Why use getters and setters/accessors?

Asked 12 years, 10 months ago Modified 5 months ago Viewed 509k times

1780

803

What's the advantage of using getters and setters - that only get and set - instead of simply using public fields for those variables?

If getters and setters are ever doing more than just the simple get/set, I can figure this one out very quickly, but I'm not 100% clear on how:

```
public String foo;
```

is any worse than:

```
private String foo;
public void setFoo(String foo) { this.foo = foo; }
public String getFoo() { return foo; }
```

Whereas the former takes a lot less boilerplate code.

java oop setter getter abstraction

Share Edit Follow

edited Dec 6, 2017 at 15:45
Tim
40.1k 18 126 142

asked Oct 14, 2009 at 18:20
user Dean J
38k 16 65 93

8 @Dean J: Duplicate with many other questions: stackoverflow.com/search?q=getters+setters – Asaph Oct 14, 2009 at 18:26

9 Of course, both are equally bad when the object doesn't need a property to be changed. I'd rather make everything private, and then add getters if useful, and setters if needed. – Tordek Oct 14, 2009 at 18:29

31 Google "accessors are evil" – OMG Ponies Oct 14, 2009 at 18:45

39 "Accessors are evil" if you happen to be writing functional code or immutable objects. If you happen to be writing stateful mutable objects, then they are pretty essential. – Christian Hayter Oct 14, 2009 at 19:10

22 Tell, don't ask. pragprog.com/articles/tell-dont-ask – Dave Jarvis Oct 14, 2009 at 22:18

41 Answers

Sorted by:
Trending sort available
Highest score (default)

1 2 Next

1163

There are actually *many good reasons* to consider using accessors rather than directly exposing fields of a class - beyond just the argument of encapsulation and making future changes easier.

Here are the some of the reasons I am aware of:

- Encapsulation of behavior associated with getting or setting the property - this allows additional functionality (like validation) to be added more easily later.
- Hiding the internal representation of the property while exposing a property using an alternative representation.
- Insulating your public interface from change - allowing the public interface to remain constant while the implementation changes without affecting existing consumers.
- Controlling the lifetime and memory management (disposal) semantics of the property - particularly important in non-managed memory environments (like C++ or Objective-C).
- Providing a debugging interception point for when a property changes at runtime - debugging when and where a property changed to a particular value can be quite difficult without this in some languages.
- Improved interoperability with libraries that are designed to operate against property getter/setters - Mocking, Serialization, and WPF come to mind.
- Allowing inheritors to change the semantics of how the property behaves and is exposed by overriding the getter/setter methods.
- Allowing the getter/setter to be passed around as lambda expressions rather than values.
- Getters and setters can allow different access levels - for example the get may be public, but the set could be protected.

Share Edit Follow

edited Oct 24, 2013 at 6:25
user Dennis Meng **5,002** 14 32 36

answered Oct 14, 2009 at 18:47
user LBushkin **126k** 32 212 261

1 @andreagalle There's many reason that could be the case. Maybe you have an attribute that is not to be changed from the outside. In that scenario you could also completely remove the setter. But maybe you want to change this change-blocking behaviour in a subclass, which could then utilize the `protected` method. – 0xff Nov 1, 2021 at 18:22

3 -1 Because point no. 1 is the dev trap #1: Classical Getters / Setters do not encapsulate behaviour. Because they do not express behaviour. Cutomer::setContractEndDate() / Customer::getContractEndDate() is no behaviour and it does not encapsulate anything at all. It's faking encapsulation. Customer::cancelContract() is a behaviour and that's where you actually can modify the logic without changing the interface. It won't work if you make atomic properties public through getters and setters. – Eugene Feb 13 at 1:24

Because 2 weeks (months, years) from now when you realize that your setter needs to do **more**

607 than just set the value, you'll also realize that the property has been used directly in 238 other classes :-)

Share Edit Follow

answered Oct 14, 2009 at 18:23

user ChssPly76
98k 24 205 195

114 I'm sitting and staring at a 500k line app where it's never been needed. That said, if it's needed once, it'd start causing a maintenance nightmare. Good enough for a checkmark for me. – Dean J Oct 14, 2009 at 18:24

22 I can only envy you and your app :-) That said, it really depends on your software stack as well. Delphi, for example (and C# - I think?) allows you to define properties as 1st class citizens where they can read / write a field directly initially but - should you need it - do so via getter / setter methods as well. Mucho convenient. Java, alas, does not - not to mention the javabeans standard which forces you to use getters / setters as well. – ChssPly76 Oct 14, 2009 at 18:27

20 While this is indeed a good reason for using accessors, many programming environments and editors now offer support for refactoring (either in the IDE or as free add-ins) which somewhat reduces the impact of the problem. – LBushkin Oct 14, 2009 at 18:59

86 sounds like pre-mature optimization – cmcginty Jun 15, 2012 at 21:31

46 The question you need to ask when you wonder whether to implement getters and setters is: *Why would users of a class need to access the class' innards at all?* It doesn't really matter whether they do it directly or shielded by a thin pseudo layer — if users need to access implementation details, then that's a sign that the class doesn't offer enough of an abstraction. See also this comment. – sbi Aug 23, 2012 at 21:13

422 A public field is not worse than a getter/setter pair that does nothing except returning the field and assigning to it. First, it's clear that (in most languages) there is no functional difference. Any difference must be in other factors, like maintainability or readability.

An oft-mentioned advantage of getter/setter pairs, isn't. There's this claim that you can change the implementation and your clients don't have to be recompiled. Supposedly, setters let you add functionality like validation later on and your clients don't even need to know about it. However, adding validation to a setter is a change to its preconditions, **a violation of the previous contract**, which was, quite simply, "you can put anything in here, and you can get that same thing later from the getter".

So, now that you broke the contract, changing every file in the codebase is something you should want to do, not avoid. If you avoid it you're making the assumption that all the code assumed the contract for those methods was different.

