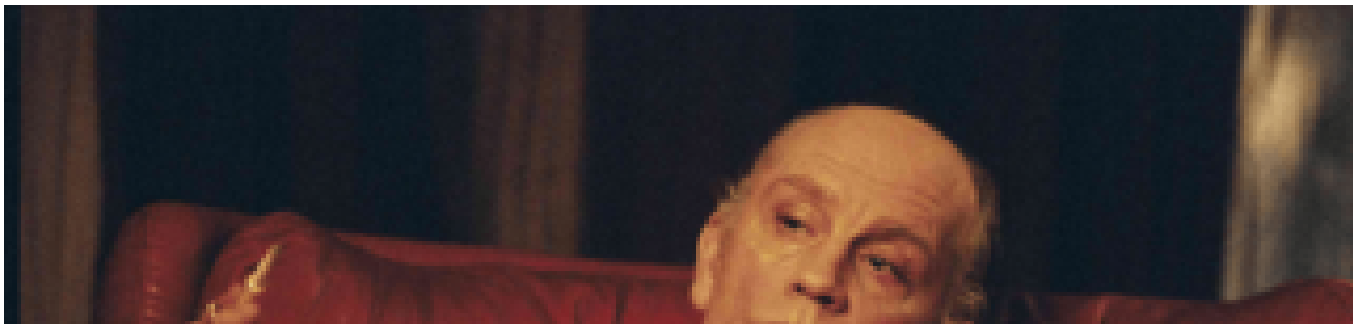
Community

yegor256.com

52 Comments • Created 3 months ago



Utility Classes Have Nothing to Do With Functional Programming

I was recently accused of being against functional programming because I call utility classes an anti-pattern. That's absolutely wrong! Well, I do consider them a terrible anti-pattern, but they have nothing to do with functional programming…

(yegor256.com)

52 Comments

Loup Vaillant • a month ago

You accuse static methods of being imperative, then you provide an example on one that is purely functional (no side effects and all that). Worse, you say that calling this very static method to initialise a variable is "imperative and procedural". What the holy fuck? This example could be transliterated in Haskell without using the IO type!

You are contradicting yourself.

1 • Reply • Share ›

Yegor Bugayenko author ➚ Loup Vaillant • a month ago

You put together "functional" and "procedural". These are two different things :)

• Reply • Share ›

David Raab ➚ Yegor Bugayenko • a month ago

No, you make a difference where no difference exists. Any static method that just have an input and some output without side-effects is automatically considered as a pure-function. You also make the mistake that you think that every static method is automatically procedural, what is also not the case. Procedural programing means asking for data, evaluating the data, and then based on the evaluation do something. Whether you do this, or not, is not something based on the fact if you use static methods or not. With static methods you can program procedural, object-oriented and/or functional.

Paradigms like OO, functional or procedural are ideas, not semantics. That is also the reason why you still can program procedural with classes and methods. You focus too much on semantics and think as soon as you see a static method that it is automatically procedural, what is absolutely not the case.

In fact even all of your Java code that you have written is 100% functional and are pure-functions. There isn't even a difference between the Lisp code you provided. You focus too much on semantics.

Reply • Share ›

Marcos Douglas Santos ➚ David Raab • a month ago

I share the concept that an object-oriented design must work only under contract (interface-type variables). If you use concrete objects with static methods is much more difficult to test in addition to being (mostly) not thread-safe.
However, in some cases, static methods could be good... maybe for performance or simplicity.

Reply • Share ›

David Raab ➚ Marcos Douglas Santos • a month ago

Actually it is not harder to test. And thread-safe has nothing to do whether you choose a static method or not. If something is thread-safe depends what you did. Pure-functions by the way are always thread-safe. But in general when we talk about thread-safe and parallel programing then functional programing is far better suited for this as object orientation.

Reply • Share ›

Marcos Douglas Santos ➚ David Raab • a month ago

> Actually it is not harder to test.

You can't mock or create some fake object to test. You need to test the real static method. Other problem is dependencies of this method inside it (context, other objects, etc)

> And thread-safe has nothing to do whether you choose a static

> And thread-safe has nothing to do whether you choose a static method or not. If something is thread-safe depends what you did.

Yes. Because that I wrote _mostly_ :)

> Pure-functions by the way are always thread-safe. But in general when we talk about thread-safe and parallel programing then functional programing is far better suited for this as object orientation.

True in real functional languages. But I'm not a functional programmer, so I can't debate appropriately if we will talking about functional vs OOP languages.

David Raab ➚ Marcos Douglas Santos • a month ago

> True in real functional languages. But I'm not a functional programmer, so I can't debate appropriately if we will talking about functional vs OOP languages.

By the way, i saw something and didn't answer correctly. Pure-functions are also always thread-safe in an OO language. "Pure-function" means.

1) There are no side-effects
2) You always get the same response if you pass in the same arguments

That is "by definition" always thread-safe, in every language and in every paradigm from procedural, OO or functional.

David Raab ➚ Marcos Douglas Santos • a month ago

> You can't mock or create some fake object to test. You need to test the real static method. Other problem is dependencies of this method inside it (context, other objects, etc)

You probably meant "You can mock...". But by the way. Mocking exists to not test something and instead provide something "fake". And by the way.

1) You also can mock static functions
2) You don't need to mock anything. In OO you use "Dependency Injection" to inject the behaviour. The functional way is to expect a function that gets executed. And because your functions expect functions you always can pass some "fake/mock" things already to it. If you want to mock anything, you are just doing it wrong. And

it. If you want to mock anything, you are just doing it wrong. And sure, also with static methods you need to learn how to write good functions. Nearly the same rules for a good class are also true for a good function.

