Community

yegor256.com

139 Comments • Created 9 months ago

Getters/Setters. Evil. Period.

Getters/Setters. Evil. Period.

There is an old debate, started in 2003 by Allen Holub in this Why getter and setter methods are evil famous article, about whether getters/setters is an anti-pattern and should be avoided or if it is something we inevitably need in object…

(yegor256.com)

139 Comments

Recommend 5 Share Sort by Newest

Join the discussion…

hasufell • 6 days ago

Very interesting angle. I think you are on the right track and probably a joy to work with as an OO programmer, but after reading this I really have the feeling that you missed out on purely functional programming. A lot of what you say here you get for free in languages like Haskell (e.g. immutability, non-spagehtti code, encapsulation, missing NULL references) and even more (no side-effects?!). There's no way to think old-style imperatively in such languages. Of course it's also different from OOP, but given the points you want to improve by the presented way of thinking... the next step is purely functional programming, IMO.

1 • Reply • Share ›

Yegor Bugayenko author ↗ hasufell • 6 days ago

Indeed, functional programming is very close to OOP, if OOP is done right. But still, I believe, they are different and OOP is more powerful.

• Reply • Share ›

Riccardo Cardin ↗ Yegor Bugayenko • 5 days ago

OOP is not a programming paradigm, such as imperative, logic or functional programming. It is an orthogonal approach. Indeed, we have languages like C++ or Java, that are object oriented and imperative; And languagues like Scala, that are object oriented and functional.

• Reply • Share ›

Ravi .S • 2 months ago

Dog object is a living being, right ! So let's say, Dog "lucy" weighed "23" kg last year and this year "she" gained 3 kg extra. How does the "living" object updates it's "weight" ? And what do you call the "method" !? setDogWeght(float kgs) or setWeight(float kgs) or addNewWeight(float kgs) or updateWeight(float kgs) !!!???

• Reply • Share ›

Yegor Bugayenko author ↗ Ravi .S • 2 months ago

Check this article, it should answer the question: http://www.yegor256.com/2014/1...

• Reply • Share ›

Dr. Mallah • 2 months ago

You are ahead of the curve. Alan Holub's article from 2003 was warning everyone of this issue:

http://www.javaworld.com/artic...

Now most 'professional' programmers are stuck with this mind-set and are simply unable to understand why it is a terrible coding practice. In another 10 years the paradigm will follow what you are doing here, and become more rational.

The programmers in my former company were stuck with this mindset. I once heard a senior developer say... "oh, you should make all your variables private... for good OO". I don't think he understands what OO is, and unfortunately is only replicated other bad habits from other people. Every single variable in every single class in the production code had a getter and setter method. Talk about wasting time in writing code like this, in addition to the physical line use in declaring the same thing over and over again for each var.

The obvious joke in all of this, most of these coders are using 'public' access to their get and set methods, making the entire process completely redundant. The code might as well use public variables to begin with.

I also read another article about this:

http://typicalprogrammer.com/d...

It is incredible how people are struggling to grasp this... but there is a great example in the comments.

Ciao.

4 • Reply • Share ›

Yegor Bugayenko author ↗ Dr. Mallah • 2 months ago

Thanks for sharing your thoughts, we're on the same page :)

1 • Reply • Share ›

Q Ball • 3 months ago

what if the object is a bank account? its obviously not immutatable and you need to be able

to manipulate the data, or do you recommend deleting and creating a new bankaccount object?

• Reply • Share ›

Yegor Bugayenko author → Q Ball • 3 months ago

The bank account object is immutable, but the money it holds are not. Just like java.io.File is immutable, but the content of the file is not. Check this article about this subject: http://www.yegor256.com/2014/1...

• Reply • Share ›

Q Ball → Yegor Bugayenko • 3 months ago

its not the same. java.io.File is a class. Im talking about an instance of the class, an object that needs daily mutable data. if one doesnt use getter setters is there even an efficient way to create a Bank program with multiple bank account where the numbers change daily? you seem very anti getters/setters but dont provide an alternative.

when reading your reply I get so confused I wonder if you even understand the difference between a class and the instances/objects one creates from that class. because you seem to think java.io.File is an object, when its a class.

• Reply • Share ›

Yegor Bugayenko author → Q Ball • 3 months ago

This is a perfectly immutable bank account object:

```
final class Account {
  private final SQL sql;
  public int balance() {
    return this.sql.exec("SELECT SUM(dollars) FROM transaction");
  }
  public void add(int dollars) {
    this.sql.exec("INSERT INTO transaction VALUES (?)", dollars);
  }
}
```

2 • Reply • Share ›

Q Ball → Yegor Bugayenko • 3 months ago

Ok it makes sense now that I understand what you mean, but I still disagree with your terminology. imo Account is a class from which you can create immutable objects, but it by itself its not an immutable object.

so if I wrote Account account1= new Account(1000);in a main method then account1 is a new immutable object which puts the number 1000 in the database. as if its the account size.

I think your article would be clearer if its not just focused on just getters/setters are evil, but data should remain in a database and not

getters/setters are evil, but data should remain in a database and not be a part of the program itself because that seems to be the core of your argument.

• Reply • Share ›

Yegor Bugayenko author ➚ Q Ball • 3 months ago

This article is about this (I'm strongly against this):

```
final class Account {
  private SQL sql;
  public SQL getSQL() {
    return this.sql;
  }
  public void setSQL(SQL sql) {
    this.sql = sql;
  }
}
```

• Reply • Share ›

Ravi .S ➚ Yegor Bugayenko • 2 months ago

Nobody gives access to private fields with setXXX() or with some other kind of terminology, unless it is must. And even if it happens, in any serious application, there will be extra layer of security and further encapsulation; for example, the setXXX method might (or rather must) verify the "privileges" of the person/system/entity "asking" to do the setting.

It all depends on the context, domain, implementation and business logic. In your example, there is absolutely no point in writing the method as setSQL(), you can write addAmount(), makeDeposit(), addInterest() ... !!!.

I haven't seen a developer using a "setXXX()" in the context you mentioned !. Good luck !.

• Reply • Share ›

karamba • 3 months ago

WTF man, too philosophical article. Too much emotions glued to labels. It is only notation, nothing to do with architecutre. Is it really such a big difference if i name method getBall() instead of give()? I don't think so.

1 • Reply • Share ›

Ravi .S ➚ karamba • 2 months ago

Bless you !.

• Reply • Share ›

Chris Knoll • 4 months ago

Eliminates NULL references, eh? So what happens when you ask the dog to give you something he never got int he first place? Just remove the whole take() call and say you

started with a dog and say give(). What? Nothing to give? Then returning null is appropriate. You'd rather have a 'DogDoesNotHaveBall' exception? have fun with that!

Point being, thinking that null references can be eliminated under your premise is absurd.

