Always Learning

Monday, 23 September 2013

Public properties, getters and setters, or magic?

Opinion seems to be divided on whether it is good practice to use public properties in PHP classes, or whether to use getter and setter methods instead (and keep the properties private or protected). A sort of hybrid third option is to use the magic methods __get() and __set(). As always, there are advantages and disadvantages to each approach, so let's take a look at them...

Public properties example

```php
class Foo
{
    public $bar;
}

$foo = new Foo();
$foo->bar = 1;
$foo->bar++;
```

In this example, bar is a public property of Foo. The calling code can manipulate that property in any way it likes, and stuff any old data in it.

Public properties advantages

- Compared with getter and setter methods, there is a lot less typing involved!
- The calling code is more readable and easier to work with than getter and setter method calls.
- A call to a public property (to either set or get) is faster and uses less memory than a method call - but the saving is small unless you are calling it many times in a long loop.
- Objects with public properties can be used as parameters for some PHP functions (such as http_build_query), and can be serialized by json_encode.

Public properties disadvantages

- There is no way of controlling what data is held in the properties - it is very easy to populate them with data that the class methods might not expect. This can be particularly problematic if the class is being used by another developer (ie. other than the author of the class), as they might not (and should not need to) be aware of the internal workings of the class (and let's face it, after a few months, the author of the class probably won't remember the internal workings either).
- If you want to expose a public property in an API, you can't do so using an interface, as in PHP interfaces only allow method definitions.

Using getters and setters example

```php
class Foo
{
    private $bar;

    public function getBar()
    {
        return $this->bar;
    }

    public function setBar($bar)
    {
        $this->bar = $bar;
    }
}

$foo = new Foo();
$foo->setBar(1);
$foo->setBar($foo->getBar() + 1);
```

Here the bar property is private so cannot be accessed by the calling code directly. The caller has to use the getBar method to retrieve the value and the setBar method to set it. The methods can perform validation processing to ensure that only valid values are allowed through.

Getters and setters advantages

- With getters and setters you can control exactly what is stored in your object properties, and reject any values that are not valid.
- You also have the option of performing additional processing when a value is set or retrieved (for example, if updating this property should trigger some action such as notifying an observer).
- When setting a value which represents an object or array (rather than a scalar value), you can specify the type in the function declaration (eg. `public function setBar(Bar $bar)`). Such a damn shame PHP won't let you do the same thing with ints and strings!
- If the value of a property has to be loaded from an external data source or the runtime environment, you can lazy load it - so the resources required to load the data are only used if the property itself is called. Of course you would need to be careful that you don't needlessly load the data from the external source on every call to the property. And it would be more common to make a single call to the database to populate all the properties rather than fetching them one at a time.
- You can make properties read-only or write-only by only creating a getter but not a setter or vice-versa.
- You can include a getter and setter in an interface to expose them in an API.

Getters and setters disadvantages

- For developers who are used to accessing properties directly, getters and setters are a pain in the neck to use! For every property you have to define the property itself, a getter, and a setter, and to use the property in the calling code you have to make extra method calls - it is much easier to say `$foo->bar++;` rather than the long-winded `$foo->setBar($foo->getBar() + 1);` (although of course you could add yet another method such as `$foo->incrementBar();`)
- As noted above, there is a small additional overhead when making a method call over using a plain ol' public property.
- By convention, getter and setter methods start with the verbs 'get' and 'set' respectively, but these verbs are also commonly used for other methods which don't necessarily relate to properties. This is not necessarily a problem for the calling code as you might not care what the implementation is, but the ambiguity about the type of implementation you are dealing with can make the code harder to understand.

Magic method getters and setters example

```
class Foo
{
    protected $bar;

    public function __get($property)
    {
        switch ($property)
        {
            case 'bar':
                return $this->bar;
            //etc.
        }
    }

    public function __set($property, $value)
    {
        switch ($property)
        {
            case 'bar':
                $this->bar = $value;
                break;
            //etc.
        }
    }
}

$foo = new Foo();
$foo->bar = 1;
$foo->bar++;
```

Here, the bar property is not publicly exposed, but the calling code still calls it as though it were public. When PHP can't find a matching public property, it calls the appropriate magic method (__get() for retrieving a value, __set() for assigning a value). This might seem like the best of both worlds, but it has a major drawback (see disadvantages below!). Note also that __get and __set are NOT called if a matching public property exists, but they are called if a protected or private property exists but is out of scope, or if no property exists.