> Yes. Because that I wrote _mostly_ :)

Not even "mostly". Using a static functions absolutely doesn't change thread-safety at all compared to a class.

> True in real functional languages. But I'm not a functional programmer, so I can't debate appropriately if we will talking about functional vs OOP languages.

Nearly every language today has functional features and implements more and more features. Passing functions as a parameter and creating lambdas works since C# 3. I think since Java 7/8 this is also possible there. And nearly every other language also has the same features. From Python, Perl, Ruby, JavaScript, C/C++, ...

And what i described so far also works in Java. The statement that static functions are not thread-safe, harder to test or you can't have something like Dependency Injection is not true at all. You can do all of that, you just need to learn "how". And well even if you don't knew functional programing i already provided code that did what you did with your interface.

But if you don't knew functional programing i only can insists you to start learning it. After that you will hate OO. ;)

Marcos Douglas Santos ➚ David Raab • a month ago

> You probably meant "You can mock...". But by the way. Mocking exists to not test something and instead provide something "fake". And by the way.

In fact I never used "mock objects", only fake objects so, by the way.

> In OO you use "Dependency Injection" to inject the behaviour.

I do not agree. DI is a way to use, but in that case I agree with Yegor: DI is an anti-pattern.

> The functional way is to expect a function that gets executed. And because your functions expect functions you always can pass some "fake/mock" things already to it. If you want to mock anything, you are just doing it wrong. And sure, also with static

> methods you need to learn how to write good functions. Nearly the same rules for a good class are also true for a good function.

---

see more

• Reply • Share ›

David Raab ➚ Marcos Douglas Santos • a month ago

> I do not agree. DI is a way to use, but in that case I agree with Yegor: DI is an anti-pattern.

Huh? The whole example that you provided was Dependency Injection. If you don't want a specific implementation and instead just want an Interface then you already have DI. Because instead that your class/method directly depends on a specific implementation it depends on an interface. And the concrete class gets injected by the caller.

> Using OOP you need to pass fake objects when do you have dependencies, for example: if you have an object that do something (and you want to test this something) but the context needs a HTTP request, you could passe a fake HTTP object to simulate this dependency.

Actually the whole problem doesn't differ whether you use OO or functional. The concept and problems behind it are exactly the same. Lets say you have an OO solution. You want to fetch some data lets say from Facebook and you get a result back that you somehow parse and you return an "Person" object. The first naive design would be you create a class that do the HTTP request, parse it, and return the object all in a single class. The way how you use it, is the following

```
var facebook = new FaceBookAPI();
var person = facebook.GetPerson("Foo");
```

This code now has the problem that the HTTP Request is encapsulated inside the FaceBookAPI class. You also lost the ability to test the class without doing a real HTTP Request. You can start to Mock the class that do the HTTP Request, but i consider that just as ugly. A better way would be to use Dependency Injection. Instead that your class silently do the HTTP Request and fetches the data and parse it, you open the ability to pass a specific instance that does the HTTP Request. For example it could look like

```
var http = new HttpRequest();
var facebook = new FaceBookAPI(http);
var person = facebook.GetPerson("Foo");
```

Now FaceBookAPI depends on an interface instead of an implementation. Instead of mocking you also now can create your FakeHttpRequest Object to test your class, that instead of doing a real HTTP Request over the network just returns the data.

```
var http = new FakeHttpRequest();
var facebook = new FaceBookAPI(http);
var person = facebook.GetPerson("Foo");
```

Now the point is, that is also exactly the same how it works with static functions. You can do the same mistakes, and you can do the same things to solve it. You for example could create a static method that returns you directly a Person

```
var person = GetFaceBookPerson("Foo");
```

If you have a function like that, you have the same problems. The function internal uses and creates the HTTP Request, and you have the same problem. You either need to mock it, and the function is somehow "hard-weired" with the HTTP request. Now OO guys often things it stops here, but it doesn't. Such functions are not good, the same way how the first OO version is bad. But it is not the problem of the static function that you have those problems, it is those of the programer that doesn't knew how to solve it. In fact you solve it the exactly the same way, how you do it the OO way. Instead of doing the HTTP Request directly inside your Function. You create a function that needs another function as his first parameter that returns the HTTP Request result. So your function Definition in C# would look like this.

```
static GetFaceBookPerson(Func<string> http, string person) {
    ....
    var requestData = http(person);
    ....
}
```

Now you also can replace the HTTP Request by any other definition whatever you want. You can for example call it like this

```
var person = GetFaceBookPerson(HTTPRequest, "foo");
```

In this case "HTTPRequest" is another static function. The logic to fetch your data is delegated to this function that was passed in. Exactly the same as the request was delegated to an object that implemented some "IHTTPRequest" interface in the OO version. How can you test it without doing a real I/O request? You pass another function that just does whatever it wants to do. As long as these function returns a string (because the passed in function was defined to return a string) you can pass any function to it. For Tests

defined to return a string) you can pass any function to it. For tests you now also can do

```
var person = GetFaceBookPerson(FakeHTTPRequest, "foo");
```

here "FakeHTTPRequest" is another static function that just returns the data, directly from memory, or whatever you want. And not only that. You also can pass directly an anoymous function/lambda to it, if you want you also can do this.

```
var person = GetFaceBookPerson(person => "{PersonName: \"Foo\"}",
```

So you directly pass in a function that just returns your fake data, and you now can test if the parsing of the String and converting to a Person object is correct, without doing any I/O at all.

