When to declare classes final

By Marco Pivetta / @Ocramius

TL;DR: Make your classes always `final`, if they implement an interface, and no other public methods are defined

In the last month, I had a few discussions about the usage of the `final` marker on PHP classes.

The pattern is recurrent:

1. I ask for a newly introduced class to be declared as `final`
2. the author of the code is reluctant to this proposal, stating that `final` limits flexibility
3. I have to explain that flexibility comes from good abstractions, and not from inheritance

It is therefore clear that coders need a better explanation of **when** to use `final`, and when to avoid it.

There are many other articles about the subject, but this is mainly thought as a "quick reference" for those that will ask me the same questions in future.

When to use "final":

`final` should be used **whenever possible**.

Why do I have to use `final`?

There are numerous reasons to mark a class as `final`: I will list and describe those that are most relevant in my opinion.

1. Preventing massive inheritance chain of doom

Developers have the bad habit of fixing problems by providing specific subclasses of an existing (not adequate) solution. You probably saw it yourself with examples like following:

```php
<?php

class Db { /* ... */ }
class Core extends Db { /* ... */ }
class User extends Core { /* ... */ }
class Admin extends User { /* ... */ }
class Bot extends Admin { /* ... */ }
class BotThatDoesSpecialThings extends Bot { /* ... */ }
class PatchedBot extends BotThatDoesSpecialThings { /* ... */ }
```

This is, without any doubts, how you should **NOT** design your code.

The approach described above is usually adopted by developers who confuse OOP with "a way of solving problems via inheritance" ("inheritance-oriented-programming", maybe?).

2. Encouraging composition

In general, preventing inheritance in a forceful way (by default) has the nice advantage of making developers think more about composition.

There will be less stuffing functionality in existing code via inheritance, which, in my opinion, is a symptom of haste combined with feature creep.

Take the following naive example:

```php
<?php

class RegistrationService implements RegistrationServiceInterface
{
    public function registerUser(/* ... */) { /* ... */ }
}

class EmailingRegistrationService extends RegistrationService
{
    public function registerUser(/* ... */)
    {
        $user = parent::registerUser(/* ... */);

        $this->sendTheRegistrationMail($user);

        return $user;
    }

    // ...
}
```

By making the `RegistrationService` `final`, the idea behind `EmailingRegistrationService` being a child-class of it is denied upfront, and silly mistakes such as the previously shown one are easily avoided:

```php
<?php

final class EmailingRegistrationService implements RegistrationServiceInterface
{
    public function __construct(RegistrationServiceInterface $mainRegistrationService)
    {
        $this->mainRegistrationService = $mainRegistrationService;
    }

    public function registerUser(/* ... */)
    {
        $user = $this->mainRegistrationService->registerUser(/* ... */);

        $this->sendTheRegistrationMail($user);

        return $user;
    }

    // ...
}
```

3. Force the developer to think about user public API

Developers tend to use inheritance to add accessors and additional API to existing classes:

```php
<?php

class RegistrationService implements RegistrationServiceInterface
{
    protected $db;

    public function __construct(DbConnectionInterface $db)
    {
        $this->db = $db;
    }

    public function registerUser(/* ... */)
    {
        // ...

        $this->db->insert($userData);

        // ...
    }
}

class SwitchableDbRegistrationService extends RegistrationService
{
    public function setDb(DbConnectionInterface $db)
    {
        $this->db = $db;
    }
}
```

This example shows a set of flaws in the thought-process that led to the `SwitchableDbRegistrationService`:

- The `setDb` method is used to change the `DbConnectionInterface` at runtime, which seems to hide a different problem being solved: maybe we need a `MasterSlaveConnection` instead?
- The `setDb` method is not covered by the `RegistrationServiceInterface`, therefore we can only use it when we strictly couple our code with the `SwitchableDbRegistrationService`, which defeats the purpose of the contract itself in some contexts.
- The `setDb` method changes dependencies at runtime, and that may not be supported by the `RegistrationService` logic, and may as well lead to bugs.
- Maybe the `setDb` method was introduced because of a bug in the original implementation: why was the fix provided this way? Is it an actual fix or does it only fix a symptom?

There are more issues with the `setDb` example, but these are the most relevant ones for our purpose of explaining why `final` would have prevented this sort of situation upfront.

4. Force the developer to shrink an object's public API

Since classes with a lot of public methods are very likely to break the SRP, it is often true that a developer will want to override specific API of those classes.

Starting to make every new implementation `final` forces the developer to think about new APIs upfront, and about keeping them as small as possible.

5. A `final` class can always be made extensible

Coding a new class as `final` also means that you can make it extensible at any point in time (if really required).

No drawbacks, but you will have to explain your reasoning for such change to yourself and other members in your team, and that discussion may lead to better solutions before anything gets merged.

6. `extends` breaks encapsulation

Unless the author of a class specifically designed it for extension, then you should consider it `final` even if it isn't.

Extending a class breaks encapsulation, and can lead to unforeseen consequences and/or BC breaks: think twice before using the `extends` keyword, or better, make your classes `final` and avoid others from having to think about it.

7. You don't need that flexibility

One argument that I always have to counter is that `final` reduces flexibility of use of a codebase.

My counter-argument is very simple: you don't need that flexibility.

Why do you need it in first place? Why can't you write your own customized implementation of a contract? Why can't you use composition? Did you carefully think about the problem?

If you still need to remove the `final` keyword from an implementation, then there may be some other sort of code-smell involved.

8. You are free to change the code

Once you made a class `final`, you can change it as much as it pleases you.

Since encapsulation is guaranteed to be maintained, the only thing that you have to care about is that the public API.

Now you are free to rewrite everything, as many times as you want.

When to avoid `final`:

Final classes **only work effectively under following assumptions**:

1. There is an abstraction (interface) that the final class implements
2. All of the public API of the final class is part of that interface

If one of these two pre-conditions is missing, then you will likely reach a point in time when you will make the class extensible, as your code is not truly relying on abstractions.

An exception can be made if a particular class represents a set of constraints or concepts that are totally immutable, inflexible and global to an entire system. A good example is a mathematical operation: `$calculator->sum($a, $b)` will unlikely change over time. In these cases, it is safe to assume that we can use the `final` keyword without an abstraction to rely on first.

Another case where you do not want to use the `final` keyword is on existing classes: that can only be done if you follow semver and you bump the major version for the affected codebase.

Try it out!

After having read this article, consider going back to your code, and if you never did so, adding your first `final` marker to a class that you are planning to implement.

You will see the rest just getting in place as expected.