



Community

 yegor256.com

35 Comments · Created 5 months ago



## Immutable Objects Are Not Dumb

After a few recent posts about immutability, including "Objects Should Be Immutable" and "How an Immutable Object Can Have State and Behavior?", I was surprised by the number of comments saying that I badly misunderstood t...

[\(yegor256.com\)](#)

35 Comments

 Recommend 1 Share

Sort by Newest ▾



Join the discussion...

**Leon** · 3 months ago

While your Page class is technically immutable, it does it by violating OO principles. When you call load(), you are returning data that is not part of the internal state of the Page. To use your own example, it's as if you call getBallDistance() on a Dog, and it returns different results every time because the ball moves while the dog stays put.

I/O is often impractical to represent with immutable classes. You'd have to copy and cache the data (once per immutable object), and keep it until all references were released. This would obviously eat up lots of memory/storage.

 |  · Reply · Share >**David Raab** → Leon · 23 days ago

Page is also technically not an immutable class, because the load() always can return something different it is just an mutable class. That he doesn't save the loaded content and directly returns it, and thus has no changing field, doesn't make it immutable. Immutability means it never changes after creation and always return

immutable. Immutability means it never changes after creation, and always return the same thing. And his Page class don't do stuff like that.

It is technically not possible to have something immutable that also does I/O. I/O is always mutable. The only thing what he could do is load something and then save the result in an immutable class. So he could just have a Page class, but that Page class already has the fetched data, and that data will never reload. So the Page class will only represent data at a specific point in time.

And if he provides a "reload" method. Then this method has to return a completely new Page object. Such an implementation would return immutable objects.

But just a "load" method that just does I/O at the time of calling and returns the data at the call is just a mutable class. Because on every call it can return something different. That he doesn't change any state in his object doesn't make anything immutable.

^ | v • Reply • Share ›



**Fabrício Cabral** → David Raab • 22 days ago

David, you said "Immutability means it never changes after creation, and always return the same thing", but let's supposed a random class, that generates a different integer every time you call it, like this:

```
Random r = new Random();  
r.nextInt() // returns 2  
r.nextInt() // return -2323
```

So, i think possible to do a Random classe immutable and this class does not always returns the same thing.

Does it make sense?

^ | v • Edit • Reply • Share ›



**David Raab** → Fabrício Cabral • 22 days ago

A Random class like you showed is mutable because every time you call nextInt() it returns something new. But there is a possibility to make it immutable. You basically have to split it in two methods/attributes. A "Current" attribute returns you the current "random" number. And the "Next" method advanced to the new object with a new state that has a new current value.

So your Random class would for example look like

```
var rng = new Random(123456);  
var r1 = rng.Current; // 2  
var r2 = rng.Current; // 2  
var r3 = rng.Next().Current // -2323  
var r4 = rng.Current; // 2  
var rng2 = rng.Next();  
var r5 = rng2.Current; // -2323
```

```
var r6 = rng2.Next().Current; // 456  
var r7 = rng.Next().Current; // -2323
```

So you always have to advance with Next() to the next state. And get the Current random number with "Current". While i really like immutability i don't really could say if that approach is a good one. The whole purpose of an RNG is that it always returns something completely new. And from a RNG you also usually expect that it changes. So a mutable object/state is really wanted here. I don't see much benefit of rewriting it to be immutable.

I also just want to point out, that this immutability only works with pseudo random number generators. It is not possible to make a true RNG immutable. That would even destroy the whole purpose of a true RNG and that would be just a broken implementation.

^ | v · Reply · Share ›



**Noëlle** → David Raab · 4 days ago

David, perhaps I've misunderstood, but I think you have it backward. Fabrício's object seems to me to be immutable; yes, the value it returns is different every time, but the state of the object never changes, just the random-number seed. This is like a File object that refers to a file that is being changed from the outside. The File object's state doesn't change, but the contents of the file it refers to do.

Meanwhile your class DOES change state: rng.Current is part of the state of the object. When you call rng.Next() you are modifying the state of the object to give rng.Current a new value.

Have I misunderstood?

^ | v · Reply · Share ›



**David Raab** → Noëlle · 4 days ago

Fabrício's object seems to me to be immutable; yes, the value it returns is different every time, but the state of the object never changes, just the random-number seed.

If the random-number seed changes, then it is not immutable anymore. Immutable means nothing changes at all.