If you can't test a static method, it is just the fault of the programer, not that it is not possible. And those techniques of providing the behaviour and inject the behaviour from outside even exists longer than the OO paradigm itself exists. And this technique is widely used. In fact if you learn a functional language, the whole idea what i described here is the fundamental and basic of any functional language. And not just functional languages. All of that stuff what i showed was C# code and directly works there. And it should also nearly 1:1 also work in Java.

And not just C# and Java, you also can use functions like that in Perl, Python, C++, JavaScript, ...

I don't even knew if any new language that comes out can't do this sort of things. But if it can't do that, it is probably not worthy to use such a language.

> I know what is function programming... I only said I'm not a functional programmer! Function and pointer to a function always existed in Pascal and continues existing in Object Pascal but today I'm still think OOP is the best way :)

I absolutely believe that you knew what functions and pointers are, but functional programing is its completely own topic. You absolutely can't compare it to the way how you used it in Pascal or C, it is a completely different world. And like you have Design Patterns in OO, style books and best practices in OO, you also have things like that in Functional Programing. From Monad, Monoid, Currying, Closures, Functor. Just knowing functions and pointers isn't really what functional programing is about.

And OO is just a dying paradigm. Nearly all new features in nearly every language today is inspired by functional programing. And even

every language today is inspired by functional programing. And even concepts are copied from it. Immutability for example is a core idea from functional programing. And first-class functional languages are getting more and more important. Often because they are much easier then OO languages. Better designed, programmers do a lot less errors, write less code for the same things, have better typing systems and support parallel programing better. If you look at new features from Java 8.

https://leanpub.com/whatsnewin...

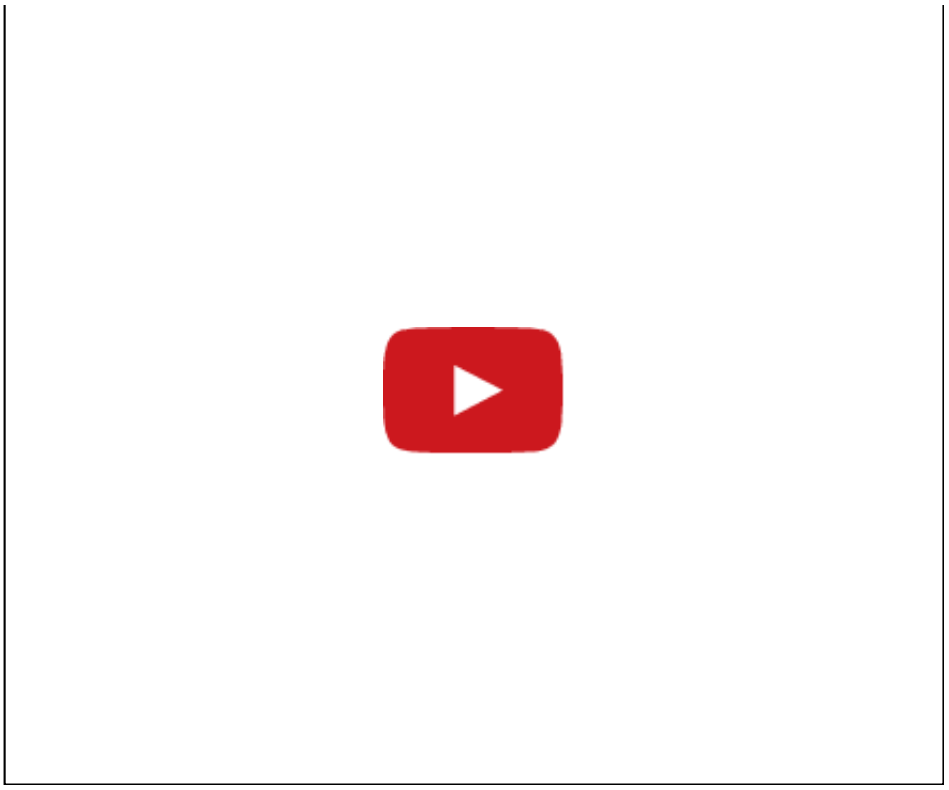
Lambdas - Functional Programing
Default methods - New feature needed for other Functional Programing features
Streams - Functional Programing
Optional - Core idea of functional because in functional languages you don't have NULL
Nashorn - Nice running "Node.js" a functional language in Java
Functional Programming in Java 8 - Nah self-explaining

This in general is how it looks in a lot of languages. OO is just outdated and bloated. I really can say, if you believe OO is the best way, just look a little bit into some Functional languages you will start to see the difference fast. ;)

Well, if you are interested in it, to get an overview.

Scala:

F#:
https://skillsmatter.com/skill...



• Reply • Share ›

Marcos Douglas Santos ➚ David Raab • a month ago

> Huh? The whole example that you provided was Dependency Injection. If you don't want a specific implementation and instead just want an Interface then you already have DI. Because instead that your class/method directly depends on a specific implementation it depends on an interface. And the concrete class gets injected by the caller.

If I don't want a specific implementation and instead just want an

Interface -- that is correct -- I do not need DI because all the dependencies needs to be passed in the contructor. Simple like that.

> Actually the whole problem doesn't differ whether you use OO or functional... [a lot of text and examples]

OK, I agree with you about these examples. Remember I didn't say "impossible" to test, but hard. But OK, is possible.
Any way I think classes is more productive and simpler to use. Do not forget encapsulation. If you only use functions, where are your data?

> I absolutely believe that you knew what functions and pointers are, but functional programing is its completely own topic.[...]