If that should not have been the contract, then the interface was allowing clients to put the object in invalid states. *That's the exact opposite of encapsulation* If that field could not really be set to anything from the start, why wasn't the validation there from the start?

This same argument applies to other supposed advantages of these pass-through getter/setter pairs: if you later decide to change the value being set, you're breaking the contract. If you override the default functionality in a derived class, in a way beyond a few harmless modifications (like logging or other non-observable behaviour), you're breaking the contract of the base class. That is a violation of the Liskov Substitutability Principle, which is seen as one of the tenets of OO.

If a class has these dumb getters and setters for every field, then it is a class that has no invariants whatsoever, *no contract*. Is that really object-oriented design? If all the class has is those getters and setters, it's just a dumb data holder, and dumb data holders should look like dumb data holders:

```
class Foo {
public:
    int DaysLeft;
    int ContestantNumber;
};
```

Adding pass-through getter/setter pairs to such a class adds no value. Other classes should provide meaningful operations, not just operations that fields already provide. That's how you can define and maintain useful invariants.

> **Client**: "What can I do with an object of this class?"
> **Designer**: "You can read and write several variables."
> **Client**: "Oh... cool, I guess?"

There are reasons to use getters and setters, but if those reasons don't exist, making getter/setter pairs in the name of false encapsulation gods is not a good thing. Valid reasons to make getters or setters include the things often mentioned as the potential changes you can make later, like validation or different internal representations. Or maybe the value should be readable by clients but not writable (for example, reading the size of a dictionary), so a simple getter is a nice choice. But those reasons should be there when you make the choice, and not just as a potential thing you may want later. This is an instance of YAGNI (*You Ain't Gonna Need It*).

Share Edit Follow

edited Feb 17, 2015 at 13:26
Lightness Races in Orbit
371k 74 623 1026

answered Aug 24, 2012 at 10:55
R. Martinho Fernandes
221k 71 425 503

19 Great answer (+1). My only criticism is that it took me several reads to figure out how "validation" in the final paragraph differed from "validation" in the first few (which you threw out in the latter case but promoted in the former); adjusting the wording might help in that regard. – Lightness Races in Orbit Jul 6, 2013 at 21:12

15 This is a great answer but alas the current era has forgotten what "information hiding" is or what it is for. They never read about immutability and in their quest for most-agile, never drew that state-transition-
[illegible]

diagram that defined what the legal states of an object were and thus what were not. – Darrell Teague Nov 6, 2013 at 14:58

114

Lots of people talk about the advantages of getters and setters but I want to play devil's advocate. Right now I'm debugging a very large program where the programmers decided to make everything getters and setters. That might seem nice, but its a reverse-engineering nightmare.

Say you're looking through hundreds of lines of code and you come across this:

```
person.name = "Joe";
```

It's a beautifully simply piece of code until you realize its a setter. Now, you follow that setter and find that it also sets person.firstName, person.lastName, person.isHuman, person.hasReallyCommonFirstName, and calls person.update(), which sends a query out to the database, etc. Oh, that's where your memory leak was occurring.

Understanding a local piece of code at first glance is an important property of good readability that getters and setters tend to break. That is why I try to avoid them when I can, and minimize what they do when I use them.

Share Edit Follow

answered Oct 14, 2009 at 21:00
user Kai
9,306 6 44 61

44 This is argument against syntactic sugar, not against setters in general. – Phil Oct 29, 2013 at 19:02

@Phil I'd disagree. One can just as easily write a `public void setName(String name)` (in Java) that does the same exact things. Or worse, a `public void setFirstName(String name)` that does all those things. – tedtanner May 11 at 13:05

1 @tedtanner and when you do, it's no longer a mystery that a method is being invoked, since it's not disguised as field access. – Phil May 12 at 20:40

2 @Phil I am with you on that one. I don't believe in hiding method calls. I just want to point out that Kai's argument is less about syntactic sugar and more about how setters (including ones that look like method calls) shouldn't have side effects like changing other data members in the object or querying the database. – tedtanner May 13 at 5:15

@tedtanner I think we share opinions as to what setters and getters should do, and as to whether they should be hidden like this or not, but the fact is when they are hidden, it's always syntactic sugar - it's always a method call, on the machine. Back in 2009 when the answer was written, and in 2013 when I wrote my comment, one used annotations and either codegen or pointcut type solutions to do this, and it was all very non-standard. Today in 2022, I haven't touched Java for years 😂 – Phil May 13 at 9:22

In a pure object-oriented world getters and setters is a **terrible anti-pattern**. Read this article:

72

Getters/Setters. Evil. Period. In a nutshell, they encourage programmers to think about objects as of data structures, and this type of thinking is pure procedural (like in COBOL or C). In an object-oriented language there are no data structures, but only objects that expose behavior (not attributes/properties!)

You may find more about them in Section 3.5 of Elegant Objects (my book about object-oriented programming).

Share Edit Follow edited Aug 21, 2019 at 7:01 answered Sep 23, 2014 at 16:39

user yegor256 **98.4k** 115 426 574

15 Interesting viewpoint. But in most programming contexts what we need is data structures. Taking the linked article's "Dog" example. Yes, you can't change a real-world dog's weight by setting an attribute ... but a `new Dog()` is not a dog. It is object that holds information about a dog. And for that usage, it is natural to be able to correct an incorrectly recorded weight. – Stephen C Jul 1, 2016 at 7:29

4 Well, I put it to you that *most* useful programs don't need to model / simulate real world objects. IMO, this is not really about programming languages at all. It is about what we write programs for. – Stephen C Jul 2, 2016 at 0:34

5 Real world or not, yegor is completely right. If what you have is truly a "Struct" and you don't need to write any code that references it by name, put it in a hashtable or other data structure. If you do need to write code for it then put it as a member of a class and put the code that manipulates that variable in the same class and omit the setter & getter. PS. although I mostly share yegor's viewpoint, I have come to believe that annotated beans without code are somewhat useful data structures--also getters are sometimes necessary, setters shouldn't ever exist. – Bill K May 1, 2017 at 16:51