Magic method getter and setter advantages

- You can manipulate properties directly from the calling code (as though they were public properties) but still have complete control over what data goes into what property.
- As with declared getter and setter methods, you can perform additional processing when the property is used.
- You can use lazy loading.
- You can make properties read-only or write-only.
- You can use the magic methods as a catch-all to handle calls to properties that don't exist, and still process them in some way (effectively allowing the calling code to decide what property names to use).

Magic method getter and setter disadvantages

- ~~The showstopper for magic methods is that they do not expose the property in any way. To use or extend the class, you have to 'just know' what properties it supports. This is simply unacceptable most of the time (unless perhaps you are one of those hard core programmers who think notepad is an IDE!), although there are times when the advantages mentioned above outweigh this limitation.~~ As pointed out by a commentator below, this disadvantage can be overcome by using the phpDoc tags @property, @property-read, and @property-write. Cool.

Which option to use

Clearly there are some significant advantages to getters and setters, and some people feel that they should therefore be used all the time (especially those who come from a Java background!). But in my opinion, they do break the natural flow of the language, and their additional complexity and verbosity puts me off using them unless there is a need to do so (I find it a little irritating when naive getters and setters don't actually DO anything other than get or set the property). My choice then is to use public properties most of the time, but getters and setters for critical settings that I feel need stricter control, would benefit from lazy loading, or that I want to expose in an interface.

Another alternative?

Before I learned PHP, I used to use C#. In C#, all properties have accessor methods, but you don't have to call them as methods, you can just manipulate the properties directly and the corresponding methods are called magically. So it is similar to PHP's magic __get and __set, but the properties are still declared and exposed. This really is the best of both worlds, and it would be great to see a feature like this in PHP.

Tragically though, the RFC for a C#-like property accessor syntax feature could not quite muster up the two-thirds majority needed to take it forward: https://wiki.php.net/rfc/propertygetsetsyntax-v1.2 Bah!

Posted by Russell Walker at 10:36

12 comments:

Adam 26 September 2013 at 19:23

This is an interesting article. I'd like to comment that the disadvantage you list for magic methods doesn't have to exist if you require that the property be declared before you return/set it.

I use the magic methods quite often. They are immensely useful when doing things like audit logging. In my magic methods, the first thing I always do is check if the property exists in the object. If it does not, an exception is thrown. Next, I check to see if a setter/getter exists for the requested property. After I verify that the property exists in the object, I check for a method named 'getXXX' or 'setXXX'. If that method exists, I call it with the values passed to the magic method. If the method does not exist, I set/return the requested property as you would if the property was public.

I must use discipline to name the getter/setter methods properly, but other than that I find it to be a useful pattern. The kicker is to enforce that all properties are declared before they are allowed to magically be accessed.

Reply

Replies

Russell Walker 26 September 2013 at 19:37

That's an interesting way of doing things, but (unless I'm misunderstanding you) it still wouldn't allow for auto-completion in an IDE, which I think is a major problem.

Adam 26 September 2013 at 21:00

You probably are correct about that. I misunderstood what you meant. However, for me, the functionality they add outweighs this one drawback, especially since most modern IDEs make it super easy to find/see the class declaration so you can see what properties are available. Sure, it isn't as convenient as having code completion available for all the properties, but it is easy enough to take a quick glance at the class source.

To each his own I suppose.

Cheers!

Steve B 27 September 2013 at 13:01

Depends on the IDE. I use phpstorm and it appears to pick up the property declarations and auto-complete off these without problems.

Reply

Guido van Biemen 26 September 2013 at 22:33

Nice article weighing the pros and cons of either method. Strugling with needless amounts of getter/setter code and property access control issues, I've come up with a trait that helps implement magic get and set methods using __call or setup limited property access through __get and __set. The trait uses annotation tags like @get and @set to enable get or set access to a property and also supports basic validation/filtering.