But being immutable means a lot more than just looking at the code and looking if some internal attribute changes or not. In fact you can decide if something is immutable by just using it. Something that is immutable **always** return the same thing because its state never changes. If you have successive calls to a method on the same object it always have to return the same thing, forever. Only if that is **always** fulfilled then you have something that is really immutable.

That also means. The **Page** class from Yegor is not an immutable class. Because on every call to the **load** method it can return something different. The content of a site can always change/mutate at any point, and the **load** method will directly resemble the mutated state. It even can happen for example that a successive call to load throws an exception, because the loading failed.

The Page class doesn't mutate any internal attributes, but it still behaves like a mutable class. And because of that, it is not considered as an immutable class. The same goes for the File class. A File class can represent a specific file on the file system. But if the File class has some "readAll" method that just loads data when you call it, it is not a immutable class anymore. Saying that File is an immutable class is just plain wrong. Because every call to a method can return something different.

If the thing he returns is saved as an attribute or not is not important at that point. Just because you don't decide to not save the mutated data in its own attribute and directly return the mutated state doesn't make it immutable. You still have an mutable class, you just don't save the mutated state in attributes and calculate everything on-the-fly.

If something like that would be immutable nothing makes sense anymore. It would mean that the same code can be immutable or not be immutable depending on the concrete implementation of something. It also means that any method always can return something different. And that is clearly not the definition of being immutable.

An Immutable Page class for example has to work like this

```
// Creates the Page and also fetches the Content
var site = Page(new URI("http://slashdot.com"));

// site.Content have to always return the same Content as Long as
Console.WriteLine(site.Content);

// Now a new Page is created that can represent the new changed Si
var newSite = site.Reload();

// Still has to print the same content
Console.WriteLine(site.Content);

// Can now print the new changed content, but it is also a new obj
Console.WriteLine(newSite.Content);
```

So from an immutable object we expect that **site.Content** always

have to return the same thing. What would be clearly wrong and not be immutable would be when `Content` always can return something new. Then it is clear that under the hood "`Content`" somehow mutate and it reflects a mutated state. But if you believe the things Yegor thinks was immutable means, it makes no sense at all. Then it also could be that calling **`site.Content`** and it always return something different can be immutable. Just depending on the implementation. if you implement it like this

```
// ... Property in the Page class
public string Content {
    get { return HTTPRequest(this.url).fetch() }
}
```

then it would suddenly always fetch the current Site Content whenever you access the "`Content`" Attribute. but it would not mutate anything. But still it would mean that sometimes **`site.Content`** that can return something new on every call would still be immutable, what makes no sense at all. It would even mean that sometimes **`site.Content`** that always return something new can sometimes not be immutable if it would additional change something. So a

```
// ... Property in the Page class
public string Content {
    get { this.counter++; return HTTPRequest(this.url).fetch() }
}
```

would suddenly not be immutable anymore, because it additionally change a counter. And you would not be able to know if something is immutable until you looked at how it is implemented. But that makes no sense at all. Both cases are "mutable" classes and not "immutable". Because they always return something new. The fact that you don't decide to fetch the content just once and return it directly doesn't make anything immutable. If that would be so easy, then basically everything could be easily changed to an immutable object if you just return every mutable data directly.

But with that idea you don't solve anything. The point of immutability is that a reference to something can never change its state, so you easily can share it. But a field/method that always return something different on every call is not what you can share, because it mutates on every call.

Meanwhile your class DOES change state: `rng.Current` is part of the state of the object. When you call `rng.Next()` you are modifying the state of the object to give `rng.Current` a new value.

Look again at my code. What you explained is not what i have written. The example that i showed would not change anything at all.

The value of **rng.Current** never changes. Even after calling **rng.Next()** **rng.Current** will still return the same value. **rng.Next()** returns a new object, and this new object has a new state. Nothing is mutated in the code i provided.

You see that exactly here.

```
var r2 = rng.Current; // 2
var r3 = rng.Next().Current // -2323
var r4 = rng.Current; // 2
```

here "r2" and "r4" are still 2 because that is what **rng.Current** is. Even the call with **Next()** on **rng** didn't change the **Current** attribute at all. "r3" is here "-2323" because **Next()** returned a new immutable object. And this completely new random object had the new state "-2323". But the current state of **rng** was never changed at all.