1 I got carried away--this entire answer, although correct and relevant, doesn't address the question directly. Perhaps it should say "Both setters/getters AND public variables are wrong"... To be absolutely specific, Setters and writable public variables should never ever be used whereas getters are pretty much the same as public final variables and are occasionally a necessary evil but neither is much better than the other. – Bill K May 1, 2017 at 17:03

1 I have never understood getters/setters. From the first Java101 class I took at the college, they seemed like they encourage building lacking abstractions and confused me. But there is this case then: your object may possibly have so many different behavior. Then it is possible that you might not hope to implement them all. You make setters/getters and conveniently tell your users that if you need a behavior that I missed, inherit and implement them yourself. I gave you accessors after all. – meguli Jun 5, 2017 at 14:24

58

There are many reasons. My favorite one is when you need to change the behavior or regulate what you can set on a variable. For instance, lets say you had a setSpeed(int speed) method. But you want that you can only set a maximum speed of 100. You would do something like:

```
public void setSpeed(int speed) {
  if ( speed > 100 ) {
    this.speed = 100;
  } else {
    this.speed = speed;
    }
  }
```

Now what if EVERYWHERE in your code you were using the public field and then you realized you need the above requirement? Have fun hunting down every usage of the public field instead of just modifying your setter.

My 2 cents :)

Share Edit Follow

answered Oct 14, 2009 at 18:27
user Peter D
4,821 2 28 30

73 Hunting down every usage of the public field shouldn't be that hard. Make it private and let the compiler find them. – Nathan Fellman Oct 14, 2009 at 18:57

12 that's true of course, but why make it any harder than it was designed to be. The get/set approach is still the better answer. – Hardryv Oct 14, 2009 at 19:05

33 @GraemePerrow having to change them all is an advantage, not a problem :(What if you had code that assumed speed could be higher than 100 (because, you know, before you broke the contract, it could!) (`while(speed < 200) { do_something(); accelerate(); }`) – R. Martinho Fernandes Oct 30, 2013 at 11:12

45 It is a VERY bad example! Someone should call: `myCar.setSpeed(157);` and after few lines `speed = myCar.getSpeed();` And now... I wish you happy debugging while trying to understand why `speed==100` when it should be `157` – Piotr Aleksander Chmielowski Feb 23, 2016 at 12:24

42

One advantage of accessors and mutators is that you can perform validation.

For example, if `foo` was public, I could easily set it to `null` and then someone else could try to call a method on the object. But it's not there anymore! With a `setFoo` method, I could ensure that `foo` was never set to `null`.

Accessors and mutators also allow for encapsulation - if you aren't supposed to see the value once its set (perhaps it's set in the constructor and then used by methods, but never supposed to be changed), it will never been seen by anyone. But if you can allow other classes to see or change it, you can provide the proper accessor and/or mutator.

Share Edit Follow

answered Oct 14, 2009 at 18:25
Thomas Owens
112k 96 306 430

30

Depends on your language. You've tagged this "object-oriented" rather than "Java", so I'd like to point out that ChssPly76's answer is language-dependent. In Python, for instance, there is no reason to use getters and setters. If you need to change the behavior, you can use a property, which wraps a getter and setter around basic attribute access. Something like this:

```
class Simple(object):
  def _get_value(self):
      return self._value -1

  def _set_value(self, new_value):
      self._value = new_value + 1

  def _del_value(self):
      self.old_values.append(self._value)
      del self._value

  value = property(_get_value, _set_value, _del_value)
```

Share Edit Follow

edited Jul 4, 2020 at 20:31
pppery
3,597 20 30 43

answered Oct 14, 2009 at 18:32
user jcdyer
18.1k 5 41 48

2 Yes, I've said as much in a comment below my answer. Java is not the only language to use getters / setters as a crutch just like Python is not the only language able to define properties. The main point, however, still remains - "property" is not the same "public field". – ChssPly76 Oct 14, 2009 at 18:40

1 @jcd - not at all. You're defining your "interface" (public API would be a better term here) by exposing your public fields. Once that's done, there's no going back. Properties are NOT fields because they provide you with a mechanism to intercept attempts to access fields (by routing them to methods if those are defined); that is, however, nothing more than syntax sugar over getter / setter methods. It's extremely convenient but it doesn't alter the underlying paradigm - exposing fields with no control over access to them violates the principle of encapsulation. – ChssPly76 Oct 15, 2009 at 21:09

17 @ChssPly76—I disagree. I have just as much control as if they were properties, because I can make them properties whenever I need to. There is no difference between a property that uses boilerplate getters and setters, and a raw attribute, except that the raw attribute is faster, because it utilizes the underlying language, rather than calling methods. Functionally, they are identical. The only way encapsulation could be violated is if you think parentheses (`obj.set_attr('foo')`) are inherently superior to equals signs (`obj.attr = 'foo'`). Public access is public access. – jcdyer Oct 16, 2009 at 15:27

1 @jcdyer as much control yes, but not as much readability, others often wrongly assume that `obj.attr = 'foo'` just sets the variable without anything else happening – Timo Huovinen Aug 10, 2013 at 9:49

9 @TimoHuovinen How is that any different than a user in Java assuming that `obj.setAttr('foo')` "just sets the variable without anything else happening"? If it's a public methods, then it's a public method. If you use it to achieve some side-effect, and it is public, then you had better be able to count on everything working *as if only that intended side effect happened* (with *all* other implementation details and other side effects, resource use, whatever, hidden from the user's concerns). This is absolutely no different with Python. Python's syntax to achieve the effect is just simpler. – ely Oct 17, 2014 at 14:00

29

Thanks, that really clarified my thinking. Now here is (almost) 10 (almost) good reasons NOT to use getters and setters:

1. When you realize you need to do more than just set and get the value, you can just make the field private, which will instantly tell you where you've directly accessed it.
2. Any validation you perform in there can only be context free, which validation rarely is in practice.
3. You can change the value being set - this is an absolute nightmare when the caller passes you a value that they [shock horror] want you to store AS IS.
4. You can hide the internal representation - fantastic, so you're making sure that all these operations are symmetrical right?
5. You've insulated your public interface from changes under the sheets - if you were designing an interface and weren't sure whether direct access to something was OK, then you should have kept designing.
6. Some libraries expect this, but not many - reflection, serialization, mock objects all work just fine with public fields.
7. Inheriting this class, you can override default functionality - in other words you can REALLY confuse callers by not only hiding the implementation but making it inconsistent.

