Private vs. Public members in practice (how important is encapsulation?) [closed]

Asked 16 years, 3 months ago Modified 8 years, 8 months ago

Viewed 5k times

22

Closed. This question is opinion-based. It is not currently accepting answers.

Want to improve this question? Update the question so it can be answered with facts and citations by editing this post.

Closed 6 years ago.

Improve this question

One of the biggest advantages of object-oriented programming is encapsulation, and one of the "truths" we've (or, at least, I've) been taught is that members should always be made private and made available via accessor and mutator methods, thus ensuring the ability to verify and validate the changes.

I'm curious, though, how important this really is in practice. In particular, if you've got a more complicated member (such as a collection), it can be very tempting to

just make it public rather than make a bunch of methods to get the collection's keys, add/remove items from the collection, etc.

Do you follow the rule in general? Does your answer change depending on whether it's code written for yourself vs. to be used by others? Are there more subtle reasons I'm missing for this obfuscation?

language-agnostic **encapsulation**

Share

Improve this question

Follow

edited May 9, 2010 at 10:55

o0'.

11.8k ● 19 ● 62 ● 88

asked Sep 19, 2008 at 4:44

Asmor

5,181 ● 6 ● 34 ● 42

possible duplicate of Why use getters and setters? – nawfal Jun 4, 2013 at 5:48

Note that most of the arguments for using accessors still work even if your accessors exposes a collection to the caller. If you're worried about encapsulation, simply clone the collection before passing it to the caller, or before overwriting the old collection value. – jpaugh Feb 2, 2018 at 21:24

22 Answers

Sorted by: Highest score (default)

16

It depends. This is one of those issues that must be decided pragmatically.

Suppose I had a class for representing a point. I could have getters and setters for the X and Y coordinates, or I could just make them both public and allow free read/write access to the data. In my opinion, this is OK because the class is acting like a glorified struct - a data collection with maybe some useful functions attached.

However, there are plenty of circumstances where you do not want to provide full access to your internal data and rely on the methods provided by the class to interact with the object. An example would be an HTTP request and response. In this case it's a bad idea to allow anybody to send anything over the wire - it must be processed and formatted by the class methods. In this case, the class is conceived of as an actual object and not a simple data store.

It really comes down to whether or not verbs (methods) drive the structure or if the data does.

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:52

Kyle Cronin

79k ● 45 ● 151 ● 167

I just wanted to say that I found your last point (whether methods or data drive the structure) particularly insightful. Thanks! – Asmor Sep 19, 2008 at 5:02

If you later wanted to enhance your class to take some action when X or Y changes, it is much easier with the getters and setters. – Trent Sep 19, 2008 at 5:13

1 That's the distinction though - if I store a point I don't expect it to do anything but store information. It's all a matter of what the class is going to be used for. – Kyle Cronin Sep 19, 2008 at 5:18

... and what the class might be used for in the future. Often worth thinking through possible future changes. – MarkJ Feb 5, 2009 at 23:24

1 What if you decide there are minimum and maximum legal values for X and Y? Then you might well want to enforce it in your non-existent getter and setter for X and Y. – MarkJ Feb 28, 2009 at 19:36

16

As someone having to maintain several-year-old code worked on by many people in the past, it's very clear to me that if a member attribute is made public, it is eventually abused. I've even heard people disagreeing with the idea of accessors and mutators, as that's still not really living up to the purpose of encapsulation, which is "hiding the inner workings of a class". It's obviously a controversial topic, but my opinion would be "make every member variable private, think primarily about what the class has got to *do* (methods) rather than *how* you're going to let people change internal variables".

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:49

Smashery

59.6k ● 31 ● 100 ● 129

8

Yes, encapsulation matters. Exposing the underlying implementation does (at least) two things wrong:

1. Mixes up responsibilities. Callers shouldn't need or want to understand the underlying implementation. They should just want the class to do its job. By exposing the underlying implementation, you're class isn't doing its job. Instead, it's just pushing the responsibility onto the caller.
2. Ties you to the underlying implementation. Once you expose the underlying implementation, you're tied to it. If you tell callers, e.g., there's a collection

underneath, you cannot easily swap the collection for a new implementation.