One more time, I know what is functional programming (FP). Today my thoughts is OOP is better than FP. I can change my opinion in the future, yes I can, but today I keep choosing OOP.

I'm not a Java programmer but I read (a little) about (featuares bellow) and that is what I think:
Default methods - Default methods in Interfaces, ie, implementations inside an Interface is abominable.
Streams - An ugly implementation, using inheritance.
Optional - Oh god... we do not need it, simply program the right way.

> This in general is how it looks in a lot of languages. OO is just outdated and bloated. I really can say, if you believe OO is the best way, just look a little bit into some Functional languages you will start to see the difference fast. ;)

OK, I will.

Thanks for the videos, I will see them later.

PS: Where are **@Yegor Bugayenko** and your answers?

David Raab ➚ Marcos Douglas Santos • a month ago

> I do not need DI because all the dependencies needs to be passed in the contructor. Simple like that.

Passing the dependencies to the construcot IS Dependency Injection

> OK, I agree with you about these examples. Remember I didn't say "impossible" to test, but hard.

I don't see what is hard, the style how to do it just differs.

> Any way I think classes is more productive and simpler to use. Do not forget encapsulation. If you only use functions, where are your data?

In an OO language using classes is more productive, yes that is because a lot of OO languages miss some features to make static methods/functions more useable. But my point was not to say that classes are bad or you should always use functions. I wanted to give a more detailed explanation and what you can do, and that using just using siomple functions is also acceptable in a lot of cases. This one-sided view of "all static methods are evil" is just "one-sided".

> Any way I think classes is more productive and simpler to use. Do not forget encapsulation. If you only use functions, where are your data?

At first. You only really need encapsulation in a mutable world. Encapsulation is to hide access to some state/data so a user cannot access/change it. In a functional world you always try that everything is immutable. With this idea encapsulation is not important, because a user cannot change things anyway. That doesn't mean you can't have encapsulation, but what it means and how you use it just differs a lot. As for the data it is simple. in a functional world everything is just bound to input/output of a function. The input/output is always immutable.

Functional languages instead try to focus on what you can pass and what returns. So you build types that represents your data. Building those types is a lot more powerful than what you probably knew, if you never used an "Algebraic Type system". Because you can build types with a lot of rules. For example you have the ability to describe "AND" and "OR" in a type. Thinks like, this type need to be A, B or C, or something like "this have to be an "A and a B". Well i don't want to go to much into detail. If you are interested in it i can give you this video.

https://skillsmatter.com/skill...

The video is also understandable if you never did functional programing or never used F#. But in general, you build those types and your functions either gets or return such types. But the general idea between OO and functional differs. While OO have the idea to combine data with functions in a functional world you do the exact opposite. You strictly separate data and functions.

> Default methods - Default methods in Interfaces, ie, implementations inside an Interface is abominable.

It is not an implementation inside an interface. It is the exact oppsite. You add a method that works on an interface without that it is added to the interface. In C# that is called "Extension Methods". If you knew how to use this technique, you can easily create interfaces that absolutely only implements what is needed.

> Streams - An ugly implementation, using inheritance.

The implementation of Streams is completely unrelevant here. While i also don't use inheritance, the important point is what Streams allows you to do. And the resulting technique are basics of the functional programing world.

> Optional - Oh god... we do not need it, simply program the right way.

If you want to do it right way, then you only use Optional. The current problem in a lot of OO languages is the existence of NULL. The problem is that it matches to absolutely every type. Everything can be null. And you have to check everything if it is "null". So if you want to do it "right". You have to check everywhere if something is probably "null". The functional way is. Something stupid like NULL doesn't exists. And yeah it is really stupid. Even the inventor of "null" says

> "I call it my billion-dollar mistake. It was the invention of the null reference in 1965."

https://www.linkedin.com/pulse...

Just the elimination of null can end up with writing ~20% less code. And the functional way is to explicitly say that something can be null. And if you explicitly say that it can be "Nothing" you always have to check if it is Nothing. And if you don't describe that something can't be Nothing, it cannot even happen that something is null, you forget to check it and it raises a NullReferenceException. And "Optional" exactly does that.

> One more time, I know what is functional programming (FP). Today my thoughts is OOP is better than FP. I can change my opinion in the future, yes I can, but today I keep choosing OOP.

Sure, everyone how he likes it, i like functional a lot more than OO. But one question, you are saying all the time you are not a functional guy and yet you say you knew functional. Did que ever learn a first-class functional language? I'm asking because C or Pascal and just

using some function pointers is absolutely not "functional". I mean did you ever for example looked into Scala, F#, Ocaml, Haskell and so on?

> Thanks for the videos, I will see them later.

Just one addition. If you want to look some of the videos. Watch the Domain Driven Design with "Scott Walshin" video first that i provided here in this post.

1 ▲ | ▼ • Reply • Share ›

Marcos Douglas Santos ➚ David Raab • a month ago

About DI, I thought you were talking about using some framework, reflection, etc.

Default methods, Streams, Optional, etc I was talking about the Java implementation. Maybe C# or F# is different, but any way...

> Sure, everyone how he likes it, i like functional a lot more than OO. But one question, you are saying all the time you are not a functional guy and yet you say you knew functional. Did que ever learn a first-class functional language? I'm asking because C or Pascal and just using some function pointers is absolutely not "functional". I mean did you ever for example looked into Scala, F#, Ocaml, Haskell and so on?

I had read about some functional languages, but never used in real projects.

> Just one addition. If you want to look some of the videos. Watch the Domain Driven Design with "Scott Walshin" video first that i provided here in this post.