The last three I'm just leaving (N/A or D/C)...

Share Edit Follow

edited Aug 23, 2012 at 21:25
user StackedCrooked **33.8k** 41 145 275

answered Mar 27, 2012 at 3:53
user595447

14 I think the crucial argument is that, "*if you were designing an interface and weren't sure whether direct access to something was OK, then you should have kept designing.*" That is the most important problem with getters/setters: They reduce a class to a mere container of (more or less) public fields. In *real* OOP, however, an object is more than a container of data fields. It encapsulates state and algorithms to manipulate that state. What's crucial about this statement is that the state is supposed to be *encapsulated* and only to be manipulated by the algorithms provided by the object. – sbi Aug 24, 2012 at 9:29

27

Well i just want to add that even if sometimes they are necessary for the encapsulation and security of your variables/objects, if we want to code a real Object Oriented Program, then we need to **STOP OVERUSING THE ACCESSORS**, cause sometimes we depend a lot on them when is not really necessary and that makes almost the same as if we put the variables public.

Share Edit Follow

answered Aug 22, 2012 at 14:47
user Jorge Aguilar **3,422** 29 34

27

EDIT: I answered this question because there are a bunch of people learning programming asking this, and most of the answers are very technically competent, but they're not as easy to understand if you're a newbie. We were all newbies, so I thought I'd try my hand at a more newbie friendly answer.

The two main ones are polymorphism, and validation. Even if it's just a stupid data structure.

Let's say we have this simple class:

```
public class Bottle {
  public int amountOfWaterMl;
  public int capacityMl;
}
```

A very simple class that holds how much liquid is in it, and what its capacity is (in milliliters).

What happens when I do:

```
Bottle bot = new Bottle();
bot.amountOfWaterMl = 1500;
bot.capacityMl = 1000;
```

Well, you wouldn't expect that to work, right? You want there to be some kind of sanity check. And worse, what if I never specified the maximum capacity? Oh dear, we have a problem.

But there's another problem too. What if bottles were just one type of container? What if we had several containers, all with capacities and amounts of liquid filled? If we could just make an interface, we could let the rest of our program accept that interface, and bottles, jerrycans and all sorts of stuff would just work interchangably. Wouldn't that be better? Since interfaces demand methods, this is also a good thing.

We'd end up with something like:

```
public interface LiquidContainer {
  public int getAmountMl();
  public void setAmountMl(int amountMl);
  public int getCapacityMl();
}
```

Great! And now we just change Bottle to this:

```
public class Bottle extends LiquidContainer {
  private int capacityMl;
  private int amountFilledMl;

  public Bottle(int capacityMl, int amountFilledMl) {
    this.capacityMl = capacityMl;
    this.amountFilledMl = amountFilledMl;
    checkNotOverFlow();
  }
```

```
    public int getAmountMl() {
      return amountFilledMl;
    }

    public void setAmountMl(int amountMl) {
       this.amountFilled = amountMl;
       checkNotOverFlow();
    }
    public int getCapacityMl() {
      return capacityMl;
    }

    private void checkNotOverFlow() {
      if(amountOfWaterMl > capacityMl) {
        throw new BottleOverflowException();
      }
  }
```

I'll leave the definition of the BottleOverflowException as an exercise to the reader.

Now notice how much more robust this is. We can deal with any type of container in our code now by accepting LiquidContainer instead of Bottle. And how these bottles deal with this sort of stuff can all differ. You can have bottles that write their state to disk when it changes, or bottles that save on SQL databases or GNU knows what else.

And all these can have different ways to handle various whoopsies. The Bottle just checks and if it's overflowing it throws a RuntimeException. But that might be the wrong thing to do. (There is a useful discussion to be had about error handling, but I'm keeping it very simple here on purpose. People in comments will likely point out the flaws of this simplistic approach. ;))

And yes, it seems like we go from a very simple idea to getting much better answers quickly.

Please note also that you can't change the capacity of a bottle. It's now set in stone. You could do this with an int by declaring it final. But if this was a list, you could empty it, add new things to it, and so on. You can't limit the access to touching the innards.

There's also the third thing that not everyone has addressed: getters and setters use method calls. That means that they look like normal methods everywhere else does. Instead of having weird specific syntax for DTOs and stuff, you have the same thing everywhere.

Share Edit Follow

edited Jun 14, 2019 at 13:07

answered Nov 29, 2015 at 14:37

user Haakon Løtveit
958 9 18

24

I know it's a bit late, but I think there are some people who are interested in performance.