These (and other) problems apply regardless of whether you give direct access to the underlying implementation or just duplicate all the underlying methods. You should be exposing the necessary implementation, and nothing more. Keeping the implementation private makes the overall system more maintainable.

Share Improve this answer

Follow

answered Sep 19, 2008 at 5:10

Derek Park

46.8k ● 16 ● 59 ● 76

3

I prefer to keep members private as long as possible and only access em via getters, even from within the very same class. I also try to avoid setters as a first draft to promote value style objects as long as it is possible. Working with dependency injection a lot you often have setters but no getters, as clients should be able to configure the object but (others) not get to know what's acutally configured as this is an implementation detail.

Regards, Ollie

Share Improve this answer

Follow

edited Apr 23, 2009 at 9:13

answered Sep 19, 2008 at 4:55

Oliver Drotbohm

82.8k ● 18 ● 232 ● 216

I tend to follow the rule pretty strictly, even when it's just my own code. I really like Properties in C# for that reason. It makes it really easy to control what values it's given, but you can still use them as variables. Or make the set private and the get public, etc.

2

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:51

Joel

16.6k ● 18 ● 75 ● 93

I love properties too. Shame C++ doesn't have them.
– Thomas Sep 19, 2008 at 4:54

Basically, information hiding is about code clarity. It's designed to make it easier for someone else to extend your code, and prevent them from accidentally creating bugs when they work with the internal data of your classes. It's based on the principle that **nobody ever reads comments**, especially ones with instructions in them.

2

Example: I'm writing code that updates a variable, and I need to make absolutely sure that the Gui changes to reflect the change, the easiest way is to add an accessor method (aka a "Setter"), which is called instead of updating data is updated.

If I make that data public, and something changes the variable without going through the Setter method (and this happens every swear-word time), then someone will need to spend an hour debugging to find out why the updates aren't being displayed. The same applies, to a lesser extent, to "Getting" data. I could put a comment in the header file, but odds are that no-one will read it till something goes terribly, terribly wrong. Enforcing it with private means that the mistake *can't* be made, because it'll show up as an easily located compile-time bug, rather than a run-time bug.

From experience, the only times you'd want to make a member variable public, and leave out Getter and Setter methods, is if you want to make it absolutely clear that changing it will have no side effects; especially if the data structure is simple, like a class that simply holds two variables as a pair.

This should be a fairly rare occurence, as normally you'd *want* side effects, and if the data structure you're creating is so simple that you don't (e.g a pairing), there will already be a more efficiently written one available in a Standard Library.

With that said, for most small programs that are one-use no-extension, like the ones you get at university, it's more "good practice" than anything, because you'll remember over the course of writing them, and then you'll hand them in and never touch the code again. Also, if you're writing a data structure as a way of finding out about how

they store data rather than as release code, then there's a good argument that Getters and Setters will not help, and will get in the way of the learning experience.

It's only when you get to the workplace or a large project, where the probability is that your code will be called to by objects and structures written by different people, that it becomes vital to make these "reminders" strong. Whether or not it's a single man project is surprisingly irrelevant, for the simple reason that "you six weeks from now" is as different person as a co-worker. And "me six weeks ago" often turns out to be lazy.

A final point is that some people are pretty zealous about information hiding, and will get annoyed if your data is unnecessarily public. It's best to humour them.

Share Improve this answer

Follow

edited Apr 7, 2016 at 3:11

community wiki
2 revs, 2 users 97%
deworde

1

C# Properties 'simulate' public fields. Looks pretty cool and the syntax really speeds up creating those get/set methods

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:55

Richard Walton
4,785 ● 3 ● 40 ● 49

1

Keep in mind the semantics of invoking methods on an object. A method invocation is a very high level abstraction that can be implemented my the compiler or the run time system in a variety of different ways.