^ | v · Reply · Share ›



**Fabrício Cabral** → David Raab · 22 days ago

David,

I would to thank you for your attention. I've been learning a lot with you, Yegor and the other people here.

I see your point, but I've a lot of doubt about what is a immutable object, what it does and its behaviour.

For example, suppose this example: you've a Person, and this Person has a name and friends. This friends are too Persons. So, you could do (Java code):

```
public final class Person {
    private final String name;
    private List<person> friends;
    public Person(final String name, Person... friends) {
        this.name = name;
        this.friends = Arrays.asList(values);
    }
    public String name() {
        return name;
    }
    public List<person> friends() {
        return Collections.unmodifiableList(friends);
    }
}
```

So far, so good. I created (I hope) an immutable class that represents my domain. The problem is every day, the person does new friendships and undoes old friendships. So, we should permit add and remove persons in the friends list. But if I modify my class to permit it, the class will be a

- 1) How make a immutable class to permit add and remove friends?
- 2) When i should make a mutable or immutable class? How I should think "Humm... this class has these behaviour, so it must be mutable"?
- 3) What do you recommend read to deep about this subject?

Thanks for your attention!

^ | v • Edit • Reply • Share ›



**David Raab** → Fabrício Cabral • 21 days ago

The problem you are faced, i faced that problem also once ago. I think there are multiple reason why you face your problem. At first, "Immutability" is really something that comes from the functional world not from the OO world. Immutability is a core concept of functional languages, and languages that was invented in the 1970 already have immutability as a primary design decision. While i don't know any OO language that was designed around immutability. Other than Yegor, i also don't think that OO thinking leads to immutability. If it would lead to immutability then even C++ would be immutable by its core. Or even Java that was invented in the early 1990.

I think that this is important, because if someone doesn't properly explain what immutability is about, you can get it easily wrong with OO, because object-thinking leads more to a mutable design. So it gets hard to understand what immutability means in an OO world. So let my explain what immutability means in a functional world instead, and then i go back to the OO world and explain what you have to do there to achieve immutability.

At first. Some difference. The OO world believes of objects. An object has some behavior. But in order to have some behavior it also need to wrap some data. So an object is the idea of data + behavior that work on it. The data are often encapsulated and hidden. So a user just sees the behavior, and the object itself can do its behavior and work or change the data that are encapsulated. An object also forbids other code to change the internal data, because it could lead that the object end in an invalid state. That is also why you have encapsulation. Instead of forbidding changes at all, you try to hide critical data, so the only one that can change critical data is the object itself. So the object can provide some behaviour to others, but the object itself always makes sure that the object itself is valid and never changes to some illegal state.

But in a functional world where immutability has its origin, you believe something else. Instead of trying to hide state with encapsulation you believe that data/state and behavior should be completely separated



from each other. For example look at your Person class. You have a Person it contains data "name" and "friends" and you also want some behavior, like "Adding a friend". In a functional world instead you just have data to begin with. You also can create a type "Person" that also have a name and a list of friends. But it doesn't contain any behavior. The idea of a functional language is, that every data should be immutable. That means, once it is defined, it cannot change anymore. That idea comes from math. In fact most of functional programming ideas comes from the math space. For example when you define in math "x = 5" then you are saying that x equals to 5. In math it doesn't make sense that you later can write "x = 6". Because it makes no sense. Either x is equal to 5 or 6, but it cannot be 5 and 6 at the same time. And that is also how you have to read data in general. You are defining symbols instead of variables, and you explicitly define what each symbol is. And once you have defined it, you cannot change it.

But now, you still want some behavior. So how do you have some behavior? Well, you can write functions. So here is the idea. A function always have some input and some output. Because every data is immutable it cannot change the input. But here is the idea. Instead of changing something you can return something completely new as the output. So the idea is simple. A function have some input, and it generates you some output. And sure the output it generates can base on the input. But the important thing, it cannot change the input. So just for example lets say you have a List of string. (I'm using F# here)

```
let x = ["Jeff"; "Frank"]
```

In F# here we define a list of strings. That List of strings is bound to the symbol "x". So every time we use "x" we really just use that list. Technically you could even substitute the x with the list in-place. Because that x is not a variable. It just represents the list. Now neither the x nor the list itself can change. But lets say we want to add something to the list, how do we do it? Well because the list cannot change we can't directly add something to the list. But we can create a new list with three elements that has the same two strings. Now in F# there exists already multiple defined functions that does that, i'm using the List.append function. This functions wants two lists and it creates a new list from it. So for example the code can look like that

```
let x = ["Jeff"; "Frank"]
let y = ["David"]
let z = List.append x y

printfn "%A" x // Prints: ["Jeff", "Frank"]
```



```
println "%A" y // Prints: ["David"]
println "%A" z // Prints: ["Jeff", "Frank", "David"]
```

So as you now can see. *x* and *y* didn't change at all. They are still the list we defined. But our `List.append` function took two Lists, and generated a new List from it, and that new List is now "*z*" that has our three elements.