I've done a little performance test. I wrote a class "NumberHolder" which, well, holds an Integer. You can either read that Integer by using the getter method `anInstance.getNumber()` or by directly accessing the number by using `anInstance.number`. My programm reads the number 1,000,000,000 times, via both ways. That process is repeated five times and the time is printed. I've got the following result:

```
Time 1: 953ms, Time 2: 741ms
Time 1: 655ms, Time 2: 743ms
Time 1: 656ms, Time 2: 634ms
Time 1: 637ms, Time 2: 629ms
Time 1: 633ms, Time 2: 625ms
```

(Time 1 is the direct way, Time 2 is the getter)

You see, the getter is (almost) always a bit faster. Then I tried with different numbers of cycles. Instead of 1 million, I used 10 million and 0.1 million. The results:

10 million cycles:

```
Time 1: 6382ms, Time 2: 6351ms
Time 1: 6363ms, Time 2: 6351ms
Time 1: 6350ms, Time 2: 6363ms
Time 1: 6353ms, Time 2: 6357ms
Time 1: 6348ms, Time 2: 6354ms
```

With 10 million cycles, the times are almost the same. Here are 100 thousand (0.1 million) cycles:

```
Time 1: 77ms, Time 2: 73ms
Time 1: 94ms, Time 2: 65ms
Time 1: 67ms, Time 2: 63ms
Time 1: 65ms, Time 2: 65ms
Time 1: 66ms, Time 2: 63ms
```

Also with different amounts of cycles, the getter is a little bit faster than the regular way. I hope this helped you.

Share Edit Follow

edited May 9, 2018 at 9:09
Nick stands with Ukraine
6,465 19 41 49

answered Aug 30, 2016 at 18:59
user
kangalioo
593 9 13

4 There's a "noticable" overhead having a function call to access the memory instead of simply loading an object's address and adding an offset to access the members. Chances are the VM flat-optimized your getter anyway. Regardless, the mentioned overhead isn't worth losing all the benefits of getters/setters.
– Alex Sep 6, 2016 at 17:28

19

We use getters and setters:

- for reusability
- to perform validation in later stages of programming

Getter and setter methods are public interfaces to access private class members.

Encapsulation mantra

The encapsulation mantra is to make fields private and methods public.

> **Getter Methods:** *We can get access to private variables.*
>
> **Setter Methods:** *We can modify private fields.*

Even though the getter and setter methods do not add new functionality, we can change our mind come back later to make that method

- better;
- safer; and
- faster.

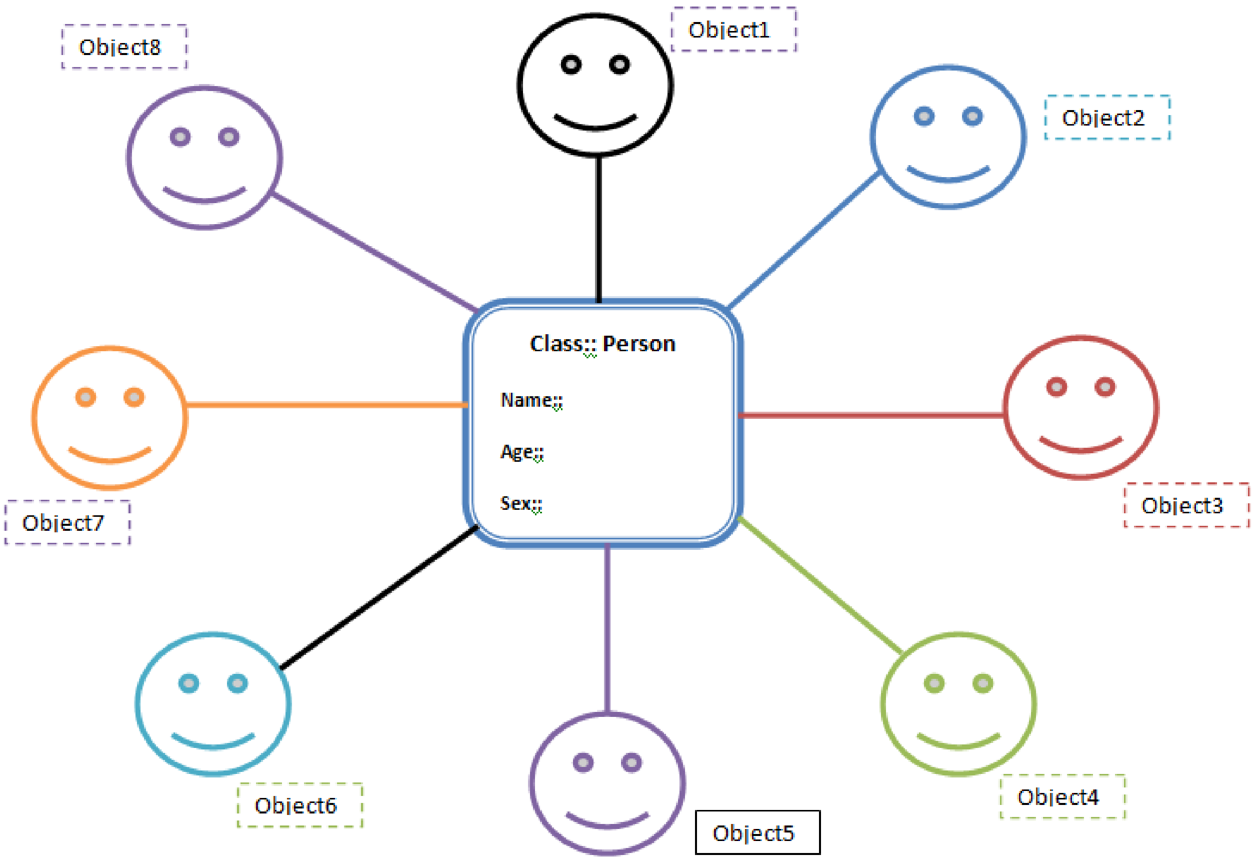
Anywhere a value can be used, a method that returns that value can be added. Instead of:

```
int x = 1000 - 500
```

use

```
int x = 1000 - class_name.getValue();
```

In layman's terms



Suppose we need to store the details of this `Person`. This `Person` has the fields `name`, `age` and `sex`. Doing this involves creating methods for `name`, `age` and `sex`. Now if we need create another person, it becomes necessary to create the methods for `name`, `age`, `sex` all over again.

Instead of doing this, we can create a bean `class(Person)` with getter and setter methods. So tomorrow we can just create objects of this Bean `class(Person class)` whenever we need to add a new person (see the figure). Thus we are reusing the fields and methods of bean class, which is much better.