If the object who's method you are invoking exists in the same process/ memory map then a method could well be optimized by a compiler or VM to directly access the data member. On the other hand if the object lives on another node in a distributed system then there is no way that you can directly access it's internal data members, but you can still invoke its methods my sending it a message.

By coding to interfaces you can write code that doesn't care where the target object exists or how it's methods are invoked or even if it's written in the same language.

In your example of an object that implements all the methods of a collection, then surely that object actually *is* a collection. so maybe this would be a case where inheritance would be better than encapsulation.

Share Improve this answer

Follow

answered Feb 10, 2009 at 18:38

Noel Walters

1,853 ● 1 ● 14 ● 20

0

It's all about controlling what people can do with what you give them. The more controlling you are the more assumptions you can make.

Also, theorectically you can change the underlying implementation or something, but since for the most part it's:

```
private Foo foo;
public Foo getFoo() {}
public void setFoo(Foo foo) {}
```

It's a little hard to justify.

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:47

SCdF

59.3k ● 24 ● 79 ● 114

2 LOL, I feel yah! That's why in C# they introduced the concept of Public Foo Foo { get; set; } so that you don't even have to create a private member variable – Jon Limjap Sep 19, 2008 at 4:54

An obvious justification is that setFoo() can do checks on foo if it fits into allowed range, for example. Also (when there is no setFoo), you can have a private member of class "FooImpl extends Foo" and have getFoo() return it as Foo, allowing you to change implementations at will. – quant_dev Feb 10, 2009 at 18:09

@quant_dev yes it *can*, but in practice it almost never does. I don't think I have ever seen validation done in the bean, it's alwasy done sometime beforehand (e.g. part of the lifecycle of the web framework if we're talking about the web). Domain

objects are always more flexible sans logic – SCdF Feb 10, 2009 at 19:38

@Jon, they could really use properties in Java – Alex Baranosky Oct 2, 2009 at 3:09

0

Encapsulation is important when at least one of these holds:

1. Anyone but you is going to use your class (or they'll break your invariants because they don't read the documentation).
2. Anyone who doesn't read the documentation is going to use your class (or they'll break your carefully documented invariants). Note that this category includes you-two-years-from-now.
3. At some point in the future someone is going to inherit from your class (because maybe an extra action needs to be taken when the value of a field changes, so there has to be a setter).

If it is just for me, and used in few places, and I'm not going to inherit from it, and changing fields will not invalidate any invariants that the class assumes, *only then* I will occasionally make a field public.

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:53

Thomas

181k ● 55 ● 376 ● 501

0

My tendency is to try to make everything private if possible. This keeps object boundaries as clearly defined as possible and keeps the objects as decoupled as possible. I like this because when I have to rewrite an object that I botched the first (second, fifth?) time, it keeps the damage contained to a smaller number of objects.

If you couple the objects tightly enough, it may be more straightforward just to combine them into one object. If you relax the coupling constraints enough you're back to structured programming.

It may be that if you find that a bunch of your objects are just accessor functions, you should rethink your object divisions. If you're not doing any actions on that data it may belong as a part of another object.

Of course, if you're writing a something like a library you want as clear and sharp of an interface as possible so others can program against it.

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:55

Jonathan Adelson

3,321 ● 5 ● 32 ● 39

0

Fit the tool to the job... recently I saw some code like this in my current codebase:

```
private static class SomeSmallDataStructure {
    public int someField;
    public String someOtherField;
}
```

And then this class was used internally for easily passing around multiple data values. It doesn't always make sense, but if you have just DATA, with no methods, and you aren't exposing it to clients, I find it a quite useful pattern.

The most recent use I had of this was a JSP page where I had a table of data being displayed, defined at the top declaratively. So, initially it was in multiple arrays, one array per data field... this ended in the code being rather difficult to wade through with fields not being next to eachother in definition that would be displayed together... so I created a simple class like above which would pull it together... the result was REALLY readable code, a lot more so than before.

Moral... sometimes you should consider "accepted bad" alternatives if they may make the code simpler and easier to read, as long as you think it through and consider the consequences... don't blindly accept EVERYTHING you hear.

