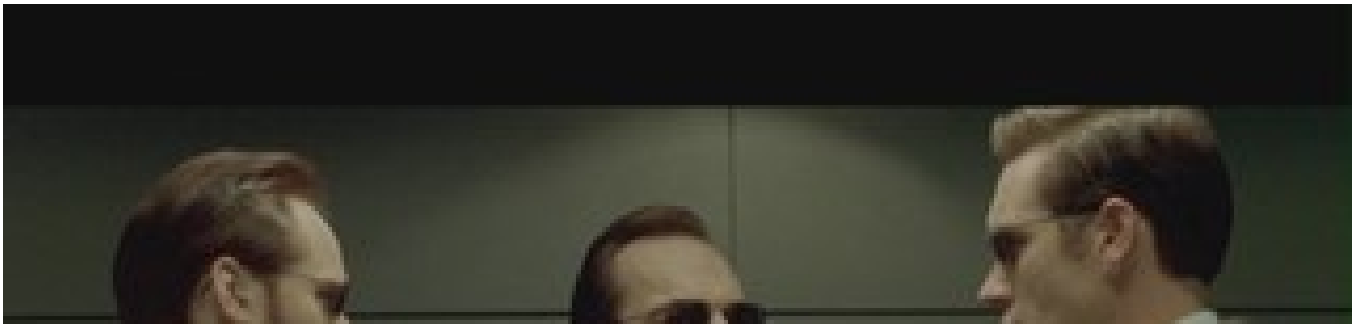
Community

yegor256.com

33 Comments • Created 5 days ago



There Can Be Only One Primary Constructor

In a properly designed class, there can only be one primary constructor and any number of secondary ones.

(yegor256.com)

33 Comments

Damokles • 3 days ago

Some programming languages have this distinction baked into the language specification and checked by the compiler.

I think of Swift as a concrete example: They have designated and convenience initializers, which have a strict ruleset to follow.

A convenience initializer has to call either another convenience initializer or one of the designated initializers (you can have more than one) and at the end of the chain a designated initializer has to be called.

I really like the idea because it takes away pain when dealing with subclasses (which constructors do i need to overwrite again?).

There are also other rules concerning calling the superclass constructor, initializing properties, etc.

If you're interested, you can check out the documentation here: https://developer.apple.com/li...

1 ^ | v • Reply • Share ›

Riccardo Cardin ➚ Damokles • 2 days ago

Also Scala language has the same concept of "Primary constructor" and convinience ones. Indeed, in Scala the type declaration contains the signature of the primary constructor.

```
class Person(val name: String, val surname: String) { /* ... */ }
```

^ | v • Reply • Share ›

Lyubomyr Shaydariv • 4 days ago

Generally speaking, I like the idea of having one constructor per a class, because such a constructor usually just assigns the parameters to the instance fields. I almost always make constructors private, so the secondary constructors do not exist (probably except of protected constructors in abstract classes), but there are static factory methods that have a few benefits over constructors: you can declare a more abstract type rather than a concrete type letting you bind to interfaces or abstract classes only; you can implement object cache and not necessarily always create new instances; hide complex object construction leaving the "primary" constructor as clean as possible.

1 ^ | v • Reply • Share ›

Kata Tunix ➚ Lyubomyr Shaydariv • 4 days ago

If you used Gang of Four's Factory Method and Abstract Factory (call them Factory for short), I would agree in almost cases. But you are using "static factory methods", I think it is an anti-pattern.

The core reason of using Factory is satisfying the "Dependency inversion principle" (DIP). Indeed, any line of code that uses the "new" keyword violates this principle. So we create abstract things such as classes/methods that are responsible for creating our product objects. With "static factory methods", you are still depending on concrete things. Since you cannot extend your "static factory methods", you also violate the "Open/closed principle".