Share Edit Follow

edited Dec 19, 2013 at 5:22
Qantas 94 Heavy
15.4k 31 63 82

answered Sep 2, 2013 at 6:51
Devrath
40.5k 51 182 271

18

I spent quite a while thinking this over for the Java case, and I believe the real reasons are:

1. **Code to the interface, not the implementation**
2. **Interfaces only specify methods, not fields**

In other words, the only way you can specify a field in an interface is by providing a method for writing a new value and a method for reading the current value.

Those methods are the infamous getter and setter....

Share Edit Follow

answered Nov 7, 2009 at 21:55

user Thorbjørn Ravn Andersen
72.4k 32 185 337

2 Okay, second question; in the case where it's a project where you're not exporting source to anyone, and you have full control of the source... are you gaining anything with getters and setters? – Dean J Nov 9, 2009 at 15:22

2 In any non-trivial Java project you need to code to interfaces in order to make things manageable and testable (think mockups and proxy objects). If you use interfaces you need getters and setters. – Thorbjørn Ravn Andersen Nov 9, 2009 at 18:13

17

Don't use getters setters unless needed for your current delivery I.e. Don't think too much about what would happen in the future, if any thing to be changed its a change request in most of the production applications, systems.

Think simple, easy, add complexity when needed.

I would not take advantage of ignorance of business owners of deep technical know how just because I think it's correct or I like the approach.

I have massive system written without getters setters only with access modifiers and some methods to validate n perform biz logic. If you absolutely needed the. Use anything.

Share Edit Follow

answered Jul 14, 2012 at 16:20

user Mohamed
179 1 2

16

It can be useful for lazy-loading. Say the object in question is stored in a database, and you don't want to go get it unless you need it. If the object is retrieved by a getter, then the internal object can be null until somebody asks for it, then you can go get it on the first call to the getter.

I had a base page class in a project that was handed to me that was loading some data from a couple different web service calls, but the data in those web service calls wasn't always used in all child pages. Web services, for all of the benefits, pioneer new definitions of "slow", so you don't want to make a web service call if you don't have to.

I moved from public fields to getters, and now the getters check the cache, and if it's not there call the web service. So with a little wrapping, a lot of web service calls were prevented.

So the getter saves me from trying to figure out, on each child page, what I will need. If I need it, I call the getter, and it goes to find it for me if I don't already have it.

```
protected YourType _yourName = null;
public YourType YourName{
  get
  {
    if (_yourName == null)
    {
      _yourName = new YourType();
      return _yourName;
    }
  }
}
```

Share Edit Follow

edited Sep 19, 2014 at 18:20

answered Oct 14, 2009 at 19:08

quillbreaker
6,041 3 28 47

15

One aspect I missed in the answers so far, the access specification:

- for members you have only one access specification for both setting and getting
- for setters and getters you can fine tune it and define it separately

Share Edit Follow

answered Oct 14, 2009 at 18:38

user jdehaan
19.5k 6 56 95

12

In languages which don't support "properties" (C++, Java) or require recompilation of clients when changing fields to properties (C#), using get/set methods is easier to modify. For example, adding validation logic to a setFoo method will not require changing the public interface of a class.

In languages which support "real" properties (Python, Ruby, maybe Smalltalk?) there is no point to get/set methods.

Share Edit Follow

edited Oct 14, 2009 at 19:13

answered Oct 14, 2009 at 18:25

user John Millikin
192k 39 210 222

1 Re: C#. If you add functionality to a get/set wouldn't that require recompilation anyway? – steamer25 Oct 14, 2009 at 19:11

5 Adding validation logic to a setFoo method will not require changing the interface of a class *at the language level*, but *it does change the actual interface*, aka contract, because it changes the preconditions. Why would one want the compiler to not treat that as a breaking change *when it is*?
– R. Martinho Fernandes Sep 26, 2012 at 5:57

3 Requiring recompilation, as mentioned in the answer, is one way the compiler can make you aware of a possible breaking change. And almost everything I write is effectively "a library for others", because I don't

work alone. I write code that has interfaces that other people in the project will use. What's the difference? Hell, even if I will be the user of those interfaces, why should I hold my code to lower quality standards? I don't like working with troublesome interfaces, even if I'm the one writing them. – R. Martinho Fernandes Oct 30, 2013 at 11:02

7

One of the basic principals of OO design: **Encapsulation!**

It gives you many benefits, one of which being that you can change the implementation of the getter/setter behind the scenes but any consumer of that value will continue to work as long as the data type remains the same.

Share Edit Follow

answered Oct 14, 2009 at 18:25

user Justin Niessner
237k 39 405 532

23 The encapsulation getters and setters offer are laughably thin. See here. – sbi Aug 23, 2012 at 21:14

5 Why should things keep on compiling if contracts change? – R. Martinho Fernandes Oct 30, 2013 at 11:06

6

You should use getters and setters when:

- You're dealing with something that is conceptually an attribute, but:
  - Your language doesn't have properties (or some similar mechanism, like Tcl's variable traces), or
  - Your language's property support isn't sufficient for this use case, or
  - Your language's (or sometimes your framework's) idiomatic conventions encourage getters or setters for this use case.

So this is very rarely a general OO question; it's a language-specific question, with different answers for different languages (and different use cases).

From an OO theory point of view, getters and setters are useless. The interface of your class is what it does, not what its state is. (If not, you've written the wrong class.) In very simple cases, where what a class does is just, e.g., represent a point in rectangular coordinates,* the attributes are part of the interface; getters and setters just cloud that. But in anything but very simple cases, neither the attributes nor getters and setters are part of the interface.

Put another way: If you believe that consumers of your class shouldn't even know that you have a `spam` attribute, much less be able to change it willy-nilly, then giving them a `set_spam` method is the last thing you want to do.