So in a (pure) functional world nothing can't change. But we have functions. And functions can take some input, and generates some new output from it. For example your `Person` class could look like that in F#

```
type Person = {
    name:    string;
    friends: Person list;
}

let x = { name = "David"; friends = [] }
let y = { name = "Jeff";  friends = [] }

// A Function that takes two Persons and returns a new Person.
// The new Person has the second person added as a friend
let add_friend person friend = {
    name    = person.name;
    friends = (List.append person.friends [friend]);
}

let z = add_friend x y

println "%A" x // Prints: {name = "David"; friends = []}
println "%A" y // Prints: {name = "Jeff";  friends = []}
println "%A" z // Prints: {name = "David"; friends = [{name = "Jef
```

So in F# i define a "Person" Type, and i create two Persons with *x* and *y*. When i want "Jeff" as a friend to David, then i have to create a new Person. As you can see in the end i have three Persons *x*, *y* and *z*. And *z* is the result of the operation i wanted. So a function took *x* and *y* as its input and returned *z* as it output. And as you can see, *x* and *y* did not changed in the operation, because they were immutable. You can't change *x* or *y*. The only way to handle your change is to create something new.

And now back to OO. That idea is basically what you also have to do in OO. The problem is, because of the way how OO works it isn't so clear what a function is and what the input and output is, because everything is mixed together into one object. And usually a method works directly on or with itself. But that is also the answer.

In an OO world you have to think the following way. An object should be immutable and no field itself should change. Any method that you create automatically has its own object itself as an input. Your method can have additional inputs, but in an immutable world you don't change yourself or some of the inputs. Your method just returns a new object.

For example an Immutable version of the F# code could look like that (in C#):

```
using System;
using System.Collections.Immutable;
using System.Linq;

public class Person {
    public string Name { get; private set; }
    public ImmutableList<Person> Friends { get; private set; }
    public Person(string name, ImmutableList<Person> friends) {
        this.Name = name;
        this.Friends = friends;
    }

    public Person AddFriend(Person friend) {
        return new Person(
            name: this.Name,
            friends: this.Friends.Add(friend)
        );
    }

    public override string ToString() {
        return string.Format("{Name = {0}; Friends = {1}}",
            this.Name,
            "[" + string.Join("; ", this.Friends.Select(x => x.ToString()))
        );
    }
}

public class Application {
    static void Main(string[] args) {
        var x = new Person("David", ImmutableList<Person>.Empty);
        var y = new Person("Jeff", ImmutableList<Person>.Empty);
        var z = x.AddFriend(y);

        Console.WriteLine(x.ToString());
        Console.WriteLine(y.ToString());
        Console.WriteLine(z.ToString());
    }
}
```



Now this code prints the exact same things as the F# code does.

And that is also what you have to write in Java. The problem is. In Java or C# that isn't so obvious, because normally nothing in C#/Java is immutable by default. Also a normal List in C# is not Immutable also not in Java. Notice here that i used a special "System.Collections.Immutable.ImmutableList" here from a library for C#. So an "Add" method on that ImmutableList has the same behavior as in F# where a new List is returned, instead of modifying the current List.

When you use Java, you also have to use special Immutable List classes. I don't know if Java has some by default.

And that is the reason why i think immutability is harder to understand in OO. Because normally nothing is immutable by default. Everything in the language was usually designed to be mutable at default. And because you don't have an simple input -> output system and you don't separate data and behavior, it is often not so obvious how you achieve immutability.

So what you really have to understand immutability is to separate data and behavior. Data is immutable and you have behavior that can work on data, but instead of modifying data it always creates new data. And i think it is hard to understand for an OO programmer at first because he learned from the beginning to not separate data and behaviour and put everything into one class.