Even with Factory, you should not overuse them, because:

> A strict interpretation of DIP would insist on using factories for every volatile class in the system. What's more, the power of the FACTORY pattern is seductive. These two factors can sometimes lure developers into using factories by default. This is an extreme that I don't recommend.
>
> Factories are a complexity that can often be avoided, especially in the early phases of an evolving design. When they are used by default, factories dramatically increase the difficulty of extending the design. In order to create a new class, one may have to create as many as four new classes: the two interface classes that represent the new class and its factory and the two concrete classes that implement those interfaces.

> concrete classes that implement these interfaces.
>
> -- Robert C. Martin

1 ^ | v • Reply • Share ›

Lyubomyr Shaydariv ➚ Kata Tunix • 4 days ago

Well, I'm only referring the object instantiation mechanism, and picked the term "factory" to let constructors get hidden behind the "factory" mechanism. Yes, I agree that statics are pure evil for "application" objects, but one cannot avoid them at the bottom-most level. Also, I admit that such static methods are good for pure utility code that is not aware of the application it is used for (Guava, I think, is a good example for such an approach).

1 ^ | v • Reply • Share ›

Kata Tunix ➚ Lyubomyr Shaydariv • 4 days ago

Sorry, I still don't get the point about your "factory". Maybe showing some code is better.

Let's define our objective: creating a Cash object in a flexible way; input can be either {/*empty*/} or {cents} or {cents, currency}. Does your "factory" code look like:

```
class Cash
{
    private int cents;
    private String currency;

    private Cash()
    {
        this(0);
    }

    private Cash(int cents)
```

see more

^ | v • Reply • Share ›

Lyubomyr Shaydariv ➚ Kata Tunix • 4 days ago

```
public final class Cash {

        private final int cents;
        private final String currency;

        private Cash(final int cents, final String currency) {
                this.cents = cents;
                this.currency = currency;
        }
```

```
        }

        public static Cash cash() {
                return cash(0);
        }

        public static Cash cash(final int cents) {
                return cash(cents, "USD");
        }

        public static Cash cash(final int cents, final String cu
                return new Cash(cents, currency);
        }

}
```

• Reply • Share ›

Kata Tunix ➚ Lyubomyr Shaydariv • 4 days ago

Great code!

First, your code still violates the "Dependency inversion principle": **Cash c =** Cash.cash**();** and **Cash c = new Cash();** are the same. Your code even introduces a method name "cash". But let's suppose it is harmless in this context. We can ignore DIP here.

Next, you said "declare a more abstract type rather than a concrete type letting you bind to interfaces or abstract classes only". Did you mean you have a derived class e.g. Cash2? And in the Cash.cash() method you can return an instance of Cash2? If I'm wrong, could you please show code again for this? Pseudo code is okay.

Finally, "implement object cache and not necessarily always create new instances". I agree with this benefit. But the caching job should be assigned to another class so we don't violate the "Single responsibility principle".

1 • Reply • Share ›

Lyubomyr Shaydariv ➚ Kata Tunix • 4 days ago

I really think that the Cash class was not the best choice to demonstrate the idea in full...

Regarding the 1st: The cash() method only implies the Cash constructor only for the concrete case. No constructors -- no application.

Regarding the 2nd: Not really. If Cash implements ICash, then the "static factory" might be declared like:

```
public final class Cash implements ICash {
    private Cash(...) { ... }
    public static ICash cash() { return new Cash(); }
                  ^___ hidden, behind the scenes, the client sho
}
```

Regarding the 3rd: I agree with SRP in general, but just imagine that Integer.valueOf(int i) in Java requires an external cache dependency to your cache storage/manager/whatever with an Integer[] array. I would like to note that if I hide the caching behind, the client still does not care about this:

```
public final class Cash implements ICash {
    private static final ICash noCash = new Cash(0);
...
    public static ICash cash(int cents) {
        return cents != 0 ? new Cash(cents) : noCash;
    }
...
}
```

1 • Reply • Share ›

David Raab ➚ Kata Tunix • 4 days ago

> First, your code still violate the "Dependency inversion principle"

If you think DIP is a good thing to follow you will **always** make all constructors private and create static instance methods instead. And first. It is not possible to avoid "new" completely. Somewhere you have to explicitly create a concrete instance. But the idea of DIP is that the user should not decides which concrete class should be instantiated, the code itself decides to pick the concrete/right classes. And the second point about DIP is, that your code relies on Interfaces instead of concrete classes.