* Even for that simple class, you may not necessarily want to allow setting the `x` and `y` values. If this is really a class, shouldn't it have methods like `translate`, `rotate`, etc.? If it's only a class because your language doesn't have records/structs/named tuples, then this isn't really a question of OO…

But nobody is ever doing general OO design. They're doing design, and implementation, in a specific language. And in some languages, getters and setters are far from useless.

If your language doesn't have properties, then the only way to represent something that's conceptually an attribute, but is actually computed, or validated, etc., is through getters and setters.

Even if your language does have properties, there may be cases where they're insufficient or inappropriate. For example, if you want to allow subclasses to control the semantics of an attribute, in languages without dynamic access, a subclass can't substitute a computed property for an attribute.

As for the "what if I want to change my implementation later?" question (which is repeated multiple times in different wording in both the OP's question and the accepted answer): If it really is a pure implementation change, and you started with an attribute, you can change it to a property without affecting the interface. Unless, of course, your language doesn't support that. So this is really just the same case again.

Also, it's important to follow the idioms of the language (or framework) you're using. If you write beautiful Ruby-style code in C#, any experienced C# developer other than you is going to have trouble reading it, and that's bad. Some languages have stronger cultures around their conventions than others.—and it may not be a coincidence that Java and Python, which are on opposite ends of the spectrum for how idiomatic getters are, happen to have two of the strongest cultures.

Beyond human readers, there will be libraries and tools that expect you to follow the conventions, and make your life harder if you don't. Hooking Interface Builder widgets to anything but ObjC properties, or using certain Java mocking libraries without getters, is just making your life more difficult. If the tools are important to you, don't fight them.

Share Edit Follow

answered Aug 19, 2014 at 4:47

user abarnert
339k 43 566 644

5

From a object orientation design standpoint both alternatives can be damaging to the maintenance of the code by weakening the encapsulation of the classes. For a discussion you can look into this excellent article: http://typicalprogrammer.com/?p=23

Share Edit Follow

edited Apr 30, 2012 at 19:08

answered Apr 30, 2012 at 18:57

user andrers52
524 6 9

5

Code *evolves*. `private` is great for when *you need data member protection*. Eventually all classes should be sort of "miniprograms" that have a well-defined interface *that you can't just screw with the internals of*.

That said, *software development* isn't about setting down that final version of the class as if you're pressing some cast iron statue on the first try. While you're working with it, code is more like clay. **It evolves** as you develop it and learn more about the problem domain you are solving. During development classes may interact with each other than they should (dependency you plan to factor out), merge together, or split apart. So I think the debate boils down to people not wanting to religiously write

```
int getVar() const { return var ; }
```

So you have:

```
doSomething( obj->getVar() ) ;
```

Instead of

```
doSomething( obj->var ) ;
```

Not only is `getVar()` visually noisy, it gives this illusion that `gettingVar()` is somehow a more complex process than it really is. How you (as the class writer) regard the sanctity of `var` is particularly confusing to a user of your class if it has a passthru setter -- then it looks like you're putting up these gates to "protect" something you insist is valuable, (the sanctity of `var`) but yet even you concede `var`'s protection isn't worth much by the ability for anyone to just come in and `set` `var` to whatever value they want, without you even peeking at what they are doing.

So I program as follows (assuming an "agile" type approach -- ie when I write code not knowing *exactly* what it will be doing/don't have time or experience to plan an elaborate waterfall style interface set):

1) Start with all public members for basic objects with data and behavior. This is why in all my C++ "example" code you'll notice me using `struct` instead of `class` everywhere.

2) When an object's internal behavior for a data member becomes complex enough, (for example, it likes to keep an internal `std::list` in some kind of order), accessor type functions are written. Because I'm programming by myself, I don't always set the member `private` right away, but somewhere down the evolution of the class the member will be "promoted" to either `protected` or `private`.

3) Classes that are fully fleshed out and have strict rules about their internals (ie *they* know exactly what they are doing, and you are not to "fuck" (technical term) with its internals) are given the `class` designation, default private members, and only a select few members are allowed to be `public`.

I find this approach allows me to avoid sitting there and religiously writing getter/setters when a lot of data members get migrated out, shifted around, etc. during the early stages of a class's evolution.

Share Edit Follow

edited May 5, 2013 at 2:13

answered May 5, 2013 at 2:07

bobobobo

62.2k 58 250 349

1 "... a well-defined interface that you can't just screw with the internals of" and validation in setters.
– Agi Hammerthief Dec 18, 2014 at 20:45

5

There is a good reason to consider using accessors is there is no property inheritance. See next example:

```
public class TestPropertyOverride {
    public static class A {
        public int i = 0;

        public void add() {
            i++;
        }

        public int getI() {
            return i;
        }
    }

    public static class B extends A {
        public int i = 2;

        @Override
        public void add() {
            i = i + 2;
        }

        @Override
        public int getI() {
            return i;
        }
    }

    public static void main(String[] args) {
        A a = new B();
        System.out.println(a.i);
        a.add();
        System.out.println(a.i);
        System.out.println(a.getI());
    }
}
```

Output:

```
0
0
4
```

Share Edit Follow

answered May 13, 2014 at 9:47

user GeZo
86 2 4

5

Getters and **setters** are used to implement two of the fundamental aspects of Object Oriented Programming which are:

1. Abstraction
2. Encapsulation

Suppose we have an Employee class:

```
package com.highmark.productConfig.types;

public class Employee {

    private String firstName;
    private String middleName;
    private String lastName;

    public String getFirstName() {
      return firstName;
    }
    public void setFirstName(String firstName) {
       this.firstName = firstName;
    }
    public String getMiddleName() {
        return middleName;
    }
    public void setMiddleName(String middleName) {
         this.middleName = middleName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getFullName(){
        return this.getFirstName() + this.getMiddleName() +  this.getLastName();
    }
 }
```

Here the implementation details of Full Name is hidden from the user and is not accessible directly to the user, unlike a public attribute.