That said... public getters and setters is pretty much equivalent to public fields... at least essentially (there is a tad more flexibility, but it is still a bad pattern to apply to EVERY field you have).

Even the java standard libraries has some cases of public fields.

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:57

Mike Stone

44.6k ● 30 ● 114 ● 140

0

When I make objects meaningful they are **easier** to **use** and easier to **maintain**.

For example: Person.Hand.Grab(howquick, howmuch);

The trick is not to think of members as simple values but objects in themselves.

Share Improve this answer

Follow

answered Sep 19, 2008 at 5:33

Al.

712 ● 1 ● 7 ● 12

0

I would argue that this question does mix-up the concept of encapsulation with 'information hiding'
(this is not a critic, since it does seem to match a common interpretation of the notion of 'encapsulation')

However for me, 'encapsulation' is either:

- the process of regrouping several items into a container
- the container itself regrouping the items

Suppose you are designing a tax payer system. For each tax payer, you could *encapsulate* the notion of **child** into

- a list of children representing the children
- a map of to takes into account children from different parents
- an object Children (not Child) which would provide the needed information (like total number of children)

Here you have three different kinds of encapsulations, 2 represented by low-level container (list or map), one represented by an object.

By making those decisions, you do not

- make that encapsulation public or protected or private: that choice of 'information hiding' is still to be made
- make a complete abstraction (you need to refine the attributes of object Children and you may decide to create an object Child, which would keep only the relevant informations from the point of view of a tax payer system)
  Abstraction is the process of choosing which attributes of the object are relevant to your system, and which must be completely ignored.

So my point is:
That question may been titled:
Private vs. Public members in practice (how important is *information hiding*?)

Just my 2 cents, though. I perfectly respect that one may consider encapsulation as a process including 'information hiding' decision.

However, I always try to differentiate 'abstraction' - 'encapsulation' - 'information hiding or visibility'.

Share Improve this answer

Follow

answered Sep 19, 2008 at 6:25

VonC

1.3m ● 558 ● 4.7k ● 5.6k

> Right... just realized my point is already in StackOverflow ;) stackoverflow.com/questions/24626/… And also: c2.com/cgi/wiki?EncapsulationIsNotInformationHiding – VonC Sep 19, 2008 at 6:32

0

@VonC

You might find the International Organisation for Standardization's, "Reference Model of Open Distributed Processing," an interesting read. It defines: "Encapsulation: the property that the information contained in an object is accessible only through interactions at the interfaces supported by the object."

I tried to make a case for information hiding's being a critical part of this definition here: http://www.edmundkirwan.com/encap/s2.html

Regards,

Ed.

Share Improve this answer

Follow

answered Oct 31, 2008 at 11:05

Ed Kirwan

Sorry for the delay, Ed: I did not check that question until now: you should leave a comment on my answer: it does appear on my response tab of my profile – VonC Nov 6, 2008 at 6:28

Interesting paper, which does not contradict the fact that, as I said, encapsulation can be viewed as a process including 'information hiding' decision. – VonC Nov 6, 2008 at 6:29

0

I find lots of getters and setters to be a code smell that the structure of the program is not designed well. You should look at the code that uses those getters and setters, and look for functionality that really should be part of the class. In most cases, the fields of a class should be private implementation details and only the methods of that class may manipulate them.

Having both getters and setters is equal to the field being public (when the getters and setters are trivial/generated automatically). Sometimes it might be better to just declare the fields public, so that the code will be more simple, unless you need polymorphism or a framework requires get/set methods (and you can't change the framework).

But there are also cases where having getters and setters is a good pattern. One example:

When I create the GUI of an application, I try to keep the behaviour of the GUI in one class (FooModel) so that it can be unit tested easily, and have the visualization of the GUI in another class (FooView) which can be tested only manually. The view and model are joined with simple glue code; when the user changes the value of field `x`, the view calls `setX(String)` on the model, which in turn may raise an event that some other part of the model has changed, and the view will get the updated values from the model with getters.