So as a final note. Immutability isn't strictly about forbidding changes. It is more about how to handle changes. In a mutable world you change something directly in-place. But in an immutable world you apply your changes that you create something completely new.

To the question when should something immutable or mutable. I think "immutable by default" is a good approach. So the most thinks you create should be immutable. But also think of it that languages like Java are not invented to be immutable by default, so sometimes at some places it could be that immutability have the effect that some places are harder. For example the AddFriend method in the C# version above. Instead of just adding a Person to the List (normally a one-liner) you have to create a full new object. While at some places it can look harder, the benefit is that at some other places it is easier.

The point of immutability is that you always knew something can't silently change. For example if i have code like this.



In a mutable world it now could be that "person" changed, without that you knew it. But if your person is immutable, you knew it absolutely cannot change. It gives you the ability later on to reason about your code what it does and what it not does. And while some methods can be harder because you always return completely new objects, some methods get easier. For example in an immutable object you only have to check the object validity at construction time. Because an object cannot change later on, you don't need to re-validate thing whenever something changes. A simple example would be if you had a mutable DateTime object. Lets assume it would be mutable, and you had the following code

```
var date = new DateTime(2015, 12, 31) // Sylvester
```

now lets assume that thing would be mutable and you later change the month to Februar.

```
date.month(2); // Whoops Error!
```

The problem here is. You can't just validate the month. The month 2 would be okay. But now the day "31" would be invalid. Because Februar 31 doesn't exists. So everytime you change the month you would need to re-validate also the day.

And it even goes one. The same could happen if you change the year. because if you change the year, you also have to check if the month was not Februar before. Because the date before could be the leap day. And with changing the year the new date is not a leap year. So every time you change the year, you also have to check the month and the day.

So the thing is, a mutable design can end up that you have to do a lot more in your methods and you have to re-validate a lot of things over again. So an immutable design can overall reduce the complexity, even if in some cases it can get harder.

Does that mean you should always have everything immutable? No. I think a black and white view is never good. Even most functional languages that have immutability be default supports mutable data structures and support mutability. But you have to explicitly activate it. The point is simple. Sometimes it could be that trying to dorce something to be immutable gets really really hard. Or readability or how you use something suffers a lot.

I think the Random class is a good example here. In an immutable Random class you have to advance forward with "Next()" explicitly and then call Current on it to get a new random number. But when do you not want that? I mean the purpose of a random class is to always give you a new random number. So creating an immutable Random

object doesn't make much sense, because you don't really want immutability here to begin with. An immutable random class is just harder to use, not easier.

So when is immutability a wrong approach? When it just makes everything harder and you don't really get a benefit out of it. If you later has to code more, code is harder to read, harder to understand then you just did the wrong thing.

I only can say. Absolute views are always bad. The idea that everything should be immutable, everything should be OO, everything should be functional, everything should be ... is always wrong. If you can write something more simpler and that "more simpler" has no disadvantages and is easier to understand for a reader. Then it is also very likely that it is the correct the solution to pick (just "very likely" because sometimes we think something has no disadvantages but it still has some disadvantages but we didn't see them).

If you want to read more about immutability, i would even suggest to look into a functional language and learn how a functional language work, because that also helps you in an OO language. And the point is a functional language is designed around immutability a typical OO language Java is not. So it is also easier to understand it. And another thing is that immutability in an OO language is still a "new thing", you have to find articles about it, and not everybody uses it. It is still normal for the average OO programmer to create mutable classes. So i think good information are rare. But because immutability is normal in functional languages for over 40 years now, you found a lot of things there and everything is designed around it.

Some examples:

<http://fsharpforfunandprofit.c...>

<https://ocaml.org/learn/tutori...>

<http://typeocaml.com/2015/01/2...>

<http://typeocaml.com/2015/01/0...>

1 ^ | v · Reply · Share ›



**Fabrício Cabral** → David Raab · 18 days ago

I'd like to thank you for this excelent and well detailed explanation. Congratulations!

^ | v · Edit · Reply · Share ›



**David Raab** → Fabrício Cabral · 15 days ago

Thanks, i'm glad it helped you.

^ | v · Reply · Share ›



**Leon** → Leon · 3 months ago



To clarify, in this example, a better solution would be to have a `PageData` class, and provide a method on it to load the data with a given `Page` instance:

```
PageData load(Page page) {}
```

This would create a new `PageData` object, holding the loaded page in a char stream, and that object('s class) could be made immutable too.