-Chris

• Reply • Share ›

Oleg Majewski ➚ Chris Knoll • 3 months ago

Ball is optional for the dog, you could return Optional.absent() instead of NULL

1 • Reply • Share ›

Máté Magyar ➚ Oleg Majewski • 3 months ago

how is that any better than null?

• Reply • Share ›

Oleg Majewski ➚ Máté Magyar • 3 months ago

```
Optional<ball> ball = dog.giveBall();

// now you cannot do ball.name(), you are forced to think about if

ball.ifPresent(b -> System.out.print(b.name())); // b is safe to b
```

compare this to:

```
Ball ball = dog.giveBall(); // ball may be null

System.out.println(ball.name()); // happy bug and NullPointerExcep
```

you can read also this: https://code.google.com/p/guav...

1 • Reply • Share ›

Máté Magyar ➚ Oleg Majewski • 3 months ago

While that is a nice idea, i'm pretty sure there are a lot of programmers, like me, who have not had the fortune to use java 8. Besides it's not a silver bullet. it may have a performance/ memory overhead, ifPresent lambda could need to use closures, that could possibly lead to complications, and many programmers would just use optional.get(), which would throw and illegalStateException, which isn't any better than a null pointer Exception.

• Reply • Share ›

Oleg Majewski ➚ Máté Magyar • 3 months ago

before java 8 you can use guava's Optional, java's Optional is

before java 8 you can use guava's Optional, java's Optional is nothing else then recreating that idea (as usual they don't have their own). In general open guavas documentation, start reading it from beginning to the end, it will change your way of programming ;-)

> it may have a performance/ memory overhead

we are in the year 2015, what are you talking about???

1 • Reply • Share ›

Máté Magyar ➔ Oleg Majewski • 3 months ago

There are, in fact places where performance doesn't matter, but it's important to remember that it's not a zero cost abstraction. It does have a very significant overhead, when you have to work on huge data sets.

• Reply • Share ›

Oleg Majewski ➔ Máté Magyar • 3 months ago

performance DOES matter a lot,
that one function call is not what you should care about, but that huge data set you mentioned like loading 1.000.000 elements first and then fetch 10 of them from memory does matter. Chears ;-)

1 • Reply • Share ›

Nikola Boricic • 4 months ago

What about a scenario where app resources are limited and performance is critical? For example in mobile applications Android ListView objects are not immutable because for long lists that would crash the application. Instead there are a few view objects which are reused as the user scrolls through the list. Do you see this as bad practice? How would you handle such scenario without mutators?

• Reply • Share ›

Yegor Bugayenko author ➔ Nikola Boricic • 4 months ago

I would still use immutable objects, but with in-memory mutable storage behind them. Check this article: http://www.yegor256.com/2014/1...

• Reply • Share ›

Afroradiohead • 4 months ago

Exactly how I program. Except!! Final properties, no methods. Once an object is constructed.. it's constructed!

However that introduces a problem in dynamic applications. So I added one more practice that I've been using quite successfully. "Every object has the ability to listen to the creation or destruction of another object type". This follows Yegor's idea that "Objects should be immutable....to other objects. But the object can change it's own state". It applies best to games because of game loop.

The idea then evolves from "Final properties, no methods" to "Final properties, onObjectCreation/onObjectTypeDestruction methods". Still applying it to my application in

the hopes that it fails me, but reading Yegor's articles has help me gain confidence that I'm headed towards a better direction of programming.

1 • Reply • Share ›

Marcos Douglas Santos ➚ Afroradiohead • 4 months ago

"But the object can change it's own state"

I not see what the difference if an object change it's own state or a second object do it. In both cases the first object is not immutable.

• Reply • Share ›

Afroradiohead ➚ Marcos Douglas Santos • 4 months ago

You're right. Immutable may be a bad choice of words if we look at it by it's definition.

I guess to further clarify this concept, I mean "An object should ONLY be internally mutable"... as it more accurately depicts real world objects. For example:

A FootballPlayer can decrease/increase it's runningSpeed. It's not the FootballGame nor FootballField nor any other type of object that changes it, it's the created FootballPlayer itself. How that works is... on creation, the criteria (events) that determine when and how it should change itself is already known by the object. This design allows objects to maintain it's integrity in respect to the application in the sense that object's are born into the world (application) knowing exactly how they should respond to their environment.

A quote that gives the perfect analogy to this is: "You can't change the way someone responds to you... but you can change the way you respond to them" .

• Reply • Share ›

Marcos Douglas Santos ➚ Afroradiohead • 4 months ago

I understood now.

Well is better than use mutable objects at all but... I don't know. If you using events to change the state of your objects maybe you have mutable objects that are changed indirectly?

If an object represents an real-world entity, you can't change the state them because if you do, you are breaking this representation, ie, you will have two objects: a) one virtual, that "inherited" from real-world entity and now has your own properties that can change and b) the real-world entity.

I think that is the theoretical concept that Yegor explains all the time at this blog.

1 • Reply • Share ›

Yegor Bugayenko author ➚ Afroradiohead • 4 months ago

Glad to be helpful :)

^ | v • Reply • Share ›

David Bowman • 5 months ago

I recently built a system that was 100% immutable, absurdly cohesive, and had zero coupling.

Only one problem; when it starts up, nothing happens; just a blinking cursor.

2 ^ | v • Reply • Share ›

Yegor Bugayenko author ➚ David Bowman • 5 months ago

I'm planning to create a Java web framework, which will contain and manipulate only immutable objects. You may be interested to see its draft: https://github.com/yegor256/ta...

^ | v • Reply • Share ›

Oleg Majewski ➚ Yegor Bugayenko • 3 months ago

@Yegor take a look at http://www.ratpack.io/

^ | v • Reply • Share ›

Hans-Peter Störr • 5 months ago

For me, it's somewhat confusing that you talk about "immutable living organisms". I see that "living organism" as object would be a useful metaphor for e.g. Scalas actors. But if it's immutable, wouldn't it be a statue of a dog, not a dog itself? Or maybe a snapshot of a dog, where actions can generate new snapshots. I'm somewhat stumped for a good methaphor for immutability.

1 ^ | v • Reply • Share ›

Yegor Bugayenko author ➚ Hans-Peter Störr • 5 months ago

Immutable doesn't mean dumb of lifeless. Take a look at this article: http://www.yegor256.com/2014/1...

^ | v • Reply • Share ›

Hans-Peter Störr ➚ Yegor Bugayenko • 5 months ago

I didn't say immutable objects are dumb, but that they have unchangeable state, whereas living beings have a changing state by definition. Thus, an immutable object can only represent a snapshot of a living organism, or be a pointer to it. But that's a detailed discussion I'll take to that article you linked.

1 ^ | v • Reply • Share ›

Mangesh Phadtare • 6 months ago

@Yegor , Indeed a nice post. Thinking in OOP. Nice Explanation.