Thanks for tip.

▲ | ▼ • Reply • Share ›

David Raab ➚ Marcos Douglas Santos • a month ago

> About DI, I thought you were talking about using some framework, reflection, etc.

No, what you probably meant was "Inversion of Control (IoC)". By the way "DI" is a form of IoC. The difference is like a callback and an event ;) But "Dependency Injection" is just a simple "passing the dependency in a constructor, setter, interface". Just look at Wikipedia: http://en.wikipedia.org/wiki/D...

For "IoC". I think IoC is a bad thing in multiple ways.

1 ▲ | ▼ • Reply • Share ›

Leon • 3 months ago

Pretty weak arguments here against the "Why not use them?" question. The main question here is whether or not there is such a thing as a static instance. Well, there is in Java, but I mean in OOP ideology. I think there should be. That would mean that instead of:

```
if (object1.equals(object2)) {}
```

we can do:

```
if (Object.equals(object1, object2)) {}
```

and avoid a potential NullPointerException. Or even an equals with varArgs. Unfortunately the Object class is written in stone.

For the arguments you gave:

> Testability. Calls to static methods in utility classes
> are hard-coded dependencies that can never be broken for testing
> purposes.

This was true years ago, not so today. You should take a look at PowerMock (https://code.google.com/p/powe....
That also solves the problem of having to make things non-private/final, just so you can test them.

Efficiency. That just depends on the implementation. Static calls can be more efficient because they don't have to create an instance, and there is more than one way to obtain lazy initialization.

Readability. That also depends on implementation. There is no reason why we couldn't have small utility classes. And you can open a method list with a keyboard shortcut in Eclipse anyway.

1 ▲ ▼ • Reply • Share ›

Oleg Majewski ➦ Leon • 3 months ago

> if (Object.equals(object1, object2)) {}

or we avoid null values, then we don't need it

1 ▲ ▼ • Reply • Share ›

Nicolas Bousquet • 3 months ago

There no real relation between what are the limitation of static methofd (a technical limitation) and OOP and functionnal programming.

One can make pure function with static methods just fine. If this method need a function in parameter and you use java 8 you can do just fine. If you want to use this static method as

a higher order function to pass it to another method, you can.

So all the argument that statics methods are not good enough for functionnal programming do not hold. There are limitations in Java8 functionnal programming, but not what is written here.

For testing, one can always mock static methods just fine too. That might not be great all the time but it works.

But what the difference with OOP? If we follow OOP principles, the implementation is sealed away, you have no clue how it works inside and can change the class behavior as easily. So you'll use tricks. Dependancy injection, a parameter to contructor, a setter, a mock library or define your mock manually to override the method that abstract the code you want don't want to test.

All theses tricks are necessary to mock OOP too. If one create a new object explicitely as part of a method and use it (a very common operation) you are powerless as it was a call to a static method. In term of expressivity, any java instruction has the same issue too, you can't change it. In functionnal program the issue is here too.

Be it dependancy injection, inheritence, mock of static methods there still the issue that somehow the caller has to know the implementation. If the Unit test need to provide a different implementation of an object what ever the way, the unit test is dependant of such knowledge. The day you refactor your code to not use this anymore or to use something else, this is the same issue as with mocking static methods: the test will no longer run.

There a difference through, if you are explicit and use good design, you can use force any caller to provide any meaningfull dependancy, for example as parameter to one of the methods you provide or to the contructor (but you loose then the interface abstraction). You are explictely dependant on the implementation of the class you use: it need a specific service, but at least it is declarative: it take it in input.

With automatic dependancy injection, with mock of static methods through, this is not declarative: you have to read at the implementation to figure it out. That's not as good. But really, the difference ends here.

Using statics methods make sence. You let the JVM optimize it. You also now it doesn't mess up with state. You know it safe in multi threaded environement and it can only operate logically on it's input. For providing some well contained features exactly as Math.abs it is perfectly fine. And well if you use Math.abs in your code and want to mock Math.abs, more often than not, I'd say you have a design issue: you do have test that are too dependant of implemenation and are too small.

My 2 cents.

1 ^ | v • Reply • Share ›

Oleg Majewski • 3 months ago

Hi yegor, some little corrections:

>Second, functional programming is based on lambda calculus, where a function can be

assigned to a variable

in a pure functional language you cannot do an assignment at all, and you should not. And if you can, that's by accident or an apology like "we have to be pragmatic"

>You can pass a static method as an argument to another method.

you mean canNOT

>FileUtils.readFile(), I will never be able to test it without using a real file on disk.

you KNOW, this is technically not true ;-) now days you can mock out everything, including static, final, private and other hard to test beasts ;-) This makes the util classes of course not better.

• Reply • Share ›

zeringus ➚ Oleg Majewski • 3 months ago

> you mean canNOT
Not true in Java anymore. myMethod(Math::abs) works!

1 • Reply • Share ›

Oleg Majewski ➚ zeringus • 3 months ago

yepp not true in java 8

• Reply • Share ›

David Raab • 3 months ago

I can't really agree nor disagree. I think the answer is much more complex than what you described. At first let's look what you want to prove. You say "Utility classes has nothing to do with functional programing". The problem is more to understand what you mean with this question. At first lets look what you probably mean. Functional programing is a whole paradigm itself and it has a lot of technices itself. From currying, higher-order functions, immutability, Monads and so on. A lot of features that shapes a functional language and somebody needs to understand. It the same in a OOP language, you have to understand SOLID, Composition, Inheritance and a lot of design patterns.