If you're interested, take a look at https://github.com/guidovanbiemen/setngeti

Reply

Replies

Russell Walker 27 September 2013 at 09:52

Thanks, interesting to see how people work round things like this!

Reply

Gofromiel 27 September 2013 at 09:28

Hi,

Your article is very interesting, although I don't understand your concern about magic properties auto-completion since they can be declared using @property (and @property-read and @property-write).

I use magic properties extensively in my framework because I find them more intuitive (and beautiful) than getXxx() / setXxx(). I can create façade properties, read-only and write-only properties, and transparent lazy loading.

I created a library for this purpose that also allows objects to be extended with properties/methods at runtime. Your opinion would be very valuable to me. https://github.com/ICanBoogie/Prototype

Another example would be my DateTime implementation. https://github.com/ICanBoogie/DateTime

Best regards,

Olivier

Reply

Replies

Russell Walker 27 September 2013 at 10:08

Hey, you're right! I didn't know about @property. Although it would be nicer if it were part of the language (like C# accessors), I guess using phpDoc pretty much eliminates the main disadvantage I was worried about. I'm glad I posted this now - always learning! :)

Reply

christian calloway 4 October 2013 at 02:53

sigh.. the problem in the php community is that there are too many opinionated people whom lack the prerequisite knowledge as thus lead others astray.

```
public function __get($property)
{
switch ($property)
{
case 'bar':
return $this->bar;
//etc.
}
}
```

First of all, a case statement.. really? How about just "return $this->$property" and "$this->$property = $value". Secondly, this doesn't make sense in the first place.. if the property exists, then instance#__get will never be called and even if it magically were to do so, you'd have a big fat blowup attempting to return a property that doesn't exist. If you WERE to implement a get/set, you should use lookup table (hash, associative array, dictionary.. depending on which language you are most familiar), but you are in essence only creating "vertical" public properties, though you are NOT polluting the instance's context. Third, the biggest tradeoff in exploiting PHP's limited metaprogramming capabilities are an increase in complexity and a (possibly severe) performance penalty.

What you are looking for is something like attr_accessor, attr_reader, attr_writer..

Reply

Replies

Russell Walker 4 October 2013 at 08:19

I'm trying not to sound opinionated, honestly. That's why I chose the title 'Always Learning' for my blog, and accepted correction from a previous poster. The idea here is to share my thoughts and grow in understanding myself by soliciting comments from others.

>First of all, a case statement.. really? How about just "return $this->$property" and "$this->$property = $value".

The implication of using a case statement is that you might have other stuff to do. If you are just going to return $this->$property, I can't see any point in using a getter at all. It also allows you to handle or throw an exception if someone attempts to access a property that you don't want them to.

>if the property exists, then instance#__get will never be called and even if it magically were to do so, you'd have a big fat blowup attempting to return a property that doesn't exist.

Maybe I misunderstand your point but that seems to be a contradiction (if it exists it would blow up because it doesn't exist?). The magic __get function is called if the property exists but is out of scope (ie. protected or private), which is what the code sample here is demonstrating.

>the biggest tradeoff in exploiting PHP's limited metaprogramming capabilities are an increase in complexity and a (possibly severe) performance penalty

I think I already made those points in the blog post.

>What you are looking for is something like attr_accessor, attr_reader, attr_writer..

Well, I used the example of C# accessors, you can use the example of Ruby accessors if you wish. It looks like magic methods with phpDoc tags are the closest we are going to get in PHP though.

Reply

Dmitry Dmitriev 9 October 2013 at 11:17

Magic methods have their disadvantages:
they are slow
their body can be hard to read if your class has a lot of magic properties (can be countered by calling sub-methods, but it will be even slower)

I'm too lazy to actually measure how slow they are, so here is some old benchmark: http://www.garfieldtech.com/blog/magic-benchmarks

Reply

olvlvl 3 November 2013 at 23:59

@Dmirty: This benchmark is seven years old !! I'd like to see one for PHP 5.4 ?

Reply

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

About Me

Russell Walker

View my complete profile

Google+ Followers