1 ^ | v • Reply • Share ›

John Balmer • 6 months ago

Interesting. But how do you argue when using magic methods inside objects, like

constructor? I think a dog is also unable to construct itself, or doesn't depend on a toString method.

2 • Reply • Share ›

Yegor Bugayenko author ↗ John Balmer • 6 months ago

Indeed, a dog doesn't construct itself. A class of dogs constructs individual dogs. Take a look at this article, it will help to explain this better: http://www.yegor256.com/2014/1...

1 • Reply • Share ›

Marcos Douglas Santos • 6 months ago

Maybe an article about CRUD with immutable objects would be great.

1 • Reply • Share ›

Marcos Douglas Santos • 6 months ago

It makes sense but not for everything, eg, if I have a Employee class and his name was registered wrong, how can I change your name without a SetName method? Does not make sense, in real life, change the BirthdayDate, Weight (this will change inside after he Eat(cookie);), and many others attributes. But people change your own name when they marry, do not like the name or whatever.

Thanks.

2 • Reply • Share ›

Oleg Majewski ↗ Marcos Douglas Santos • 3 months ago

>if I have a Employee class and his name was registered wrong, how can I change your name without a SetName method? Does not make sense, in real life

```
employee = employee.withName("other name");
```

see also https://immutables.github.io/i...

• Reply • Share ›

Marcos Douglas Santos ↗ Oleg Majewski • 3 months ago

Yeah, I'm already familiarized about immutability now -- this post is old -- but thanks for the answer.

One thing: using withName or setName doesn't matter if you return a new object. Just because the name begins with "set" isn't wrong. In fact, I don't use either of them now. :)

• Reply • Share ›

Oleg Majewski ↗ Marcos Douglas Santos • 3 months ago

technically not, but using a setter nobody would expect it to create a fully new immutable object and copy all values but not that one from the setter, which is the case for the wither methods.
What are you using instead?

• Reply • Share ›

Marcos Douglas Santos ↗ Oleg Majewski • 3 months ago

I did not needed, so far, recreate objects through their own methods - I'm talking about new projects, of course, because the old projects continues using Get/Set. So, all my objects born and die without recreate them.

But I have some questions to Yegor answer. If all these questions are answered -- not only by Yegor -- I will use only immutable objects in all my projects, even the old projects.

^ | v • Reply • Share ›

Yegor Bugayenko author ↗ Marcos Douglas Santos • 6 months ago

This article should answer the questions you asked:
http://www.yegor256.com/2014/1...

^ | v • Reply • Share ›

Hlodowig • 6 months ago

You must have one helluva dog, telling its weight and all! Mine's pretty dumb.

1 ^ | v • Reply • Share ›

Zavael • 6 months ago

Your posts are interesting and even if I dont follow you thougts absolutely, they keep stimulate me to rething some concepts. But I cannot imagine, how is your code organized without all of the *Worker, *Builder, *Helper, *Util, *Service classes, which I use pretty often if some code can be separated into bussines/logical parts with the method name sufficient to describe the functionality.
With the "classes represent real life objects" i think you will have much less classes in mid range project with much more lines of code per class, won't you?

^ | v • Reply • Share ›

Yegor Bugayenko author ↗ Zavael • 6 months ago

In reality, exactly the opposite is happening. You end up with more classes and each of them is very short. You may find these two articles interesting, about this very subject: http://www.yegor256.com/2014/1... and http://www.yegor256.com/2014/1...

1 ^ | v • Reply • Share ›

Zavael ↗ Yegor Bugayenko • 2 months ago

Thank you they are interesting, The composite decorators article even more! I have to use them much more...

^ | v • Reply • Share ›

oddparity • 7 months ago

No, coding is not a religion, like you state down below. And the article also supports this notion of yours, IMO. Coding is a process to produce a tool. The tool must work, it must be maintainable, extendable, debuggable, testable. This tool can work with getters and setters as well.

5 • Reply • Share ›

Yegor Bugayenko author ➚ oddparity • 7 months ago

Yes, it will work with getters, setters, singletons, static methods, without unit tests, with God objects, with spaghetti code, and without any documentation. Let's be honest, most of the code we see in the industry is done exactly that way. Do we like it? Do we want to improve this? Do we love our profession or we're working for food?

I think that coding is a lifestyle/religion/art, not a "process to product a tool". You're spending 1% of your time in dating, and 80% of your life in front of a computer. Why do we want to date beautiful women/men and we don't care about how beautiful is our code?

3 • Reply • Share ›

oddparity ➚ Yegor Bugayenko • 7 months ago

I love beautiful code, too. But renaming getBall() to giveBall() will not change much. Plus, I don't like surprises that much. I mean, some colleagues might implement spitBall(), some even getBall() (because they always say "get me the ball"). I'd like to find the method returning the memorized ball at once, so I rather stick to common naming conventions and also like other people doing it. Rather I'd like my team members invest their creativity in good design (e.g. weighing choices on the other topics you mentioned) and good micro coding. Also my objects are not the computerized incarnations of living things, they are the coded projection of a file, a folder, a form and its contents. The paper and what's written on it has no intelligence - the process working on it has. I like Mr. Fowler, but I am a fan anemic design, I guess.

7 • Reply • Share ›

veggen • 7 months ago

Wow, what an incredible amount of mental masturbation! I've never seen a post less connected to reality. Respect the principles outlined here and you're well on your way to make a program that will serve one rigid scenario with no route for usage in any similar scenario without modification. This might sound OK, but is very much not. Use it in framework code and it makes sure no one will ever be able to use that framework. But, hey!, it's great OOP. Go stroke your mental penis while thinking of it.
This crap might be good for killing time in academy, but when you need to make something that other people need to you in their own use-cases, it just stops making any sense.

7 • Reply • Share ›

Fred ➚ veggen • 5 months ago

I've been working for 4 years now, thinking that computer sciences was my religion since I am 8, but I have worked too many times with guys thinking like you "I like the crap I am doing every day, convinced about the crap which other mentally deranged people have created" that now I question myself "Should I quite this fool world or should I accept to make shit ".

This stupid pattern has been created by a consensus of fools which were really c-

oriented programmers and did not understand AT ALL what Object Oriented Programming was. Of course, as the majority of people is composed of c-programmer fools, that stupidity became a convention.

1 • Reply • Share ›

veggen ➚ Fred • 5 months ago

Ok, so people that invented the language have no idea how to code in it, but you do? Check. And it's shit because it's been serving its purpose so well these past 2 decades? Double check.

Don't get me wrong, I'm very much for new approaches to things and improving what we do daily, but replacing simple one liners with complicated noise constructs for the sake of OO purity serves none of those goals. Sure, if we all just decided that C style is bad and switched to "pure OO", we would no longer see creating a whole new class to replace a setter as noise, but it would still be noise - as it expresses nothing except that the author really likes feeling smug.