So the rules are:
1) User does not call new (decide which concrete object to use), the code decides which concrete object should be used
2) Everything depends on Interfaces instead of concrete implementations

And his static "cash" method exactly does that. And just from a functional-programming standpoint. In a functional world we name such functions "create". I would also suggest to have a "Cash.Create()" method instead of "Cash.cash()". It makes code more predictable and better understandable to have such standards. So this "Create" method should decides which concrete

implementation should be used. In his implementation it will always be an object of the "Cash" class, but he is still free to always change it later at any point without that any code breaks.

He always can introduce different Cash classes for example. For example a bad thing in both of your code is the "Currency" string. Because that string is really used for different states, and depending on the state the behaviour can change. It can make sense for example to create different Cash classes for the different currencys, so for example create a CashUSD class and a CashEUR class for example. When he later decides to do it. Nothing has to be changed. He still can use his **Cash.Create()** method, but depending on the input it can either return a Cash object, a CashUSD object, a CashEUR, or other different implementations.

In fact to fullfill the second rule of DIP you should not return concrete instances, you should return interfaces. So the Create method should change to something like that (C#)

```
public class Cash {
    // Private Constructor
    private Cash(int cents) { ... }

    // static Constructor
    static ICash Create(int cents, string currency) {
    // Based on currency it can return differen implementations
    switch ( currency ) {
        case "USD":
            return new CashUSD(cents);
        case "EUR":
            return new CashEUR(cents);
       }
    }
}
```

And now this code fulfills also the second rule. Instead of returning a concrete class it now returns an interface. And the function can decide which concrete object should be used. And just note, just having "new" in the code doesn't mean you break DIP. At some level you have to call new. The point is like i said not to avoid "new". the point of DIP is that a user doesn't decide the concrete implementation, the code in charge decides it. And the above code does exactly this.

Now his implementation didn't do that, but by making the constructor private and always forcing to go over a static constructor method, you always can change your code without breaking anything. He can swap his implementation with mine

breaking anything. He can swap his implementation with mine, without that any code based on Cash breaks.

And why will you always do it like that if you want to follow DIP strictly? Because you only can fulfill both rules in this way. A constructor with "new" cannot return just an Interface. A constructor always have to return a concrete class. The constructor is just there for the initialization of the concrete class. Not to decide which concrete class should be picked.

So if you really believe that everything should depend just on interfaces. You always make the constructor private, and just provide a static method for creation.

> Next, you said "declare a more abstract type rather than a concrete type letting you bind to interfaces or abstract classes only". Did you mean you have a derived class e.g. Cash2? And in the Cash.cash() method you can return an instance of Cash2? If I'm wrong, could you please show code again for this? Pseudo code is okay.

He meant that you just can return an Interface and you can decide which implementation to use. That is the whole point of DIP btw. Your code don't rely on concrete classes, it relies on abstract types (interfaces).

But there are also other possibilities. For example you always want valid objects. What happens if at the construction level you see something is invalid? In a normal constructor the only chance is to throw Exceptions. But with a static constructor like this you also can return for example an "Optional<icash>" or IList<icash> for example. Also this is not possible with a normal constructor. Because your constructor now even returns a wrapped thing of what you wanted.

> Finally, "implement object cache and not necessarily always create new instances". I agree with this benefit. But the caching job should be assigned to another class so we don't violate the "Single responsibility principle".

It makes sense to separate it, so you don't have to re-implement caching over and over again. But even if you decide to put it in the static constructor it wouldn't break the SRP because the objects itself don't have any additional functionality. The objects itself don't change at all. They still do the one thing they do.

1 ^ | v • Reply • Share ›

Kata Tunix ➚ David Raab • 4 days ago

I appreciate your knowledge about DIP, they are true in general.