In one project, there is a GUI model which has 15 getters and setters, of which only 3 get methods are trivial (such that the IDE could generate them). All the others contain some functionality or non-trivial expressions, such as the following:

```
public boolean isEmployeeStatusEnabled() {
    return
pinCodeValidation.equals(PinCodeValidation.VALID);
}

public EmployeeStatus getEmployeeStatus() {
    Employee employee;
    if (isEmployeeStatusEnabled()
            && (employee = getSelectedEmployee())
!= null) {
        return employee.getStatus();
    }
    return null;
}

public void setEmployeeStatus(EmployeeStatus
status) {
    getSelectedEmployee().changeStatusTo(status,
getPinCode());
```

```
    fireComponentStateChanged();
}
```

Share Improve this answer

Follow

edited Feb 10, 2009 at 0:02

answered Feb 9, 2009 at 23:44

Esko Luontola

73.6k ● 17 ● 118 ● 128

-1

In practice I always follow only one rule, the "no size fits all" rule.

Encapsulation and its importance is a product of your project. What object will be accessing your interface, how will they be using it, will it matter if they have unneeded access rights to members? those questions and the likes of them you need to ask yourself when working on each project implementation.

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:48

RomanM

6,671 ● 9 ● 35 ● 42

-1

I base my decision on the Code's depth within a module. If I'm writting code that is internal to a module, and does not interface with the outside world I don't encapsulate things with private as much because it affects my programmer performance (how fast I can write and rewrite my code).

But for the objects that server as the module's interface with user code, then I adhere to strict privacy patterns.

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:49

Robert Gould

69.7k ● 61 ● 191 ● 275

-1

Certainly it makes a difference whether your writing internal code or code to be used by someone else (or even by yourself, but as a contained unit.) Any code that is going to be used externally should have a well defined/documented interface that you'll want to change as little as possible.

For internal code, depending on the difficulty, you may find it's less work to do things the simple way now, and pay a little penalty later. Of course Murphy's law will ensure that the short term gain will be erased many times over in having to make wide-ranging changes later on where you needed to change a class' internals that you failed to encapsulate.

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:50

Eclipse

45.5k ● 20 ● 116 ● 172

-1

Specifically to your example of using a collection that you would return, it seems possible that the implementation of

such a collection might change (unlike simpler member variables) making the utility of encapsulation higher.

That being said, I kinda like Python's way of dealing with it. Member variables are public by default. If you want to hide them or add validation there are techniques provided, but those are considered the special cases.

Share Improve this answer

Follow

answered Sep 19, 2008 at 4:56

John Mulder

10.1k ● 7 ● 34 ● 37

-1

I follow the rules on this almost all the time. There are four scenarios for me - basically, the rule itself and several exceptions (all Java-influenced):

1. Usable by anything outside of the current class, accessed via getters/setters
2. Internal-to-class usage typically preceded by 'this' to make it clear that it's not a method parameter
3. Something meant to stay extremely small, like a transport object - basically a straight shot of attributes; all public
4. Needed to be non-private for extension of some sort

Share Improve this answer

Follow

answered Sep 19, 2008 at 5:06

agartzke

3,905 ● 6 ● 26 ● 18

-1

There's a practical concern here that isn't being addressed by most of the existing answers. Encapsulation and the exposure of clean, safe interfaces to outside code is always great, but it's much more important when the code you're writing is intended to be consumed by a spatially- and/or temporally-large "user" base. What I mean is that if you plan on somebody (even you) maintaining the code well into the future, or if you're writing a module that will interface with code from more than a handful of other developers, you need to think much more carefully than if you're writing code that's either one-off or wholly written by you.

Honestly, I know what wretched software engineering practice this is, but I'll oftentimes make everything public at first, which makes things marginally faster to remember and type, then add encapsulation as it makes sense. Refactoring tools in most popular IDEs these days makes which approach you use (adding encapsulation vs. taking it away) much less relevant than it used to be.

Share Improve this answer Follow

answered Sep 19, 2008 at 5:32

community wiki
Matt J