The solution for the `Dog` example would involve a separate `Distance` class with a method like:

```
Distance measure(Positionable p1, Positionable p2) {}
```

With `Positionable` an interface with method `Position position()` {} that both `Dog` and `Ball` implement.

So `Distance.measure(ball, dog)` will call `measure()` on both and calculate the distance, and return an immutable `Distance` object.

`Distance` has other properties like `value()` and `units()` that return for example 12 and INCHES.

^ | v · Reply · Share ›



**Lambda Pool** · 4 months ago

Rich Hickey is right, we aren't there yet.

People just sux at these so simple concepts.

Why insist using mutable when they should be using immutable only for stupid insistence and lack of adaption for new concepts?

In that case there are really old systems running based on immutable state and they are running more than 40 years at Ericsson. Please speak one minute with a Erlang programmer and ask him about keeping the state of objects, probably he'll laugh on your face...

Don't be stupid, try using Akka with mutable objects and see what happens... PLEASE TRY IT!

^ | v · Reply · Share ›



**Hendrik Belitz** · 5 months ago

I don't think the whole problem of "getting it" (with it being the concept of immutability) is that objects are not data structures (which, besides, from my point of view is wrong), but the fact that they don't recognize what an object represents.

`File` for example is a reference to a file on disk. It is immutable because this reference will not change (with the single exception that it will become invalid if you delete the file)). It is explicitly not a reference to the contents of the file (which is not part of `File`'s state), which is freely allowed to change. I don't have any attributes referencing the contents. If I would need such an entity, I would introduce a `FileContents` class, preferably implementing two interfaces (one for read-only (and therefore immutable) access, and a mutable one for write access).

The same applies to the `URI` class. This class does not represent a website. It represents a resource locator that I can e.g. use to retrieve a website, preferably by instantiating a `Website` class (which again, may be mutable or not, depending on its usage scenario).

^ | v · Reply · Share ›



**Andre** · 5 months ago

**Andre** · 5 months ago

As always, both immutability and mutability have costs and benefits.

Immutable objects are easy to handle, can be freely shared among threads without fear of data races (as long as they are safely published, read at [shipilev.net](http://shipilev.net)), but they require a new instance for each change, which means lots of allocation (pooled flyweights, e.g. Integers in the 0.255 range, are an exception, but they're not very common, because many problems need a bigger state space than can be handled with flyweights). This is a no-go in high performance settings.

Even so, parts of the application that change seldom if at all, or that run while the hot code doesn't need the caches can use immutable objects where it improves readability. Note that sometimes using mutable objects may convey intent more clearly. So let's not put immutability on a pedestal. If I have to choose immutability of my objects or maintainability of my code, I'll choose the latter. If I need to reach a performance target I'll throw out the immutable object instances that hog my caches (after measurement, of course).

^ | v · Reply · Share ›

**Lambda Pool** → Andre · 4 months ago

its not proven by any benchmark that immutable is slower than mutable its only on your mind.

^ | v · Reply · Share ›

**Lambda Pool** · 5 months ago

if he is wrong, a lot of prominent people at Science Computer are too, like Joe Armstrong and the author of Clojure Rich.

1 ^ | v · Reply · Share ›

**Martin** → Lambda Pool · 5 months ago

Are you trying to pull an argument by authority? Good luck.

1 ^ | v · Reply · Share ›

**Martin** · 5 months ago

You're still wrong.

"It is a living organism representing a real-world entity inside the object's living environment (a computer program)."

How exactly are "living organisms" immutable?

The problem is not that people don't understand you. Everything's fine with immutability. We all get it. What's wrong is your assertion that all objects should always be immutable. All of them. Always. Period. And anyone claiming otherwise just doesn't get it.

Your assertion that objects are real-world entities is in direct contradiction with the assertion that they don't mutate. Why? Because real-world entities do mutate, constantly. Perhaps you mean that an object is a representation of a real-world entity (which is closer to the truth). But with that definition, an object could be (and actually is) mutable. But perhaps you mean that an object is the representation of the state of a real-world entity at a given, fixed



point in time. Now, that's settle your case for immutability. If and only if an object represents something immutable (such the state of an object at any given point in the past, however recent that past is) it makes sense to say that the object is immutable.

