http://www.yegor256.com/2014/09/16/getters-and-setters-are-evil.html

Getters/Setters. Evil. Period.

16 September 2014 modified on 24 October 2014 Yegor Bugayenko

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-oriented programming. I'll try to add my two cents to this discussion.

The gist of the following text is this: getters and setters is a terrible practice and those who use it can't be excused. Again, to avoid any misunderstanding, I'm not saying that get/set should be avoided when possible. No. I'm saying that you should **never** have them near your code.

Arrogant enough to catch your attention? You've been using that get/set pattern for 15 years and you're a respected Java architect? And you don't want to hear that nonsense from a stranger? Well, I understand your feelings. I felt almost the same when I stumbled upon Object Thinking by David West, the best book about object-oriented programming I've read so far. So please. Calm down and try to understand while I try to explain.

Existing Arguments

There are a few arguments against "accessors" (another name for getters and setters), in an object-oriented world. All of them, I think, are not strong enough. Let's briefly go through them.

Tell, Don't Ask Allen Holub says, "Don't ask for the information you need

to do the work; ask the object that has the information to do the work for you".

Violated Encapsulation Principle An object can be teared apart by other objects, since they are able to inject any new data into it, through setters. The object simply can't encapsulate its own state safely enough, since anyone can alter it.

Exposed Implementation Details If we can get an object out of another object, we are relying too much on the first object's implementation details. If tomorrow it will change, say, the type of that result, we have to change our code as well.

All these justifications are reasonable, but they are missing the main point.

Fundamental Misbelief

Most programmers believe that an object is a data structure with methods. I'm quoting Getters and Setters Are Not Evil, an article by Bozhidar Bozhanov:

> *But the majority of objects for which people generate getters and setters are simple data holders.*

This misconception is the consequence of a huge misunderstanding! Objects are not "simple data holders". Objects are **not** data structures with attached methods. This "data holder" concept came to object-oriented programming from procedural languages, especially C and COBOL. I'll say it again: an object is **not** a set of data elements and functions that manipulate them. An object is **not** a data entity.

What is it then?

A Ball and A Dog

In true object-oriented programming, objects are living creatures, like you and me. They are living organisms, with their own behaviour, properties and a life cycle.

Can a living organism have a setter? Can you "set" a ball to a dog? Not really. But that is exactly what the following piece of software is doing:

```
Dog dog = new Dog();
dog.setBall(new Ball());
```

How does that sound?

Can you get a ball from a dog? Well, you probably can, if she ate it and you're doing surgery. In that case, yes, we can "get" a ball from a dog. This is what I'm talking about:

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

Or an even more ridiculous example:

```
Dog dog = new Dog();
dog.setWeight("23kg");
```

Can you imagine this transaction in the real world? :)

Does it look similar to what you're writing every day? If yes, then you're a procedural programmer. Admit it. And this is what David West has to say about it, on page 30 of his book:

> *Step one in the transformation of a successful procedural developer into a successful object developer is a lobotomy.*

Do you need a lobotomy? Well, I definitely needed one and received it,

while reading West's Object Thinking.

Object Thinking

Start thinking like an object and you will immediately rename those methods. This is what you will probably get:

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

Now, we're treating the dog as a real animal, who can take a ball from us and can give it back, when we ask. Worth mentioning is that the dog can't give `NULL` back. Dogs simply don't know what `NULL` is :) Object thinking immediately eliminates NULL references from your code.


© A Fish Called Wanda (1988) by Charles Crichton

Besides that, object thinking will lead to object immutability, like in the "weight of the dog" example. You would re-write that like this instead:

```
Dog dog = new Dog("23kg");
int weight = dog.weight();
```

The dog is an immutable living organism, which doesn't allow anyone from the outside to change her weight, or size, or name, etc. She can tell, on request, her weight or name. There is nothing wrong with public methods that demonstrate requests for certain "insides" of an object. But these methods are not "getters" and they should never have the "get" prefix. We're not "getting" anything from the dog. We're not getting her name. We're asking her to tell us her name. See the difference?

We're not talking semantics here, either. We are differentiating the procedural programming mindset from an object-oriented one. In procedural programming, we're working with data, manipulating them, getting, setting, and deleting when necessary. We're in charge, and the data is just a passive component. The dog is nothing to us — it's just a "data holder". It doesn't have its own life. We are free to get whatever is necessary from it and set any data into it. This is how C, COBOL, Pascal and many other procedural languages work(ed).

On the contrary, in a true object-oriented world, we treat objects like living organisms, with their own date of birth and a moment of death — with their own identity and habits, if you wish. We can ask a dog to give us some piece of data (for example, her weight), and she may return us that information. But we always remember that the dog is an active component. She decides what will happen after our request.

That's why, **it is conceptually incorrect to have any methods starting with `set` or `get` in an object**. And it's not about breaking encapsulation, like many people argue. It is whether you're thinking like an object or you're still writing COBOL in Java syntax.

PS. Yes, you may ask, — what about JavaBeans, JPA, JAXB, and many other Java APIs that rely on the get/set notation? What about Ruby's built-in feature that simplies the creation of accessors? Well, all of that is our misfortune. It is much easier to stay in a primitive world of procedural COBOL than to truly understand and appreciate the beautiful world of true

objects.

PPS. Forgot to say, yes, dependency injection via setters is also a terrible anti-pattern. About it, in one of the next posts :)