›

Lambda Pool • 5 months ago

I share the vision of author, including about other topics in OO. The problem is in Java and related OO languages. We have the solution in Scala and even Groovy treats null exceptionally better than those languages. You have a lot of problem when for example, trying some unit tests and all of those branches without test because of, well, NULL CHECKS.
You are 100% right from my opinion, null checks are a big crap.

1 • Reply • Share ›

Elbow Tard Lambda Pool • 5 months ago

YEAH, get another vodka and go to bed, will you?
You did not even understand the problem here, ok?

• Reply • Share ›

Lambda Pool Elbow Tard • 5 months ago

come with argument and we can debate. you dont have a point here only bullshit.

4 • Reply • Share ›

Richardmsiska • 5 months ago

If I understand correctly, I think what Jeffery is trying to say is that by using null objects the code reverts to using a default object as a place holder. Let us say that I have a date application and something goes wrong in my code, the default object could return today's date or earliest date recorded by the class. However, there is also an argument of this could be easily done by throwing an exception and returning the default object easily.

I prefer not to throw nulls around when I can, in a language like php it can become a pain to check for every null value every time you pass it into a method. I do however find nulls to be beneficial when working with custom data structures.

If(tree.left==null);

The above statements makes a lot of sense to me as these nodes are just references. If the tree is referencing nothing then it should surely point to null.

Another thing is that, the null object wouldn't solve anything in terms of checking. If a default value is returned the program would still have to check for default objects. I would say it does decrease the chance of your application blowing up due to less null pointer exceptions but a competent programmer would add a try and catch to handle that.

I am a bit 50/50, I like the idea of a default object mostly because it decreases the number of null pointer exceptions in my application but at times they just make the most sense.

1 ^ | v • Reply • Share ›

Daniele Maccari • 6 months ago

It seems to me you're expecting your objects to be more stupid than it needs to be. Why would they not check for 'empty' return values? That's perfectly reasonable under most circumstances, provided you don't litter your code with null checking and error handling all the way around (if that's feasible, that's it: sometimes things just get complex, get over it). You're being very dogmatic at best, and the null object pattern really doesn't change a bit in your example, unless you're fine with the returned employee to possibly be some void shell which may or may not perform the operations you expect it to correctly. In fact, following the fail fast rule, if not finding the employee actually has to be an error, you're better off knowing it with some exception or special return value rather than ignoring it until some later time.

Of course there are cases where always returning an object, even if it simply wraps a null, makes perfect sense, but I argue in those cases you really don't care about any particular method your 'real' object implements and your fake one does not. If you care about objects existing, then you should check for their existence; it's really all there is to it.

^ | v • Reply • Share ›

Jcl • 6 months ago

Using null objects is as bad (or as good) as using null. It might even cause more conceptual problems... if I need an explicit object, why would some other object which is not the one I requested do anything? In this case, there's no need to return null or a null object... if you don't want to use nulls, then you can just raise an exception on when getting the object. But then you are using exceptions to control the flow of a program.

I know there's discussion around this, but I don't like using exceptions for error handling... they are called exceptions because they should be an exception (otherwise they'd be called "errors"). If I request customer "john", the fact that there is no customer named john is perfectly valid (although I'd need to raise an error), and should not be an exception. An exception would be requesting the customer "john" and the computer having no memory to find any users, or the storage I'm requesting the customer john from is not available due to a network shutdown or a disk failure.

I find the use of NULL perfectly appropiate... and if you want the code readable, there should be something like: if(CustomerExists("John")) RetrieveCustomer("John"). If you argue that would probably iterate twice (I've seen you complaining this pattern on Java Map on some other article), then your customer service object should maintain a weaklyreferenced internal cache, and cache the customer if found to be able to retrieve it without enumerating when retrieving the customer after asking if it exists. Using this pattern, the usage of exceptions remain... but they should be, as it name implies, exceptions, not flow instructions.

1 • Reply • Share ›

Yegor Bugayenko author ➦ Jcl • 6 months ago

NULL object doesn't mean that the object is acting like NULL. It means that the object implements its methods in a different and a more primitive way. For example, a NULL employee may ignore a "raise" request. No exception, just ignore.

• Reply • Share ›

Martin • 8 months ago

In some languages there's a third (and arguably better) alternative to exceptions or the null object pattern: the optional pattern. Java recently got a type Optional of type T added to its standard library, inspired by the Option-type of Scala (in turn inspired by the Maybe-type of Haskell).