Is something OOP just because you can write a class? Absolutely not. You also can write static methods in a class and it has nothing to do with OOP programing. In fact you can also write a class with methods and have more things in common with procedural programing. But is the ability of writing a class a part of OOP? Absolutely yes. The same is truth for writing a function or a static method. When you write a static method is this what somebody can describe as functional programing? Absolutely not. The idea behind functional programing is a lot more than this. But is writing a simple function that returns something part of functional programing? Absolutely yes.

In fact you write helper functions like this all over the place in a functional language. In F# for example you also can group your functions in modules. And it absolutely makes sense to do that. I mean you probably don't want to have no namespaces and have all functions in one giant namespace. So in F# you for example have the "List.map" function that does a map on a List. A "Seq.map" function that does a map on a Sequence and also an

map on a List. A "Seq.map" function that does a map on a Sequence and also an "Array.map".

Grouping you functions somehow absolutely makes sense. Because even without it you probably want to do that. If modules would not exists in F# how would you otherwise name three map functions that operate on something different? "list_map", "seq_map" and "array_map"? And here i just want to mention that you can't write three different "map" functions that just differentiate in the types of the arguments. Because that is not the way how functional languages typically work. So you really need three functions with three different names.

So, is writing a utility class what you can describe for functional programing? Absolutely not, it is far more than this. But is writing helper function what you do in a functional language? Absolutely.

But it get interesting with your article. Let me point out that i don't think anything is wrong, just that for me you use the wrong terminology. For example you description about imperative vs declarative. What you describe here has nothing to do with imperative vs declarative. You absolutely say correctly at first that imperative is "how" and "declarative" is "what". But later on you don't show anything besides that. It is basically the same. For example your "MyMath.f" function and your LISP implementation does the exact same thing. You first calculate the "max" then pass the result to the "abs" function. You also could have writen the Java code as

return Math.abs(Math.max(a,b))

and i don't knew LISP well enough but you could probably also bloating the LISP implementation when you just bound the result from "abs" to its own variable as you did in Java. That is the reason why both your Java and your LISP code is basically the same. A better explanation for imperative vs declarative is a simple sum functions. Assume you have a List of integers and you want to sum then. In C# that would look like this:

```
var sum = 0;
foreach ( var num in integerList ) {
sum += num;
}
```

This code is imperative. Why? Because you exactly tell "how" the algorithm should work. You start and tell that you need a "sum" variable that will hold you result. You also need to initialize it with zero to correctly work. You exactly tell to iterative over each single element, and you say exactly what to do with every single element. But how would a declarative implementation looks like? (F#)

let sum = List.sum integerList

The difference is you don't do anything of the above. You basically just saying that you want the sum of "integerList" and you get it back. You don't explicitly iterate over it, create a temp variable and so on.

Now you could probably say that it is just a matter of a library to have such a "sum"

function. And in fact. Yes it is. If you also have a "sum" function in any other language or implement it yourself, then you also do declarative programing. And you also can do declarative programing in an OOP language. And also here is an important note. Somewhere you need an imperative implementation, also in a functional language that loops over each element. But why do a lot of people still say that OOP languages are more imperative and functional are more declarative? Because you can archive a declarative implementation a lot easier in a functional language.

Another simple example to demonstrate this is if we want to multiply every element by two before summing it. In an OOP language even if you have a "sum" function you would probably end up again with an imperative "foreach". Again in C#

```
var sum = 0
foreach ( var num in integerList ) {
sum += num * 2;
}
```

Well let's assume you have a "List.Sum" function in an OOP language, how could you reuse it? You could also first create a new list that has every element multiplied by two, and then use the result on List.sum. But the result is even longer.

```
var tmpList = new List<int>();
foreach ( var num in integerList ) {
tmpList.Add(num * 2);
}
var sum = List.Sum(tmpList);
```

A "better" way would be if the creation of the list and applying some code to every element could somehow be "reused". But the problem is a "pure" OOP language can't really do that. Well you could create a class where you have to override a method that gets passed each element before adding it to the List, but nobody would probably create a new class for something "simple" as multipling each element by two. But in a functional language something like that is easy. Because you can pass a whole function as an argument. (You also can do that since C#3 / Java8). But it is important that you need features that functional languages basically have decades and a lot of the built-in functions use that. So multipling by two and summing something up is just (F#)

```
let sum =
integerList
|> List.map (fun x -> x * 2)
|> List.sum
```

And to make it more declarative you also can write some helper methods

```
let multiplyBy2 x = x * 2
let sum =
integerList
|> List.map multiplyBy2

|> List.sum
```

.

And here it is important to note that "multiplyBy2" is a function passed to "List.map" and "List.map" executes this function for every element and creates a new list. Functional language are often described as "declarative" because it is extremly easy to be declarative. But you also can achive stuff like this in C#. Because today a lot of languages have some functional features. In fact LINQ in C# is basically just this stuff. And you could write.

var sum = integerList.Select(x => x * 2).Sum();

in C# and you also have a declarative way to do the same because "x => x * 2" is just a function exectued for every element. So also OOP languages get more declarative. Another thing is why declarative is important. When you use "map" in F# or "Select" in C# you are just saying that you want a new list and you want that the function that get passed is executed for every element and the result forms the new list. But you are not saying exactly how to do that. You are not saying. Give me element 0 apply the function. Add the result to the list. Get the next element, and so on.