• Reply • Share ›

Sihle ➚ veggen • 7 months ago

Never knew OO topics can conjure images of masturbation and private parts. Learn to encapsulate your emotions a bit please!

6 • Reply • Share ›

Yegor Bugayenko author ➚ veggen • 7 months ago

Look at these libraries:

* jcabi-http
* jcabi-github
* jcabi-dynamo

They are made with these principles in mind. And people use them, rather actively.

• Reply • Share ›

veggen ➚ Yegor Bugayenko • 7 months ago

Using something and using it with pleasure are two different things. Let me illustrate. Your dog object is created by a third-party library (e.g. framework code) and needs to be passed to yet another third-party library. A fairly common scenario in integration tasks. All is dandy until you realize the second library requires dog weight in kilograms while the first one passed it in pounds. No problem you think, I'll just dog.setWeight(dog.getWeight() * 0.45). Oh wait, you can't because setter are evil and bad OO. Ok then, I'll just make a new dog and copy everything over. That's brilliant OO, right? Some dude on the net said so, so it must be true.

4 • Reply • Share ›

Oleg Majewski ➚ veggen • 3 months ago

>Ok then, I'll just make a new dog and copy everything over.

```
Dog dog = ... from some framework
Dog dogForOtherFramework = dog.withWeight(dog.weight() * 0.45) //
```

>That's brilliant OO, right?

dunno if it's brilliant OO, it's rather about immutability and less about OO, but it has advantages, e.g. you don't need to care about the state of the input dog and do a safe logging, even after the operation is done, you have the guaranty that dog did not change, as it is immutable. Also it's thread safe, you can easily do that mapping from one dog to another in parallel for many of them, etc.

• Reply • Share ›

Yegor Bugayenko author ➚ veggen • 7 months ago

I would rather "decorate" a dog, instead of using setters/getters or copying everything over. See https://en.wikipedia.org/wiki/...

3 • Reply • Share ›

veggen ➚ Yegor Bugayenko • 7 months ago

Brilliant, now you've replaced an one-liner with a whole new class. I'd rather have a simple setter, with well-understood semantics even among newbies, then have an extra wrapper class with dubious semantics (like DogWithWeighInKg) adding noise for the next poor soul to look at that code. The reason why I call this mental masturbation is that it's there to make the original author feel good, not to achieve anything substantial, especially in the long run.
If you were arguing for immutability of objects for easier concurrency, I'd understand the merits, but you have no issues with actually setting stuff, you just want it more complicated and noisy.

9 • Reply • Share ›

Bruno Skvorc • 7 months ago

This is nitpicking for fame and fortune. Build good apps, call your methods what you will.

7 • Reply • Share ›

Yegor Bugayenko author ➚ Bruno Skvorc • 7 months ago

Yeah, be a good boy, listen to your mom and everything will be fine :) This is a good slogan for kids, but in a serious software development we need rules, principles, and discipline. Object-oriented programming gives us that discipline if we understand it correctly.

• Reply • Share ›

Felipe de Melo Pimentel • 7 months ago

This article proposes that we bring the limitations of the real world into the virtual world. The great magic of programming is doing things in the virtual world that would be physically impossible in the real world, and this is what makes program's users say WOW. I can change the weight of the dog without altering its physical form is a magical, eliminate

change the weight of the dog without altering its physical form is a magical, eliminate gravity, making objects fly, and other impossible things that we think are the big difference in the world of computer programs.

Still has the issue of objects that don't exists in the real world that must be completely invented, as an example of ControlProviderFactory, impossible to make the analogy with the real world. Soon somebody will say that I can't create a wheel attribute in a dog object because a dog in the real world doesn't have wheels. But in a virtual world I can want that my dog have wheels, why not, this is big plus, making the impossible possible. I think that think OO is essential for programmers today, but we can't let this wish bring the limitations of the real world into our programs, after all the virtual world should be unlimited and should goes where the imagination of each one programmer is able to reach.

2 ▲ | ▼ • Reply • Share ›

Sihle ➚ Felipe de Melo Pimentel • 7 months ago

If your problem domain is to write games and such, maybe what you are talking about is plausible. But making a Savings Account to have a negative million or a Complex Number figure is just plain crazy and is out of touch with the real world. I suppose context matters, in which case you could solve "unreal" world problems using your "unlimited" approach.

▲ | ▼ • Reply • Share ›

Lev • 7 months ago

Come on, it's all just about naming.

Your example is perfect for such a canonical real-word beginner-book objects like Dog, Cat, Animal, Human. But it will not work for an object like ControlProviderFactory. It simply has no real-word analogy to obtain the verb for naming the method. Or I'm just lazy to think out a method name and I better stick with a usual getSomething() name. Doesn't matter how you named your getter, you just get the value of some inner state through it.

Overall OOP is just a way arrange the code in a reusable and interchangeable way. Nothing more, nothing less. I think you are canonizing OOP too much.

P.S. I still agree with you on mutability minimization. Setters are evil.

3 ▲ | ▼ • Reply • Share ›

Yegor Bugayenko author ➚ Lev • 7 months ago

I believe that OOP is more a religion than a way to arrange code :) And a thing named "ControlProviderFactory" is not an object, in terms of OOP. It doesn't abstract a real-world entity, it has an "-er" inside its name, and it is a factory, which is an anti-pattern. I'm going to publish a "What Is An Object" article soon, which will explain it in more details. My point here is that when your objects are true/proper objects, you won't have problems with names inventing. You won't need "getters". When it's a dog in front of you - you immediately understand what to do with her. On the other hand, when it's a "factory" in front of you, you have no idea what is it and how he/she (?) can help you.

▲ | ▼ • Reply • Share ›

pjmartos • 7 months ago

I do think there are legitimate uses of the getter/setter idiom, mostly when communicating with external sources and retrieving data. Although a pure OOP aproach could be achieved, it would be too hard to develop it that way without leaking any of the data via a getter/setter method masquerading itself as a not-so-different thing (as the examples you propose, I got to say), and there's a good chance it'd make your API unnecessarily complex. The "Tell Don't Ask" principle (or "Information Expert", as Craig Larman calls it) is great, makes your code more reusable and cohesive, avoids breaking encapsulation and stuff... but being obsessive about it (or any other pattern, for that matter) might turn the best practice into an antipattern without you noticing ;)

P.S. I share your opinion about the setter injection. The main drawback about constructor injection (subclasses need to override the constructor) is not that bad if you avoid deep inheritance hierarchies and favor composition instead.