Option types improve upon the null object pattern by explicitly encoding in the type system that a function may not return a value, but also by allowing the resolution of the value to be conveniently postponed. See http://en.wikipedia.org/wiki/O...

1 • Reply • Share ›

Yegor Bugayenko author ➦ Martin • 8 months ago

Even though Optional looks convenient from a "computer thinking" perspective, it doesn't make any sense from an "object thinking" perspective. As in the example above, when I'm calling and asking for Jeffrey, I don't want to get an "optional" Jeffrey. It is just counter-intuitive. I want to get someone who can help me. Either Jeffrey or someone pretending to be Jeffrey (NULL Object). But I don't want to get something that I should ask again -- "do you have Jeffrey inside you?".

Think as an object not as a computer programmer moving bits and bytes :)

• Reply • Share ›

Martin ➦ Yegor Bugayenko • 8 months ago

Actually I think it's the most intuitive option (no pun intended) available: You're asking for a Jeffrey, but you're not guaranteed to have one back. To put this into types would mean that you need something to differ a successful call (returning an Employee) from a non-successful one.

If you're returning a null-object you're essentially returning the empty husk of an Employee; you treat it as Jeffrey when it's in fact not. Only when you examine it you see that you didn't get what you were asking for, but the type system doesn't help you reach this conclusion; a user would have to know where to look, in this case the test for id == 0.

A return type of Optional<employee> lets any API-user know that they may in fact not receive an Employee, even without looking at the source code or any ad-hoc documentation. This is a powerful concept on its own but many languages take it one step further.

(As I am most used to the Scala version of Optional my example will be based around that, but the Java variety should be very similar.)

In Scala you would work on Optional data in a fail-safe manner without actually knowing if your call returned an existing Employee or not. It is very uncommon to if-check properties on the Option, you just work over it like a fancy list of zero or one element. Only when you need an actual Employee you "realise" the object by calling optionalEmployee.getOrElse(new Employee("Patrick Bateman")). If you prefer an exception to a default user, you use something like optionalEmployee.getOrThrow(new EmployeeNotFoundException("No such employee found")).

Note that this is all in the hands of the user of the API. The user decides when to realise/fail the Employee, not the designer. Also, it cannot be stressed enough, the user needs no former knowledge of the API to access the object correctly, as the state of existence is moved to the type-level.

• Reply • Share ›

Yegor Bugayenko author Martin • 8 months ago

I understand your reasoning, it makes sense, but let me try to convince you again. All that you said above is very effective from a programmer's point of view, who wants to stay in full control over the object returned to him. I believe, in object oriented programming, we should aim for something opposite. We want to decouple the code we're working with from all its dependencies, as much as possible. And this decoupling can be done by abstraction. Here is an example of the approach you're suggesting (if I got it right):

```
Optional<employee> optionalEmployee = office.call("Jeffrey");
if (optionalEmployee.getOrElse(new FakeEmployee("Patrick")).areYo
  System.out.println("my friend is happy!")
}
```

In this code, the responsibility for areYouHappy() behavior implementation is spread between three classes: the original Employee, FakeEmployee and our own code that is doing that IF statement. The code is tightly coupled, which is what we want to avoid in OOP. Our code is making the decision it should **not** make. It is not our responsibility to decide how employees behave. We should let **them** decide. Here is the code that is much more decoupled:

```
Employee employee = office.call("Jeffrey");
if (employee.areYouHappy()) {
  System.out.println("my friend is happy!");
}
```

In this example, we let the office decide who to return. And we let the employee decide, is he/she happy or not. Makes sense?

• Reply • Share ›

Martin Yegor Bugayenko • 7 months ago

But you're explicitly asking for Jeffrey. It makes no sense to get back whatever employee or fake available. When a method can guarantee you get a valid object back there is no need to use an Option.

You are actually not using the null object pattern in your last code example as you simply assume it is valid and use it to draw conclusions about the object.

Would the null object return true/false for areYouHappy or throw an exception? In the former case, why is the case "no employee available" considered a happy/unhappy employee? On the other hand, if you throw an exception from this method in the null object, you have again created a null reference exception in new clothes (NullObjectEmployeeDoesntHaveFeelingsException)

Again, these two unsatisfactory scenarios vanish when you use an Option. It does not have as much to do with forcing responsibility on the user as it does being honest and explicit with the possibility of a missing value. If there can't be a missing value, or if there is a sensible default value, you don't return an Option, just like you would not return a null object in this case.