This is important, because if you are declarative like this, instead of imperative you can change the way how it works. For example you could say that instead a foreach spawn multiple different. Thread 1 calculate all elements from 0 to 100, Thread 2 all elements from 101-200 and so on. Stuff like this for example is also a built-in in C# (LINQ) and you end just up with

var sum = integerList.AsParallel().Select(x => x * 2).Sum();

So "AsParallel()" returns another object that has "Select", "Sum" and so on implemented in a different style that uses multi-threading. If you end up writing declarative you can easily change the implementation.

This is more what "imperative vs. declarative" is about. Not what you descibed.

Another thing is your "gift card" example. Because that also has nothing to do with imperative vs declarative. Also in a functional programing language you just have function that executes and directly return something. What you are describing here is lazy-evaluation. Sure a function in a functional language *can* return a function that is latter executed, but it is not that every function in a functional language have to do that or only functions like these are "functional". Normal functions that calculates and directly return something are also part of a functional language. The following function in F# is for example something like this

let sum x y = x + y

It is just a function to multiply two values that get directly executed when i write:

let x = sum 10 20

Nothing is lazy-evaluated here like your gift-card example. Neither is it needed to be "functional". And also some things that you describe here are also language dependent. In efficiency you are saying that a utility class/function is always directly executed. Well like

described above, it is also not that every functional language have to be lazy for every function. And it is highly language dependent. In C# you can easily write static class methods that are lazy and only calculate what you need. Well you seem to use Java, but it is important to note that this is more a Java burden not a general burden that every static/help utility class need to implement. And in C# you also don't need to stuff everything in one giant ".cs" file and you can easily split it up because C# support partial classes. The problem is not static utility classes it is more a Java burden if Java doesn't support stuff like this.

▲ ▼ • Reply • Share ›

Oleg Majewski ➚ David Raab • 3 months ago

what a long answer and yes there are people really reading it ;-)
your explanation declarative vs. imperative is also wrong, what you describe is just implementation of a function vs. a call of that function.
I would say if you want a real example you should compare that max implementation vs. a SQL statement or a DSL, but nothing that you can program.
One more thing
>But is the ability of writing a class a part of OOP? Absolutely yes
If you want to say, without the ability to write a class you cannot do OO, then that's not true.

▲ ▼ • Reply • Share ›

David Raab ➚ Oleg Majewski • 3 months ago

If a function call could not be declarative, then declarative programing wouldn't exists. A function call is declarative if it hides the concrete implementation detail. SQL or a DSL is not more or less declarative. You also need an implementation to parse those.

> If you want to say, without the ability to write a class you cannot do OO, then that's not true.

No, that's not what im saying. I'm saying that the ability of writing a class is part of OO. But that alone doesn't mean something is truly OO or represent the whole OO programing. The same that the ability to write a function is part of functional programing, but that alone doesn't mean something is functional.

▲ ▼ • Reply • Share ›

Oleg Majewski ➚ David Raab • 3 months ago

the ability of writing a class is part of java or c# or yet another language which implements OO by using classes, but it's not part of OO. You can do OO also in a pure functional language, without any single class.

>A function call is declarative if it hides the concrete implementation detail.

Declarative language is about hiding the flow of control from you,

technically it means also hide the implementation, which controls the flow or in other words it is about describing what to do and not how to do it. You are right you need something which parses SQL or DSL or a config file, but this part does not have to be on your class path, neither that implementation is bound to the programming language used by your program. SQL is the best example, in your program it's just a string and you send it to some server (not necessary) and you don't care how it is executed.

David Raab ➚ Oleg Majewski • 3 months ago

> the ability of writing a class is part of java or c# or yet another language which implements OO by using classes, but it's not part of OO. You can do OO also in a pure functional language, without any single class.

I know that, and writing a class is still part of OO programing. The same that Mercedes is a car, but not every car needs to be a mercedes. That you also can have cars that are not Mercedes doesn't change that Mercedes is a car. The class keyword is part of OO programing and only makes sense in this context. You still can code OO without a class keyword, and it doesn't change that writing a class is still OO programing and a part of it.

> Declarative language is about hiding the flow of control from you, technically it means also hide the implementation which controls the flow or in other words it is about describing what to do and not how to do it.

What a function call is, and that is also exactly what i described. A

let m2 = List.map (fun x -> x * 2) integerList

is thus declarative because it just describes what should be done with every element. Besides of what to do it contains no other information like, iterating through every element, creating a new list, adding the result of the function to the new list on so on. The flow could also be changed to run in parallel without changing much. A foreach construct on the other side does not only described what to do, it also describes how to do it. So something like List.map a function call is declarative and a foreach is imperative.

> You are right you need something which parses SQL or DSL or a config file, but this part does not have to be on your class path, neither that implementation is bound to the programming language used by your program.

None of this is important if something is declarative or not. Something is declarative if it is like you said: If it just describes what

to do. Where exactly the code relies is not important. SQL for example will not be less declarative if you have a Java Library and parse SQL yourself.

> SQL is the best example, in your program it's just a string and you send it to some server (not necessary) and you don't care how it is executed.

What you describe is more abstraction. SQL doesn't get less declarative if a Java/C#/Whatever library parses the string and do something with it. The same that List.map doesn't get less declarative just because it is part of the language or is accessible in some library file.

• Reply • Share ›

Oleg Majewski ➚ David Raab • 3 months ago

replace declarative with functional and I totally agree with everything ;-)
some are saying functional programming is declarative, but cannot agree, as functional implementation is still an implementation which do exactly the same thing as the imperative implementation, it just describes the implementation in a different way. Ok there of course reasons and differences, but in general it is just this. For me something which is declarative should not be programmable at all and this is more abstract then functional programming.