1 ^ | v • Reply • Share ›

Yegor Bugayenko author → pjmartos • 7 months ago

I've never seen a best practice turning into an anti-pattern, maybe I need more time :)

^ | v • Reply • Share ›

pjmartos → Yegor Bugayenko • 7 months ago

Well, going "The OOP way" all the time (or wanting to) could result mostly in over-engineering, but the worst outcome would be to start suffering from analysis paralysis. If you reach that point, you're bound for quite an important refactor in your application to drop in a few accessors in order to come back to the comfort zone and regain control. I used to be myself a die-hard fan of the Tell, Don't Ask principle, but for my own experience it turns out sometimes the benefit you get is not worth the extra effort you have to put into expressing each and every role and relationship in a way that makes you feel you are exclusively "telling" others to do stuff for you, and not exposing anything to the outer world. I don't think any system you could ever possibly develop is worth that (much) pain ;)

1 ^ | v • Reply • Share ›

Yegor Bugayenko author → pjmartos • 7 months ago

Well, I have a few Java products, developed recently - they don't have any getters (or setters). They are not huge, but somewhere around 10-15K lines of code. But I got your point and I agree - it may be very difficult to stay excellent on a large scale :)

^ | v • Reply • Share ›

An0nym0usC0ward • 7 months ago

I think you meant something else. You meant that getters and setters are they are used nowadays are evil. But think of the classic example of shapes in a drawing model. You may want to be able to recolor shapes. Whether you say Shape.paint(red) or Shape.setColor(red) is a matter of idiom. However, paint() is IMO an absolutely legitimate method of a shape object. Now, when you want to get the color of the shape, you can call

method of a shape object. Now, when you want to get the color of the shape, you can call the getter simply color(), or getColor() - again, it's a matter of idiom. Fact is, there are several scenarios where getting the color might be useful - like for instance having to identify a brand new, not already used color for a new shape. Either way, you definitely don't want to make the internal, private member that holds the color public (if there is such a member at all).

It's just that the Java beans idiom of setters and getters has become so ingrained that we no longer think about properly naming getters and setters to reflect their actual meaning in terms of the logic they provide, and instead name them simply based on what they do - which is indeed bad.

1 ▲ ▼ • Reply • Share ›

Yegor Bugayenko author ➦ An0nym0usC0ward • 7 months ago

Yes, it's a matter of idiom, which affects our entire thinking. Our mindset. This is how we think when getters and setters is a legitimate practice:

```
Pixel pixel = new Pixel();
pixel.setColor(Color.RED);
pixel.setX(50);
pixel.setY(100);
void place(Pixel pixel, Screen screen) {
  int color = pixel.getColor();
  int x = pixel.getX();
  int y = pixel.getY();
  screen.draw(x, y, color);
}
```

In this example, class `Pixel` is a mindless carrier of data. It's not an object in terms of OOP. But we are **forced** to make it like this because of the framework written for us (method `place()` is provided by the UI framework). We have to give an instance of this `Pixel` to the UI framework, so that it can extract X/Y/Color from it and put it on

see more

▲ ▼ • Reply • Share ›

Hlodowig ➦ Yegor Bugayenko • 6 months ago

Sure, but that works with simple objects with three properties. More complex things can become unbearable if you blindly handle them that way.

▲ ▼ • Reply • Share ›

Yegor Bugayenko author ➦ Hlodowig • 6 months ago

If you follow this approach, you won't have "more complex things". See this article, it explains how immutability and absence of getters/setters helps in making better design: http://www.yegor256.com/2014/1...

▲ ▼ • Reply • Share ›

oddparity ➦ Yegor Bugayenko • 7 months ago

So, the Person object would have a method called presentOnScreen()? And then someone comes along and wants annother form of view, say for an overview screen: presentOnOverviewScreen(). And then comes printing and reporting to your system, and we get presentOnReceipt(), presentOnCustomerReport(). Later there is exportToXMLForSAPDataExchange(). Where does that go?

3 ▲ | ▼ • Reply • Share ›

Yegor Bugayenko author ➚ oddparity • 7 months ago

Yes, exactly like that. We have a `Serializable` interface in Java, which does exactly what your `presentOnScreen()` would do. When an object implements this interface, it can serialize itself into a data/byte stream. So, only the object knows how to serialize itself. The same with the screen - only the object can know how to preset itself on the screen. If you have to make your object so smart and capable of presenting itself on multiple media (screen, XML, receipt, report, etc) - start thinking about refactoring. The object is too big and has too much responsibility.

▲ | ▼ • Reply • Share ›

An0nym0usC0ward ➚ Yegor Bugayenko • 7 months ago

:-) How might that draw() operation's implementation look? I can think of something like calling screen.drawAt(this.x, this.y, this.color) - that's IMO not much different from screen.getPixels()[this.x][this.y] = this.color;. I do like screen.drawAt() more, but I don't think screen.getPixelArray()[][] = ... is that much worse.

IME you will always have some data exposed between collaborating objects, unless you use a purely functional style and pass only tuples around (giving up much of the type safety Java provides). And IMO that's OK, as long as it stays within reasonable limits - i.e. it only happens among classes which are inherently very close to each other. When you add getters and setters by default for each private field of an object, and make all of them public, you definitely no longer stay within a reasonable limit.

OTOH, when you do have a situation where too much of the data is both exposed and used via thin wrappers like getters and setters, you probably haven't gotten your abstractions right. In the pixel/screen example above, I don't know if the Pixel class makes sense.

On yet another hand, there may be good reason why C++ uses the distinct concepts of class and struct. Struct, seen as a dumb, dead, lifeless alternative to class, may have its reason to exist - sometimes you are better off passing around clumps of data, instead of encapsulating them in objects - encapsulating clumps of data is always possible, it sometimes just isn't the most economical or convenient thing to do. Java doesn't have struct, and consequently doesn't allow the semantic distinction via how a class is declared. Which is why occasionally you need to use class as a struct (not

as frequent as most of the code I've seen does, though - the typical abomination are @Entity-annotated classes serialized via an ORM when what you should have instead is a functionality-rich model).

• Reply • Share ›

Yegor Bugayenko author ➚ An0nym0usC0ward • 7 months ago

"stay within a reasonable limit" - this is where the problem is. I've never seen any code that managed to stay there :) In reality, getters-and-setters style takes over your code once you start using it. Like a tumor it will eat your entire code, unless you cut it off at the early beginning :)

BTW, I think that C++ structs is a terrible mistake in language design. As well as operator reloading and friend classes.

• Reply • Share ›

An0nym0usC0ward ➚ Yegor Bugayenko • 7 months ago

You seem to consider that it's a good practice to reduce the expressive power of a language just because some features can be misused. If that was true, we'd still have no generics in Java (not even the botched implementation currently available - no type checking at runtime). Besides, that's somewhat offensive - it treats programmers like irresponsible little children who can't be trusted to act maturely.

There's more than just friend classes wrong with the initial spec of C++, which, due to backwards compatibility requirements, still stays there. But operator overloading is definitely not something wrong. Just go ahead and write libraries for working with sets (union, intersection etc.), for complex numbers, for matrix calculations, then compare code written on top of them with and without operator overloading.