1 • Reply • Share ›

Jcl Martin • 6 months ago

This "If there can't be a missing value, or if there is a sensible default value, you don't return an Option, just like you would not return a null object in this case." sums it up to me. In case there should never be a missing value from the original query, then the fact that there is IS an exception, and a exception should be risen. In case the query might return an absence of a result, then you need to get back an absence of a result (in this case, called null, but you could call it "Optional<>" if you want, or anything else) and have your program act accordingly. You shouldn't get a mock or fake object of your original request which may throw exceptions on usage (or even worse, not throw them and act as if it was the object you were requesting): that is NOT what you were requesting. It's not "object-thinking" or whatever the author wants to call it: it's a function returning something you haven't requested it to return, and that is wrong at many levels.

1 • Reply • Share ›

PaulMScully . ➦ Martin • 7 months ago

I'm with you here, Martin. NullObject is a poor solution. Null (or option) expresses the actuality of the query result.

It seems to me that "null is bad" arguments usually are just disguised "crashes are bad" arguments.

Yes, crashes are bad, and a program that allows itself to crash is poorly designed =)

Don't get your code into that situation unless something exceptional has happened! If you expect the result of the query might be a "null", then either a) why are you asking such a "stupid" question in the first place, or why are you not handling the fully expected null answer?

1 ^ | ˅ • Reply • Share ›

Jcl ➦ PaulMScully . • 6 months ago

Fully agreed

^ | ˅ • Reply • Share ›

Invisible Arrow ➦ Jcl • 6 months ago

One problem I see with Option<> is that something might be optional today, but in future becomes mandatory (pretty common in any kind of software where rules change).
But the code still remains saying I'll get an Option<> value which can be misleading to another developer looking at it. Is it really optional or not? With NullObject pattern, this little detail is hidden (abstracted out) which in my opinion is a good thing from a developer's perspective.

^ | ˅ • Reply • Share ›

Karol Stasiak ➦ Martin • 8 months ago

Also, optionals fix the hole in the type system: a reference to an Employee is now guaranteed to be a valid one, it no longer can be null or point to a stubbed null object.

^ | ˅ • Reply • Share ›

Yegor Bugayenko author ➦ Karol Stasiak • 8 months ago

As I replied above, this may look convenient from a computer perspective. From an "object thinking" point of view this "optional" employee reveals implementation details to me. I was hoping to talk to Jeffrey. You're giving me something that is not a human, that can't talk to me and that I should ask "do you have Jeffrey somewhere there inside you?"

It is counter intuitive, if you think in terms of objects, not bytes, bits, NULLs and pointers.

^ | ˅ • Reply • Share ›

• Reply • Share ›

Karol Stasiak ↪ Yegor Bugayenko • 8 months ago

"Optional Employee" means that it is either "an Employee" or "Nothing", and you are forced to explicitly handle both cases at some point, be it immediately or later. These cases are explicitly separated, not conflated into this "Schrödinger's Employee" (that may exist or not) most OO languages offer.

• Reply • Share ›

Yegor Bugayenko author ↪ Karol Stasiak • 8 months ago

"you are forced to explicitly handle both cases at some point, be it immediately or later" - not exactly. You're not expected to handle these cases in your code. You may want to do this, if you get runtime exceptions. In that case you will see that some Employees are not good enough for the questions you're asking. Some of them are not capable of doing what you need. But maybe you don't need that features in your code. The point is that you should **not** assume anything about the objects you're working with. They should know how they work and fail when they can't work.

• Reply • Share ›

Massimiliano Tomassi ↪ Yegor Bugayenko • 8 months ago

If you are "searching" for something, using Optional as a return type allows you to explicitly say "hey, be aware that what you're looking for might not be available". If it's not an unexpected case, a method might well decide to return an "absent" value instead of a default null object. Surely depends on the use case, but I personally think that Optional can make sense even in the "object thinking" point of view.

1 • Reply • Share ›

Yegor Bugayenko author ↪ Massimiliano Tomassi • 8 months ago

Semantically that "Optional" design pattern is wrong. Because, as I mentioned above, it forces you to think as computer not as an anthropomorphized object. Let's try to translate this code, into English:

```
Optional<employee> jeffrey = office.call("Jeffrey, please");
if (employee.isPresent()) {
  System.out.println("Jeffrey's salary is " + jeffrey.salary());
}
```

I'm reading it like this:

```
- Hello, may I talk to Jeffrey please?
- Speaking...
- Are you really an employee or a fake?
- I'm an employee, I swear!
```

> ```
> - OK, what is your salary?
> ```

Does it make sense? :)

^ | ⌄ • Reply • Share ›

Jcl ➦ Yegor Bugayenko • 6 months ago

With your "object thinking" scenario, depending on the object implementation that could be:

- Hello, may I talk to Jeffrey please?
- Some employee that may or may not be Jeffrey speaking
- Are you really an employee or a fake?
- I'm an employee!
- Ok, what's your salary?
- $50K
- Ok, let's transfer you $50K

Meanwhile, somewhere else, the real Jeffrey:
- WTF? My salary is $100K !