• Reply • Share ›

Maxim Kornienko • 3 months ago

I think Testability example is quite far-fetched. First, it doesn't have anithing against Math utils as Math utils doesn't produce side effects. Second, if your code doesn't use any wrapper around IO processing code, it's not a fault of utility class that program lacks indirection. Third, as I understand you can avoid reading from a disk with tmpfs.

• Reply • Share ›

Zheka Gibbons • 3 months ago

In Haskell you also have a lot of utility methods:
abs 4 is the same as Math.abs(4)
readFile "abc.txt" is the same as FileUtils.readFile("abc.txt")

Want to pass a static method to another method? No problem:
f abs is the same as f(Math::abs)
f readFile is the same as f(FileUtils::readFile)

Want laziness? Use Suppliers.memoize:
Suppliers.memoize(() -> Math.abs(<heavyweight expression>))

• Reply • Share ›

Oleg Majewski ➚ Zheka Gibbons • 3 months ago

yepp lambda's in java 8 are great, but it does not make it a functional language, it only enables you to do some functional stuff.

>Suppliers.memoize(() -> Math.abs(<heavyweight expression="">))
is a good example, it shows how you need yet another util class (Guava in that case), making the whole thing even more hard to test.

• Reply • Share ›

Carlos Alexandro Becker • 3 months ago

I fully agree with you. I'm not an OOP fundamentalism, but I'm strongly against static methods, and that utility classes are not even remotely FP-ish.

But I'm also curious about your opinion on the BigDecimal class, for example (just because you pointed Math out and everything). I think that, as a Number, it should have the `+`, `-`, for example, `methods`, instead of `add` and `subtract`. What do you think?

• Reply • Share ›

Yegor Bugayenko author ➚ Carlos Alexandro Becker • 3 months ago

Yeah, absolutely, BigDecimal should be very similar to Long or Double. They all have to implement the same interface, which is Number. I think the problem with Java is that it inherited so much from C++, including that scalar types. They ruin the entire OOP paradigm.

• Reply • Share ›

Carlos Alexandro Becker ➚ Yegor Bugayenko • 3 months ago

Seems legit, thanks!

• Reply • Share ›

Lyubomyr Shaydariv • 3 months ago

I disagree. You wrote: "Testability. Calls to static methods in utility classes are hard-coded dependencies that can never be broken for testing purposes. If your class is calling FileUtils.readFile(), I will never be able to test it without using a real file on disk." That's true, I will never be able to test it not reading content from the file. But let's just take a look closer: the Math class from Java 1.0 and FileUtils are concrete examples. They aren't abstractions per se. If I want to test Math.abs, simple asserts are enough. Moreover, if it I want to test FileUtils, I *must* have something to test on the disk, because this is FileUtils named self-descriptively. However, you need to test it if you write low-level I/O routines, because it's not a high-level abstraction like IBytesProvider or just a simple java.io.InputStream.

2 • Reply • Share ›

Oleg Majewski ➚ Lyubomyr Shaydariv • 3 months ago

>If your class is calling FileUtils.readFile(), I will never be able to test it without using a real file on disk." That's true
no, technically this is wrong, as you can mock it away, but it still not good, as mocking makes your tests hard and coupled to the implementation details, but that's another story.

• Reply • Share ›

Carlos Alexandro Becker ➚ Lyubomyr Shaydariv • 3 months ago

I believe it was talking about unit testing. Testing a File*Anything* with a real file is not unit testing, I believe: it's an integration test.

• Reply • Share ›

Lyubomyr Shaydariv ➚ Carlos Alexandro Becker • 3 months ago

It doesn't matter that much indeed. If you're going to test FileUtils -- you finish up with real disk I/O.

• Reply • Share ›

Carlos Alexandro Becker ➚ Lyubomyr Shaydariv • 3 months ago

Still not unit testing.

• Reply • Share ›

Invisible Arrow ➚ Lyubomyr Shaydariv • 3 months ago

He isn't talking about testing FileUtils itself. He is talking about other pieces of code using FileUtils. These pieces of code are now untestable without a real file, since the FileUtils usage is hard-coded. If the code instead depends on an InputStream instead, for example, then you can unit test it by injecting a ByteArrayInputStream in the test.

2 • Reply • Share ›

Lyubomyr Shaydariv ➚ Invisible Arrow • 3 months ago

This is exactly why I mentioned "a high-level IBytesProvider" above.

• Reply • Share ›

Lukas Eder • 3 months ago

You do, of course, realise that my original criticism, to which you are now eloquently replying, was based on your refusal to accept that `Math::max` is a good-enough *functionalish* approach that Java took at creating what you consider pure FP, right? :) (and, at the time, you preferred the OOP approach)

Nonetheless, good article!

• Reply • Share ›

Yegor Bugayenko author ➚ Lukas Eder • 3 months ago

I still refuse to accept that utility classes/methods have anything to do with functional approach. They are procedures, not functions. No matter how we compose them and how they look. Their overwhelming presence in modern Java code is a big misfortune for all of us. And there is no excuse for their existence. Those who invented `Math` class in Java SDK should be severely punished :)

• Reply • Share ›

Lukas Eder ➚ Yegor Bugayenko • 3 months ago

More from yegor256.com

yegor256.com recently published

How Cookie-Based Authentication Works in the Takes Framework

11 Comments Recommend

yegor256.com recently published

There Can Be Only One Primary Constructor

33 Comments Recommend

yegor256.com recently published

How to Avoid a Software Outsourcing Disaster

8 Comments Recommend