As for staying within reasonable limits never happening: by formally forbidding setters and getters you'd just get developers to be creative about naming, not about the practice per se. Worse, accessor and mutator infested code is a small problem, when compared to other code stenches, like long methods or shotgun surgery (or most of the other stenches described by Fowler in the introductory part of his book on refactoring). If you want developers to write good code, educate them to make sound judgments, and trust them that they made the right call when deciding to add a getter or a setter. Anything else, such as trying to enforce an extreme rule, will fail - the same way the prohibition failed in the US a few decades ago, proving that probably any extreme cure is worse than the disease.

3 • Reply • Share ›

Ivano Pagano ➚ An0nym0usC0ward • 7 months ago

I don't agree.
What you say seems sensible, but for me the issue is another.

The wide-spread adoption of java classes whose role is just that of "property holders" with no logic, is the downside of the javabean conventions.
While I don't adhere to Yegor's doctrine, I agree that the habit of depriving objects from their behavioral parts is indeed against what OOP should be used for.
An object is behaviour attached to data (hidden or not). Otherwise I'd say that Object Orientation becomes pointless.

I still find that from a practical POV some application context can be correctly served by property-holders (CRUD, UIs).
But this choice should not be taken for granted and based only on thoughtless defaulting.
And in any case getter/setter don't adhere to Uniform Access Principle.

This is where I stand on this debate.

^ | v • Reply • Share ›

Yegor Bugayenko author ➚ Ivano Pagano • 7 months ago

Both CRUD and UIs can and must be expressed in object-oriented philosophy. Again, there are no property-holders in OOP. Each object is a living organism. An organism (a dog, for example) can carry some other organism for you (a ball, for example). In this case we get ask her to give this ball back to us. But we can't call the dog a ball-holder. It's offensive :)

^ | v • Reply • Share ›

Karl • 7 months ago

Mutable properties are the heart and soul of any UI toolkit I know, in any language, all along history. Stuff like setColor("red"), setEnabled(false), setFont("sans-serif") are not evil. In fact, there is no other sane way for expressing stuff like that.

So please give a hint how to implement a UI toolkit without get/set, instead of talking about dogs. Otherwise this debate is futile. Period.

2 ^ | v • Reply • Share ›

Yegor Bugayenko author ➚ Karl • 7 months ago

See my answer above with an example. `setColor()` assumes that somewhere else we have `getColor()`. This means that our object is a mindless carrier of data. Yes, we're **forced** to use that mindless carriers of data because of frameworks created with this incorrect concept in mind. However, if we're writing something new, we should avoid setters/getters.

^ | v • Reply • Share ›

Karl ➚ Yegor Bugayenko • 7 months ago

You really should have a deeper look on the internals of UI toolkits and their complexity. Qt's QWidget, Swing's JComponent, the new JavaFX Node class, or the ubiquitous HTML DOM Element are all crammed with getters

and setters for guiding layout, geometry, textual appearance and styling of a certain UI element. A screen/scene is typically represented by a highly complex tree structure of such objects. So these objects naturally carry *a lot* of data, and to some degree they seem to be mindless. But that's ok, because their primary purpose is to represent a structural model of the UI.

It would be interesting how advocates of the "anti-getter/setter"-camp would implement a comparable toolkit. If it cannot be done in a sane way, then the paradigm is simply not worth being discussed further, because it has failed in central area of programming. Period.

1 • Reply • Share ›

Oliver Doepner ↗ Karl • 4 months ago

One thing I did when I had to design a web UI layer of a large application : I used a generic ValueHolder<t> class with set(T value) and T get() method. Subclasses of this base type were used for all leaves of the UI model tree. This way the value is still mutable, but the holder objects can be long-lived parts of a mostly immutable model tree.

With this approach, I never used setEnabled(), setEditable(), setRendered(), etc. Instead, I set boolean callbacks on the value holders to implement the conditions of enabled, editable, rendered, as injectable permanent code blocks, not mutable fields. Similar things can be done for conditional color changes and the like. I can give more concrete code examples if anyone is interested.

1 • Reply • Share ›

Yegor Bugayenko author ↗ Karl • 7 months ago

Yes, it would be interesting to see, I agree. My teams never worked with projects like that before (like Swing or JavaFX), but once we have something like that, I'll write about our results.

ps. Maybe this article may help: http://www.yegor256.com/2014/1...

• Reply • Share ›

Ivano Pagano • 8 months ago

To all people disagreeing with this article on the basis that "real work" must be accomplished, I guess you miss the main point of the article, which is to state that OOP as developed by the mainstream community (java, c#) in the last 15-so years is basically flawed...

If you do search and read some articles about flaws in oop paradigm you'll see that a global discussion is starting to take off about the failure of many models (e.g. patterns, ORM, DI frameworks) that we, as oop devs, take for granted.
If you care, read this: http://www.smashcompany.com/te...

As a side note: "large scale applications" have nothing to do with accessors... there are many apps working large-scale today that doesn't even use OOP.

To me get/set is mostly a convention established to ease up generic GUI frameworks' development that has become a standard even when not needed.

2 • Reply • Share ›

Yegor Bugayenko author ↗ Ivano Pagano • 8 months ago

Totally agree, thanks.

• Reply • Share ›

Pierre-Yves Saumont • 8 months ago

You are basically right when you say that objects are not (only) data structures, but with all due respect, you fail at explaining why. When you write:

```
dog.take(new Ball());
Ball ball = dog.give();
```

It is not different from:

```
dog.setBall(new Ball());
Ball ball = dog.getBall();
```

I understand you mean that it is the dog that is acting when it gives you the ball, so one should not call the
method getBall(). The dog gives the ball. You don't take it from it. But this is totally irrelevant. The main characteristic of objects in OOP is polymorphism. This lets you create an interface Animal:

```
public interface Animal {
void talk();
}
```

You may the define subclasses:

```
public class Dog implement Animal {
void talk() {
System.out.println("Ouah, Ouah!");
}
}

public class Bird implement Animal {
void talk() {
System.out.println("Cui, Cui!");
}
}
```

Now you may use them without knowing their real type:

```
Animal animal = petShop.getAnimal();
animal.talk();
```

If objects where only data structures, you would have to write:

```
Animal animal = petShop.getAnimal();
if (animal instanceof Dog) {
System.out.println("Ouah, Ouah!");
} else if (animal instanceof Bird) {
System.out.println("Cui, Cui!");
}
```

You can derive much more powerful patterns from polymorphism, such as inversion of control. But in my opinion,
polymorphism is a failure because you cant do the following:

```
Animal animal = petShop.getAnimal();
play(animal);
void play(Dog animal) {
// play with the dog;
}

void play(Bird animal) {
// play with the bird;
}
```

It would be great if Java could figure what real type is "animal" and call the corresponding method. Instead of that,
you have to write:

```
Animal animal = petShop.getAnimal();
play(animal);
if (animal instanceof Dog) {
Dog dog = (Dog) animal;
play(dog);
} else if (animal instanceof Bird) {
Bird bird = (Bird) animal;
play(bird);
}

void play(Dog animal) {
// play with the dog;
}

void play(Bird animal) {
```

```
// play with the bird;
}
```

This is horrible. Another solution is to use the visitor pattern, which is even more horrible since you have
to modify your classes in order to use it. (Or you can use introspection!) This one reason, in my opinion,
why objects should not be consider anything more that data structures and code repositories
(which, by the way, are very useful).