But I disagree about the static methods. When you are still depending on something concrete (static methods), you violate DIP. In our situation, you will get rid of this only by using Factory Method or Abstract Factory pattern. I mean, without these two patterns, your factory is still a concrete factory.

Now let's consider static methods are okay, your C# code is what I expected, the Create() can return an instance of CashUSD in the type ICash. That is what I meant in my last comment: "return an instance of Cash2 in the type Cash". But wait, could you tell me the relationship between Cash and ICash, CashUSD? I know ICash is an interface and CashUSD, CashEUR implement ICash. But I don't get the idea of the Cash class because you declared it as: **public class Cash {** If Cash is merely a factory, why does it still have **private Cash(int cents) { ... }**

[Updated] From the newest code of Lyubomyr: **public final class Cash implements ICash {** I can see Cash is also a derived class of ICash, just like CashUSD, CashEUR, Smartphone. Well, Cash is a very special ICash, huh? It has knowledge about all of its siblings, moreover, it can CREATE its siblings. If you guys think this is good, then ... :D

About the SRP, you're right, of course. But I could not recognize that you agree or disagree with me :) Even if you disagree with me, debating on this is unnecessary.

Thanks for interesting debate, guys.

David Raab ➚ Kata Tunix • 4 days ago

> But I disagree about the static methods. When you are still depending on something concrete (static methods), you violate DIP.

There is no difference if you either use **Cash.Create()** or put the **Create** function in its own Factory class and call **CashFactory.Create()**

```
public class CashFactory {
    public static ICash Create() { ... }
}
```

It is exactly the same. You either depend on **Cash** or on **CashFactory**.

And even if you decide that CashFactory is a class with a

"CreateCash" method instead of a static method

see more

• Reply • Share ›

Lyubomyr Shaydariv ➚ Kata Tunix • 4 days ago

Sorry for the "Smartphone" typo: it was intended to demonstrate with `Smartphone implements IDevice`, but you caught it before I noticed my typo there.

• Reply • Share ›

Lyubomyr Shaydariv ➚ David Raab • 4 days ago

Just a little note why I prefer the static method to be called `cash`. I'm just a fan of static imports in Java, but methods named `create` can clash. For example:

```
System.out.println(cash(34) + " / " + amount(12));
```

rather than

```
System.out.println(Cash.create(34) + " / " + Amount.create(12));
```

As far as I know, C# now supports static imports as well (but as far as I remember, called in another way). But I avoid static imports for methods that are called `newInstance()`, `create()`, because these names are ambiguous.

1 • Reply • Share ›

David Raab ➚ Lyubomyr Shaydariv • 4 days ago

Ah, yes i understand the design decision. Yes C# will support static imports with C# 6. Actually the idea to name it "create" comes from functional languages like F#/Ocaml. These languages already support static imports, but it is still prefered to use the full name.

So we still use "Cash.create" instead of statically importing the functions. But sure one point of why it is this way is that every function is really just a static method and you would just import too much functions if someone would import "Cash". If someone would come to the idea to import "List" he would have hundreds of functions that can clash with other types. So instead of naming functions like **list_map** to be unique even after importing we just use **List.map** without importing anything.

And anyway, if you just want shortcuts you can just bind functions to another local symbol. For example if you don't want to write **Cash.create** all the time you can just do

```
let cash = Cash.create
```

```
let x = cash 15 "USD"
let y = cash 25 "EUR"
```

without importing every function from the "Cash" module. So also with static imports it is common to create "create" functions for every type and don't do "static imports", at least in functional languages. That is also the reason why i would still prefer "Create" even in C#. You also can do such local naming in C#, and since Java 8 it should also work there.

In C# that would look like that

```
Func<int, ICash> cash = Cash.Create;
var x = cash(10);
var y = cash(50);
```

It is not so nice, because you can't use "var" here and use type-inference. You have to explicitly use the Func definition. That should also work with Java 8 btw.

• Reply • Share ›

Kata Tunix • 5 days ago

For sure, as developers, we have faced this situation frequently. Your approach is called "telescoping constructors" and it is good enough for a short list of parameters. In case the list is long (e.g. 5, 6 ... parameters) and cannot be "telescoped", Java Builder Pattern is a good choice but it introduces more code and one more class.