The fallacy with your File example (similar to the sad "dog" example you used in the other posts), is that the File object does not represent the actual file. It represents a reference to a file (such a reference could, in turn, be mutable, but for the purposes of the discussion we'll pretend it could not). If it truly was a representation of the file (not a reference to the file), its state would not be the filename, but the bytes in the filesystem, the access permissions, the creation date, the date of last modification, audit information, and anything that we consider to constitute a file.

Let me put it the other way. In Java, you could have two File objects referencing the same actual file in the filesystem. But that's an implementation issue. The designer of the File class said: "objects of this class represent references to files in the filesystem". But in OOD, you could very well have an ActualFile object and multiple File objects acting as references to it. ActualFile could (and probably should) be mutable. You can add content to an ActualFile without creating a new one. Sure enough, modifying the ActualFile referenced by the multiple instances of File do not change, in any way, the state of any of those File objects. The fact that you can code the ActualFile class to be immutable does not mean, in any way, that you should.

There's a difference between OOD and OOP. In OOD you can and should have mutable objects.

2 ^ | v • Reply • Share ›



**Lambda Pool** ➔ Martin • 5 months ago

well its clear for me that you don't understand anything about the problem. mutable objects can be easily corrupted.

there is no such thing of "represent real world" the computer is a simulation you can't represent the real world because its completely impossible from a static machine like a computer.

keep it real.

^ | v • Reply • Share ›



**Martin** ➔ Lambda Pool • 5 months ago

I like how you argument without using arguments. Trollbaiting at its finest.

1 ^ | v • Reply • Share ›



**Marcos Douglas Santos** ➔ Martin • 5 months ago

"It is a living organism **representing** a real-world entity inside the object's living environment (a computer program)."

See he said: representing a real-world.

"How exactly are "living organisms" immutable?"

I think this is confusing for most people. IMHO he shouldn't say "living organisms" because these objects only **represent** "living organisms" or entities in a real-world. But the concept is right.

I said before:

*Objects represent entities in real-world. If they can change their own properties, they can change the reality. So they must be immutable.*

So, if an object, eg User, could change your Name (STATE) using setName() you do not have an object that represent a real-world entity, you have a "virtual clone" of this entity.

^ | v · Reply · Share ›



**Martin** → Marcos Douglas Santos · 5 months ago

"I think this is confusing for most people"

It was a rethorical question, actually. It's not confusing to me at all.

"if an object, eg User, could change your Name (STATE) using setName() you do not have an object that represent a real-world entity, you have a "virtual clone" of this entity"

But the author said that objects ARE real-world objects, not representations. That's why I elaborated further on. And, as I said, there is something that is the user, in a very real sense. Whether you chose to represent this with an object in your software or not, it's another matter. The author uses the File example and explicitly chose not to represent File as an object, but the reference to file. This is what he makes immutable: the reference. In your case, the User is a reference, not the actual representation of the user, because you chose to extract the representation out of your software. Why? Well, I can't speculate as to what motivates that decision.

But there's a huge number of representations of "User", all of them mutable,

[see more](#)

1 ^ | v · Reply · Share ›



**Marcos Douglas Santos** → Martin · 5 months ago

>>"But the author said that objects ARE real-world objects"  
I disagree. Yegor always said objects represents, not ARE, real-world objects.

>>"But there's a huge number of representations of "User"  
I'm sorry, my fault. I meant User class. So:

```
User bob = new User("bob");  
bob.setName("john");
```

This I meant.

You do not have an User object but an instance of that - of course you know that.

>>"From the point of view of the relational paradigm, any other representation is a reference"

The relational paradigm it has nothing to do with objects, only data.

>>"There's no user in the real world"

Yes, only bob, john, marcos, martin, etc. :)

>>"In the real world, you plug-in your refrigerator and it runs"

Yes, in the **real world**!

As I said before, if you have an object in your software that represents your refrigerator, you can't change the real-world using `setXXX` in your object, right? -- maybe only Neo in Matrix.

You can not change the reality through your object so keep your state immutable.

^ | v · Reply · Share ›



**Martin** → Marcos Douglas Santos · 5 months ago

"I disagree. Yegor always said objects represents, not ARE, real-world objects."

Yes, my mistake. But he also said that objects are "living organisms", that's what I meant.

"You can not change the reality through your object so keep your state immutable."