Another reason is that polymorphism don't work well with parameterized types. A List<dog> is not a subtype of List<animal>. To use a different example, think about implementing a flatten() method in List that would return the list itself if elements are not lists,but a flattened version if elements are themselves list (List<list<element> should return a List<element> being the concatenation of all sublists). This is very easy to do with a static method. Not so with an instance method.

  • Reply • Share ›

Yegor Bugayenko author ➚ Pierre-Yves Saumont • 8 months ago

You mentioned two interesting problems, and I totally agree that fixing them would be a great improvement in Java, for example. But do we have to abandon the entire object-oriented paradigm just because Java has a few flaws?

About your comment that it's totally irrelevant what name to use `getBall()` or `give()` I have to agree that it is absolutely irrelevant if you are thinking like a computer. If you look at your object as a glass box data structure. Not even a glass box, but a heap of broken glass where you should search and find what you need. In this case, getters is the right way to search that heap.

However, this is not what object oriented programming is about. OOP encourages us to anthropomorphize our object - to look at them as living organisms. They talk, they answer, they behave, but they never show us what is inside. They are black boxes for us. If/when you start thinking like this you will see that getters as an insult to your objects. Your objects will get offended by this getter/setter attitude :)

  • Reply • Share ›

oddparity ➚ Yegor Bugayenko • 7 months ago

Is this what you personally heard from the "inventors" of OOP? Perhaps they just thought: Oh, there are those bad structs, and if I have a person struct and a customer struct, I have to write the code of the person struct again where I define the customer struct (no ineritance). And then there are functions doing something with my person structs, and since there are no braces embracing the structs and the functions, coders don't easily know that they belong together. And then I have to implement something which could work with a person struct or a customer struct, but how do I declare the function? What if someone invented OOP with these simple problems in mind and only later someone began to see living things in those objects and called objects which were merely pure data holders "anemic"?

called objects which were merely pure data holders "anemic"?

• Reply • Share ›

Yegor Bugayenko author ➚ oddparity • 7 months ago

Good point :) I don't really care what "inventors" or OOP had in mind. I'm using object-oriented programming the way I understand it and I'm trying to prove my way of thinking with logic. Opinions, no matter how authoritative they are, don't count. For example, in his InversionOfControl article Martin Fowler says: "*A library is essentially a set of functions that you can call, these days usually organized into classes*". With all due respect, functions are not organized into classes. This statement completely ruins the entire idea of OOP. Can we blindly trust everything else Mr. Fowler says? No, we can't.

• Reply • Share ›

oddparity ➚ Yegor Bugayenko • 7 months ago

Well, if you look at the reality around your ecosystem of the "true OOP way", what Mr. Fowler says is true. In your utility classes article you complain about the Jakarta commons and the like. Usually these libraries organize their helpers in classes. What else could they do, while programming in Java? But even if Mr. Fowler expressed it that way, it doesn't tell us if he calls this OOP. Perhaps he would share your critics.

• Reply • Share ›

Yegor Bugayenko author ➚ oddparity • 7 months ago

Perhaps he would :)

• Reply • Share ›

Bruce Phillips • 8 months ago

In your example how does the Dog update its weight when it changes?

1 • Reply • Share ›

Yegor Bugayenko author ➚ Bruce Phillips • 8 months ago

I am in favor of immutable objects, which do not change their state ever. Take a look: http://www.yegor256.com/2014/0... When it's time to change the weight, we will have a new dog, with a new weight.

• Reply • Share ›

David ➚ Yegor Bugayenko • 7 months ago

What happen if the gain weight? Do I need to create a new dog just because it decided to eat more?

2 • Reply • Share ›

Bruce Phillips ➚ Yegor Bugayenko • 8 months ago

I'm writing an application for the world's largest kennel - 1,000,000 dogs stay there. Each day the kennel must weigh the dogs and record their new weight. I want to use JPA to manage persistence/simplify CRUD operations

weight. I want to use JPA to manage persistence/simplify CRUD operations.

Doesn't your concept of creating a new Dog with the new weight make using JPA much more difficult?

Isn't the concept of ORM built partially on changing the state of existing Objects?

I don't follow how your ideas would work in large-scale real-world applications.

5 ▲ | ▼ • Reply • Share ›

Yegor Bugayenko author ➚ Bruce Phillips • 8 months ago

I'm planning to write an article about ORM and why they are also a bit evil in terms of OOP. JPA (I had to use it in a few projects) as well as JavaBeans are two good examples of why setters/getters should be avoided. You may also find this article relevant: http://www.yegor256.com/2014/1...

How about this:

```
final class Kennel {
  void weighThemAll(Journal journal) {
    for (Dog dog : this.dogs()) {
      journal.record(dog, dog.weight());
    }
  }
}
```

Or I didn't get your idea about the kennel?

▲ | ▼ • Reply • Share ›

Sihle ➚ Yegor Bugayenko • 7 months ago

Yegor, drawing from the pixel example above that you provided, wouldn't it make more sense to have

final class Kennel {
void weighThemAll(Journal journal) {
for (Dog dog : this.dogs()) {
dog.record(journal);
}
}
}

1 ▲ | ▼ • Reply • Share ›

Yegor Bugayenko author ➚ Sihle • 7 months ago

Yes, absolutely. You got the idea! :)

▲ | ▼ • Reply • Share ›

Tom Whitmore ➚ Yegor Bugayenko • 7 months ago

What you're doing here, is separating the dog's _state_ from it's _identity_; and recording a versioned history of state.

This is a valid design, for some purposes -- but such design choices have effects. For example, the dog no longer has a current weight.. unless it's attached to a 'journal' object. What about which kennel the dog is housed in? It's owner (ownership can be transferred)? The owner's home address/ phone number? What food it should be fed? And how much? Etc etc etc.

All of a sudden, your proposed design is becoming amazingly ugly.

Storing "current state" is accepted and efficient practice both in the database, and in memory. Versioned histories are fine -- where required -- but are normally preferable to store in a separate table or data-structure, because of their cost & management requirements (archiving etc).

What use is a "Dog" aka 'dog information record' if it doesn't have any information?

And for shared responsibilities -- feeding the dog, contacting the owner -- these must be done in collaborating with food, email service etc. The dog does not feed itself, nor phone it's owner.