I would like to know your own idea about a long list of parameters passed into constructors. And, is there any general way to avoid such a long list?

4 • Reply • Share ›

Yegor Bugayenko author ↗ Kata Tunix • 3 days ago

Marcos is right, if you have too many parameters, there is something wrong with your class. It is simply too big. Break it down into pieces. Builder Pattern is not solving the problem, but is just hiding it. I would not recommend to use it. Any class that has over five arguments of a constructor is definitely a bad design. No exceptions.

4 • Reply • Share ›

Kata Tunix ↗ Yegor Bugayenko • 2 days ago

I agree with you guys in general. There might be an exception, for example, a view object containing five textboxes is not complex. I want to pass five strings into its constructor to fill into those textboxes.

What do you think?

• Reply • Share ›

David Raab ↗ Kata Tunix • 13 hours ago

Usually it is better to work with the MVVM idea instead of MVC. In MVC you often pull your data. But a better idea is if you have a pushing system through events and data binding. That is often called MVVM then. If you use that you also usually don't have long constructors. The controller is replaced through an event/data binding version that is then called "ViewModel"

Usually the view is just as dump as possible. It only contains the text boxes or other gui elements a user can act on, but besides that, nothing else, not even formating or other stuff. Then instead of a controller that pulls for the changes you use create a ViewModel class.

The ViewModel handles all communication between the model and the view. Usually the model has an event system. So the ViewModel can register on the different events when they appear. If an event appears from the model, the ViewModel automatically gets all data, it does I10N, L18N and all other formatting what is needed

see more

1 • Reply • Share ›

Kata Tunix ➚ David Raab • 10 hours ago

Yes, MVVM is an extreme case of applying the Observer Pattern:

V <==========> VM ==========> M

- VM observes M: hey M, if you have something new, feel free to let me know, un-formatted data are okay.
- V observes VM: hey VM, if you have something beautiful, please tell me so I can render them to users.
- VM observes V: hey V, if users fire something on you, please notify me.

I can see writing the code to register all these observations would be a nightmare. There should be some techniques (e.g. markup, notation ...) to help to simplify this, right?

• Reply • Share ›

David Raab ➚ Kata Tunix • 10 hours ago

> I can see writing the code to register all these observations can be a nightmare. There should be some techniques (e.g. markup, notation ...) to help to simplify this, right?

Depends on the language. I'm not a Java guy, but C# already has a built-in "event" type since C# 3. So it is pretty easy to define an event in C#. Just a single line of code, the same goes for

registering/deregistering that is also just a single line of code. And even without it you could implement it easy yourself because C# support Delegates/Lambdas since C# 3.

Observables on the other hand are even more powerful. You have a library called "Rx" that works with event streams that can be combined. As far as i knew that library was also ported to Java. But usually that is not what you will use for something simple like data binding or simple events. But if you find it simpler. Usually in Rx you just can create a Subject where you can connect/publish to.

But usually if the language supports some low-level functional programing you also can create a list of functions where you can (de)register. Usually should not be too hard to implement that yourself with few lines of code and should be possible since Java 8.

• Reply • Share ›

Yegor Bugayenko author ➚ Kata Tunix • 2 days ago

I think that a "view object" is an anti-pattern in the first place :) Planning to write about it soon. So, yes, five strings in a constructor is not a good design.

2 • Reply • Share ›

Kata Tunix ➚ Yegor Bugayenko • a day ago

Yegor, I'm thinking of one bad thing about a view object is that it encourages other objects to expose their internal states, through getters.

• Reply • Share ›

Yegor Bugayenko author ➚ Kata Tunix • a day ago

Yes, that is the main problem. View objects are pure data holders that do nothing except holding the data and exposing them through getters. This is not what OOP is about.