Share Edit Follow

edited Jul 5, 2016 at 22:32

answered Dec 10, 2015 at 20:44

Pritam Banerjee
16.9k 10 83 102

3 To me having tons of getters and setters that do nothing unique is useless. getFullName is a exception because it does something else. Having just the three variables public then keeping getFullName will make the program easier to read but still have that fullname thing hidden. Generally I'm completely fine with getters and setters if a. they do something unique and/or b. you only have one, yeah you could have a public final and all that but nah – FacelessTiger Dec 14, 2016 at 1:06

2 The benefit with this is that you can change the internals of the class, without changing the interface. Say, instead of three properties, you had one - an array of strings. If you've been using getters and setters, you can make that change, and then update the getter/setters to know that names[0] is first name, names[1] is middle, etc. But if you just used public properties, you would also have to change every class accessed Employee, because the firstName property they've been using no longer exists. – Andrew Hows Dec 8, 2017 at 0:42

3 @AndrewHows from what I've seen in real life, when people change the internals of the class they also change the interface and do a big refactoring of all the code – Heetola May 6, 2020 at 7:41

5

If you don't require any validations and not even need to maintain state i.e. one property depends on another so we need to maintain the state when one is change. You can keep it simple by making field public and not using getter and setters.

I think OOPs complicates things as the program grows it becomes nightmare for developer to scale.

A simple example; we generate c++ headers from xml. The header contains simple field which does not require any validations. But still as in OOPS accessor are fashion we generates them as following.

```
const Filed& getfield() const
Field& getField()
void setfield(const Field& field){...}
```

which is very verbose and is not required. a simple

```
struct
{
   Field field;
};
```

is enough and readable. Functional programming don't have the concept of data hiding they even don't require it as they do not mutate the data.

Share Edit Follow

answered Jun 8, 2021 at 5:38

user user982042
199 2 6

4

Getter and setter methods are accessor methods, meaning that they are generally a public interface to change private class members. You use getter and setter methods to define a property. You access getter and setter methods as properties outside the class, even though you define them within the class as methods. Those properties outside the class can have a different name from the property name in the class.

There are some advantages to using getter and setter methods, such as the ability to let you create members with sophisticated functionality that you can access like properties. They also let you create read-only and write-only properties.

Even though getter and setter methods are useful, you should be careful not to overuse them because, among other issues, they can make code maintenance more difficult in certain situations. Also, they provide access to your class implementation, like public members. OOP practice discourages direct access to properties within a class.

When you write classes, you are always encouraged to make as many as possible of your instance variables private and add getter and setter methods accordingly. This is because there are several times when you may not want to let users change certain variables within your classes. For example, if you have a private static method that tracks the number of instances created for a specific class, you don't want a user to modify that counter using code. Only the constructor statement should increment that variable whenever it's called. In this situation, you might create a private instance variable and allow a getter method only for the counter variable, which means users are able to retrieve the current value only by using the getter method, and they won't be able to set new values using the setter method. Creating a getter without a setter is a simple way of making certain variables in your class read-only.

Share Edit Follow

answered Aug 30, 2012 at 13:02

Sumit Singh
24.2k 8 75 100

4

There is a difference between DataStructure and Object.

Datastructure should expose its innards and not behavior.

An Object should not expose its innards but it should expose its behavior, which is also known as the Law of Demeter

Mostly DTOs are considered more of a datastructure and not Object. They should only expose their data and not behavior. Having Setter/Getter in DataStructure will expose behavior instead of data inside it. This further increases the chance of violation of **Law of Demeter**.

Uncle Bob in his book Clean code explained the Law of Demeter.

> There is a well-known heuristic called the Law of Demeter that says a module should not know about the innards of the objects it manipulates. As we saw in the last section, objects hide their data and expose operations. This means that an object should not expose its internal structure through accessors because to do so is to expose, rather than to hide, its internal structure.
>
> More precisely, the Law of Demeter says that a method f of a class C should only call the methods of these:
>
> - C
> - An object created by f
> - An object passed as an argument to f
> - An object held in an instance variable of C
>
> The method should not invoke methods on objects that are returned by any of the allowed functions. In other words, talk to friends, not to strangers.

So according this, example of LoD violation is:

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

Here, the function should call the method of its immediate friend which is ctxt here, It should not call the method of its immediate friend's friend. but this rule doesn't apply to data structure. so here if ctxt, option, scratchDir are datastructure then why to wrap their internal data with some behavior and doing a violation of LoD.

Instead, we can do something like this.

```
final String outputDir = ctxt.options.scratchDir.absolutePath;
```

This fulfills our needs and doesn't even violate LoD.

Inspired by Clean Code by Robert C. Martin(Uncle Bob)

Share Edit Follow

answered Mar 30, 2021 at 14:25
user aditya lath
370 4 6

3

Additionally, this is to "future-proof" your class. In particular, changing from a field to a property is an ABI break, so if you do later decide that you need more logic than just "set/get the field", then you need to break ABI, which of course creates problems for anything else already compiled against your class.

Share Edit Follow

answered Oct 14, 2009 at 18:25

user Pete
11.1k 4 40 53

2 I suppose that changing the behaviour of the getter or setter isn't a breaking change then. </sarcasm>
– R. Martinho Fernandes Sep 26, 2012 at 5:54

3

One other use (in languages that support properties) is that setters and getters can imply that an operation is non-trivial. Typically, you want to avoid doing anything that's computationally expensive in a property.

Share Edit Follow

answered Oct 14, 2009 at 18:27

user Jason Baker
184k 131 363 509

3

One relatively modern advantage of getters/setters is that is makes it easier to browse code in tagged (indexed) code editors. E.g. If you want to see who sets a member, you can open the call hierarchy of the setter.

On the other hand, if the member is public, the tools don't make it possible to filter read/write access to the member. So you have to trudge though all uses of the member.

Share Edit Follow

answered Jan 3, 2017 at 16:10

user Rakesh Singh
51 2

---

1 2 Next