1 ^ | v • Reply • Share ›

Bruce Phillips ➚ Yegor Bugayenko • 8 months ago

Your example code won't work - it only records the Dog's previous weight not the Dog's new weight (assuming dogs is a collection of Dog objects that were created based on information [e.g. dogid, dog weight] stored in a data repository).

Remember we need to store each dog's new weight in the database and associate that weight with the specific dog.

So following your technique after we weigh the dog we would need to create a new Dog object with the new weight and the old Dog object's dogid. Your process of not allowing an object's state to change leads to more verbose and harder to follow code.

In the real world we use ORM to speed up development and to work at higher level of abstraction - which are not evil -- but get the job done well.

4 ^ | v • Reply • Share ›

Sihle ➚ Bruce Phillips • 7 months ago

But your assumptions are based on matching database IDs with Object IDs, which doesn't scale well. That's why ORMs are considered bad. The database provides a historical context of Dog states, which is opposite to the Object goal of giving you things as

states, which is opposite to the Object goal of giving you things as they are right now. As another example: When I make chess moves, I only care about what's possible right now to win, not what happened before. Each move gives me a new game situation with new possibilities. The old situation means nothing for my ability to win. Just a thought...

1 • Reply • Share ›

oddparity ➚ Sihle • 7 months ago

False. Professional players record the moves and analyze later. And they want to know then which of the two towers moved somewhere. Would you design that by storing complete game situations? No, not really.

• Reply • Share ›

Sihle ➚ oddparity • 7 months ago

Recording and analysis is an important part of the enhancing oneself (chesspiece.record(chesspiece.move())), but to win the current game you need to look at possibilities in front you.

• Reply • Share ›

Николай Иванчев ➚ Yegor Bugayenko • 8 months ago

Just a second.. Does it mean that dog.eat() should return a new dog? Because hey.. living dogs are mutable.. I am also in favor of non-mutable objects, but when it makes sense. 100% of everything is evil

3 • Reply • Share ›

Николай Иванчев ➚ Yegor Bugayenko • 8 months ago

That says it all! I rest my case.. What's wrong with the old dog? She has gained a little weight and now is time for a new dog.

2 • Reply • Share ›

Николай Иванчев • 8 months ago

Have you ever written something serious without get/set. How the idea of not generating setters sounds to you when it comes to immutability. Let me ask you are question.. Lets say we have a dog dentist.. How exactly - dentist.extractTooth(dog) would work? Don't tell me - dogs are immutable and that makes them immune to teeth and gum issues..

• Reply • Share ›

Yegor Bugayenko author ➚ Николай Иванчев • 8 months ago

Well, I haven't written a single getter/setter in the last two years. I wrote them before, I have to confess. Yes, dogs are immutable. Check this article: http://www.yegor256.com/2014/0...

• Reply • Share ›

Николай Иванчев ➚ Yegor Bugayenko • 8 months ago

You are missing a point.. I didn't ask you about writing getters and setters.. I asked you about doing writing something serious

asked you about doing writing something serious..

1 • Reply • Share ›

Yegor Bugayenko author ➚ Николай Иванчев • 8 months ago

Every line of code I write I take very seriously :) You can see a list of open source objects I personally took participation in, as a hands-on developer and an architect: http://www.teamed.io/portfolio...

ps. Look at this article, it explains a rather serious library designed recently from purely immutable objects: http://www.yegor256.com/2014/1...

• Reply • Share ›

Crak Lod Mes • 8 months ago

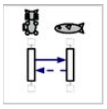
I see that you hate beans

2 • Reply • Share ›

Yegor Bugayenko author ➚ Crak Lod Mes • 8 months ago

Can't stand them at all

• Reply • Share ›



umlcat • 8 months ago

I may change the article to "getters and setters may be evil used" instead ...

• Reply • Share ›

Yura Nakonechnyy • 8 months ago

Very decent article, totally agree that many claimed OOP-professionals doesn't really understand the core concept behind OOP

1 • Reply • Share ›

Yegor Bugayenko author ➚ Yura Nakonechnyy • 7 months ago

You may find this article interesting too: http://www.yegor256.com/2014/1...

• Reply • Share ›

Mantas • 8 months ago

Never tried to ask dog for its weight, usually I put dog on scales and scales shows the weight :D But I will try, maybe dog will answer :)

3 • Reply • Share ›

Yegor Bugayenko author ➚ Mantas • 8 months ago

Let me know how it ends :) But even with scales, you're not "getting" weight from it, but asking it for the weight:

```
int weight = new Scales().howBigIs(dog);
```

1 • Reply • Share ›

Maxim Konstantinovski ➚ Yegor Bugayenko • 8 months ago

Finding a good name for any thing in code can be quite difficult at times.

int weight = new Scales().measure(dog);
Or prehaps evaluate(dog) or simply weigh(dog). The last on probebly the best.

• Reply • Share ›

Николай Иванчев ➚ Maxim Konstantinovski • 8 months ago

Since weight is property of the Dog and guess what.. dog can't tell her weight.. That is going to be fun..
It is always sunny in California when you talk the big picture.. The evil is the gory details when it comes to actually paint it..

3 • Reply • Share ›

Massimiliano Tomassi ➚ Николай Иванчев • 8 months ago

Actually in the article he says: "We can ask a dog to give us some piece of data (for example, her weight), and she may return us that information". Why do you say that the dog can't tell her weight?

• Reply • Share ›

Николай Иванчев ➚ Massimiliano Tomassi • 8 months ago

Show me a dog that can.

• Reply • Share ›

Yegor Bugayenko author ➚ Николай Иванчев • 8 months ago

In OOP we should anthropomorphize objects. That means that every object is like a living creature. A truck, a dog, a house, a road, a bottle, a weight, a temperature, a date, a file - they are all living organisms. They all talk and listen. They communicate. So, yes, a dog can tell its weight :)

1 • Reply • Share ›

Николай Иванчев ➚ Yegor Bugayenko • 8 months ago

A talking dog.. Aaargh.
Now seriously.. The implementation of weight() method is the same as getWeight(). So now what now.. Getters are evil but implementing them and naming them otherwise isn't.
Properties are properties, they aren't objects. This is a rule of OOP. Immutability is not. You are mixing concepts. But If the article was 'setters are evil in immutable data', there wouldn't be an article, right?

2 • Reply • Share ›

Andrej Istomin ➚ Николай Иванчев • 7 months ago

>> 'setters are evil in immutable data'
Hmm... What about 'mutable data is evil'? Somebody will say that our world is mutable... but it's a question of philosophy :) Heraclitus said that our world is really immutable and 'no man ever steps in the same river twice'. Really, mutability is cause of many problems especially

in large scale applications.

Alan Artigao • 8 months ago

I think it's a simple convention. I can't imagine working with different frameworks and different domain model objects that have particular function names to access/modify its fields. Seems a way to complicate programming. Anyway, thanks for the reference book, i'll