• Reply • Share ›

Kata Tunix ➚ Yegor Bugayenko • a day ago

Hmm ... view objects still have their own jobs: rendering/drawing GUI. Yes, they still expose their internal state but for human :)

• Reply • Share ›

Yegor Bugayenko author ➚ Kata Tunix • a day ago

Than we're talking about different objects :)

• Reply • Share ›

Kata Tunix ➚ Yegor Bugayenko • a day ago

Sorry, I should say "GUI objects" (I mentioned about five textboxes so I didn't think you misunderstood).

But still waiting for a post about your view objects.

• Reply • Share ›

David Raab ➚ Kata Tunix • a day ago

Nah, just apply "Yegor think". Remove the "get" from the methods and magically you don't have getters anymore... And for the View thing. Just create gigantic god classes that has a method for every possible way a thing can be viewed, that is what he thinks is right.

But just for the discussion sake. Can you explain why you think that exposing the internal state is bad?

• Reply • Share ›

Kata Tunix ➚ David Raab • a day ago

Maybe Yegor hates typing "get" :)

There are a lot of talks about getter methods on internet. I'm not sure about the definition of a getter method: yes, it is a method that returns something (i.e. not void), but is that "something" internal state i.e. a private property?

From my point of view, getter/setter or whatever methods are okay as long as they belong to the contract of the class (i.e. an interface). If the contract declares a getter method, then what's wrong with the method? The contract itself doesn't care the returned value is internal state or not.

In case the class doesn't have any contract (is that bad?). If it exposes internal state and thus violates encapsulation, clearly, it's bad.

I guess you're thinking that exposing internal state is not bad when the internal state is stable and thus doesn't violate encapsulation. So, the actual point of encapsulation is: "depend on stable things, not volatile things, of a class". Internal states are often (yes, I say "often") volatile, and contracts (interfaces) are often stable, right?

• Reply • Share ›

David Raab ➚ Kata Tunix • a day ago

Could be that he hates typing "get". But the important thing is that removing "get" makes nothing "not a getter". That you write all your getters/setters with "get" and "set" in front is just a language convention, and not every language have such a convention. Whether in Perl nor in C# you ever write a Getter/Setter with "get/set". You would always just write.

Perl: **$dog->weight()**
C#: **dog.Weight**

while in Java you would write:

Java: **dog.getWeight>()**

But if you read his Getters/Setters Evil post. http://www.yegor256.com/2014/0... Yegor clearly thinks that just writing "weight()" instead of "getWeight()" doesn't make it an getter anymore.

So the answer is. Hey in "Yegor think". Just do a string-replace on your source-code and delete the "get" characters from every method and now none of your objects expose any internal state anymore. And yes, this is sarcasm.

> In case the class doesn't have any contract (is that bad?). If it exposes internal state and thus violates encapsulation, clearly, it's bad.

At first, no it is not always bad. And second that is not an explanation at all. You do not argue by just saying that something is bad because something else is bad. If you argue that it is bad because it would break encapsulation then explain why breaking encapsulation is bad.

And to just give a short explanation. OO has encapsulation because the very idea behind OO was to have mutable data and hide those mutable data in a class so only specific methods can access and change the mutable data. That idea is important, because if mutable data are open to everyone, it can happen that mutable data are changed from the outside to an invalid state and thus the objects can break from the outside. So the idea of encapsulation was born and every mutating data is hidden in a class. Breaking encapsulation is bad only if you have mutable data, because it allows your objects to change at any time even to some invalid state.

Functional languages on the other hand always prefers immutable data and through this eliminate all state at all. And because you only have immutable data you also don't need encapsulation. The problem that someone else can break something from the outside doesn't exists at all. Because nobody is able to change something in the first place.

So, when you also start applying functional programming to OO and you start using immutable objects. Then you also don't need encapsulation anymore. Breaking encapsulation is only bad on mutable objects.

So the answer is, if you have immutable objects. Exposing state is

So the answer is: if you have immutable objects. Exposing state is completely valid and breaking encapsulation did not really exists. If you have mutable objects then it is the opposite.

But even that is not something that is always right. You are saying that you develope games? The most common way how to develope games is through some sort of "Entity Component System". That