Unless you triple-check the employee is really Jeffrey... with the "office.call()" method you never know that the result you got back is really Jeffrey, it all depends on the implementation of the object and your usage of the returned object afterwards.

Having office.call return an empty object (or throwing an exception if requesting a non-existing employee is something that should never be done on "office") only forces you to check if the call succeeded (be it by checking against null, or be it by handling the exception).

You are making everything verbose, complicated, and hugely implementation dependant, which makes it in turn unmainteinable (unless it's only you on the programming team, and you made everything from scratch), or extremely hard to document and follow by just reading the code. Your "object thinking" is defeating the whole purpose of making your code readable and understandable in first place, which is what you seem to aim on your articles.

^ | ⌄ • Reply • Share ›

Karol Stasiak ➦ Yegor Bugayenko • 7 months ago

You misunderstood that piece of code. It should be read as:

- Hello, may I talk to Jeffrey please?
- He may be not available, let me check.
- So, is he present?
- Yes, he is, I'm redirecting the call
- Hi Jeffrey, what is your salary?

Meanwhile, using the null object pattern leads to:

- Hello, may I talk to Jeffrey please.
- Speaking.
- What is you salary?
- Mwahaha I'm not Jeffrey, I'm not even an employee. I'm NULL!

4 ^ | ˅ • Reply • Share ›

Martin ➦ Karol Stasiak • 5 months ago

It's funny because it's true.

^ | ˅ • Reply • Share ›

Massimiliano Tomassi ➦ Yegor Bugayenko • 8 months ago

Well, in my opinion it really depends on the context. If you call a service that will pass your call to an employee based on the name you ask, that service may well reply with "Sorry that employee is not available" (Optional.absent), instead of passing you a fake employee (null object) that will try to help you even if it's not the original.

A more pratical example: if I'm calling a "store service" in order to obtain a value object based on its id and I ask for a not existent id, I don't want to receive an object that is not able to give me the information I need, I just want to know that the object I'm looking for doesn't exist (Optional.absent). As an alternative the service might throw a ResourceNotFoundException, but this really depends if asking for a not existent object is an "exceptional case".

Summarizing, returning a Null Object, an Optional or throwing an exception are, in my opinion, all reasonable solutions. Choosing which one is better really depends on the use case we're trying to model.

Obviously, I'm keen to hear different opinions :)

^ | ˅ • Reply • Share ›

Roland Bouman • 8 months ago

I don't really see how the "NULL Object design pattern" solves anything in this respect. At least I can't think of a practical case where one would want to continue operation in the usual manner if the "special" NULL object would be returned. The "advantage" would be that no NULL pointer would occur, but instead the program gets to continue basically doing nonsense operations (or rather, doing valid operations on a nonsense object). Of course, one could explicitly check if the NULL object is returned, but I don't see how that is any better than checking if the method returns null.

I'm curious if someone has a real-world example showing the benefit of the NULL Object design pattern. Thanks in advance.

EDIT: come to think of it, I'm also unsure how the iterator solution for Map is an improvement. How is it better to check for "hasNext()" as opposed to checking for == null? The complexity of the flow is the same (one if required), and the iterator solution requires

The complexity of the flow is the same (one if required), and the iterator solution requires one extra type (the iterator) and a method call to achieve essentially the same thing.

5 ^ | ˅ • Reply • Share ›

IngeSpring ➦ Roland Bouman • 6 months ago

@Roland: Exactly my thought too.
Or in "object oriented thinking":

If you send someone into a room to get a chair, and the room is empty, you don't want an imaginary chair returned, or a mockup you cant use, you simply want a message saying that the room was empty.

^ | ˅ • Reply • Share ›

Yegor Bugayenko author ➦ Roland Bouman • 8 months ago

NULL Object not necessarily is an object that throws exceptions whenever anyone touches it. This may be an object that does something, but fails to do something else. Very often I don't need to exploit all behavior of an object.

And here is a practical example of a NULL Object (in Java):

```
interface Day {
  Day NEVER = new Day() {
    @Override
    public long milliseconds() {
      throws new UnsupportedOperationException(
        "this day will never occur"
      );
    }
  };
  long milliseconds();
}
```

It is an interface of a calendar day. It has a NULL Object pattern implementation, which we can use like this:

```
Day day = when_will_this_project_be_finished();
if (day.equals(Day.NEVER)) {
  throw new Exception("hey, what do you mean never??");
}
```

Make sense?

^ | ˅ • Reply • Share ›

Karol Stasiak ➦ Yegor Bugayenko • 8 months ago

This is actually a case of bad design. "Never" is not an actual "Date", and if you let this kind of object into the type system, suddenly you may see an attempt to find a time period between two nevers...

In this case, "Day", or "Date" (whatever) should always be an actual date

In this case, "Day", or "Date" (whatever) should always be an actual date, and when_will_this_project_be_finished() should return Option<day> or you could roll a new type for EstimatedDayOfArrival or something.

In your example, both assuming that the programmer will check against NEVER when necessary and won't perform the check paranoidally when unnecessary, is wishful thinking.

For example, a function first_wednesday(int year, int month) cannot return NEVER. Assuming that such a different object like NEVER can be of the same type as the result of first_wednesday(...) leads to very brittle code.

2 • Reply • Share ›

Yegor Bugayenko author Karol Stasiak • 8 months ago

"leads to very brittle code", which is exactly what we want! :) The code has to be as brittle as possible. That's what Fail Fast is about: http://martinfowler.com/ieeeSo...

We want our code to be brittle and fail on every incorrect attempt to use it. In my example, like you mentioned, if someone will try to calculate the difference between two days and one of them is NEVER, we'll have a runtime exception. That will be an indicator that this comparison can't be successful always. It needs additional logic path implemented.

Again, brittle code is a merit, not a disadvantage.

• Reply • Share ›

Martin Yegor Bugayenko • 7 months ago

I would argue at best it is an advantage only in dynamically typed languages where you don't have the compiler to sort things out for you.

Just like Karol Stasiak mentions in his comment it is infinitely better to catch these errors during compile time. That way, you won't be fooled into believing your system is stable just because it compiles, which is a dangerous effect of runtime exceptions.

1 • Reply • Share ›

Karol Stasiak Yegor Bugayenko • 7 months ago

I don't know why you think runtime exceptions are better than a compile error. By "brittle code", yes, I meant code that breaks at runtime, but what's better is code that has been proven by the compiler to be correct.

"Never" is not a date, so it shouldn't be of the same type as all actual dates.

2 • Reply • Share ›

Roland Bouman ➦ Yegor Bugayenko • 8 months ago

Thanks for the reply. I guess I'm going to have to ruminate on it for a bit.

• Reply • Share ›

Sean Mitchell • 9 months ago

This is a very dogmatic view. Null is not bad... it is more fair to say null is often misused. I'd much rather return null in a find() method and document this than force the caller to catch exceptions.

I tend to return null (and document that I will) in cases where a reasonable input has no output. I throw an execption when the input is unreasonable, or when something unexpected happens that prevents me from fulfilling the request. Pretty much everyone understands this paradigm. And while NPEs can be a problem... well... an uncaught RuntimeException is no different. Or do we want to open the checked vs unchecked exceptions can of worms?

I'm not a fan of returning null collections, in general. Occasionally it makes sense, but usually not.

The bottom line here is that we need to be careful about making sweeping statements about "X is Bad" or "always use Y". Certainly I would agree that it's worth very careful consideration about when you want to use null and when you should not.

4 • Reply • Share ›

Gregory Pakosz • a year ago

Call me pedant but... Why are you insisting on "NULL" + Java code based argumentation while it should be "null" (which you even use here and there)?

1 • Reply • Share ›

Yegor Bugayenko author ➦ Gregory Pakosz • a year ago

Good question :) I think it's a tradition that came from C/C++, where NULL is a macro, and C macros are all-capitals, but convention. Wikipedia is also using all-capital version: http://en.wikipedia.org/wiki/N...

• Reply • Share ›

Michael • a year ago

Don't necesary agree that Java Map has any design flaws You can replace this code :

```
if (!employees.containsKey("Jeffrey")) { // first search
throw new EmployeeNotFoundException();
}
return employees.get("Jeffrey"); // second search
```

with this one :

More from yegor256.com

yegor256.com recently published

There Can Be Only One Primary Constructor

33 Comments Recommend

yegor256.com recently published

Three Things I Expect From a Software Architect

23 Comments Recommend

yegor256.com recently published

How to Avoid a Software Outsourcing Disaster

8 Comments Recommend