This very point is what's wrong, in my opinion. Since reality changes, your model should mutate to accurately represent the new reality. Because an object model is not, and it's not supposed to be, a set of "snapshots" or "virtual clones". In an object model you have one, and only one, User object representing "Marcos Douglas Santos". In the realm of your system, that object IS the user.

This subject is really fascinating to me, but unfortunately I'm at work right now. I'll try to revisit later.

1 ^ | v · Reply · Share ›



**Marcos Douglas Santos** → Martin · 5 months ago

"Yes, my mistake. But he also said that objects are "living organisms", that's what I meant."

I agree. As I said before, he shouldn't say "living organisms" because these objects only represent "living organisms" or entities in a real-world.

"Since reality changes, your model should mutate to accurately represent the new reality"

Of course, that's right.

See <http://www.yegor256.com/2014/1...>

Imagine you have two User objects (`obj1` and `obj2`) that represents the same real-world entity, eg, you Martin.

If the `obj1` uses `setName('bob')`, changing your own state, the `obj2` will

see that? Do you think this is correct? Because that, the data is stored outside the object; only the state needs to exist inside the object.

"In the realm of your system, that object IS the user."

That object **represents** the user. That is the difference.

If this object represents the user in real-world, I can not change your name, eg, through an object because this object represents him! So, the object need to represents the user - an user in real-world - not change your attributes virtually.

^ | v · Reply · Share ›



**Yegor Bugayenko** author ➔ Marcos Douglas Santos · 5 months ago

Let me try to clarify. First, I believe that an object indeed **is a living organism** and it lives inside its living environment, a method, for example. Second, **represents** a real-life entity, which lives somewhere else, outside of object's living environment, for example in another method or on disc or in Buenos Aires. Third, an immutable object is an **empty-handed** representative of a real-life entity, it doesn't have any personal belongings, all it has is a set of coordinates of the entity it represents. And finally, an empty-handed object is not dumb, it can perfectly transfer our requests to that real-life entity and pass back its answers. But it never keeps anything to itself, everything goes through its hands. It is **not greedy** :) Make sense?

1 ^ | v · Reply · Share ›



**Marcos Douglas Santos** ➔ Yegor Bugayenko · 5 months ago

For me, yes.

Now I understood better when you say "object indeed is a living organism". This is not THE organism in real-world, but other that represents this one. Maybe will confuse (more) some people, but I understood.

Do you not agree with something I said now or before?

^ | v · Reply · Share ›



**Yegor Bugayenko** author ➔ Marcos Douglas Santos · 5 months ago

I don't think I disagree, just wanted to clarify this "living organism" definition. It's not mine, by the way, I got it from "Object Thinking" book by David West: <http://www.amazon.com/gp/produ...>

1 ^ | v · Reply · Share ›



**Invisible Arrow** · 5 months ago

Would a cached result in an object lead to the object being mutable? A small trivial example given below:

```
public class Sum {
```

```

private final int a;
private final int b;
private int sum;
private boolean calculated = false;
public Sum(int first, int second) {
    this.a = first;
    this.b = second;
}
public int sum() {
    if (!calculated) {
        sum = a + b;
        calculated = true;
    }
    return sum;
}
}

```

Here we're sort of "caching" the result. Of course, this is a rather trivial operation in this example which doesn't require any caching (also not considering thread safety), but let's say there is some costly operation which is going to be invoked multiple times, where caching the result might be beneficial. From the client perspective, it gets the same behavior each time, but the object is not strictly immutable.

Also, maybe a better design would be to handle the caching with AOP?

^ | v • Reply • Share ›



**Gunter Hå Olsen** → Invisible Arrow • 5 months ago

So what you're asking is whether a Cache Proxy object should be mutable or not?

Data - State - Behaviour.

Data: The cache data. Possibly in memory, or in a file. The data can change at any time.

State: The cache meta data, represented as an object or a struct containing fixed values that do not change, such as where the cache data is stored and where the cache data comes from (which invocation is associated with it), and so on.

Behaviour: Requesting the particular data, through the cache proxy. The Cache Proxy is immutable in that this particular object cannot change where the data comes from, or where it is stored/cached. But what is mutable is the data itself, which is not part of the meta data (STATE), but part of its BEHAVIOUR, and conceptually a part of the object.

But the object remains immutable and cannot change what it represents, it will always point to the cached news article #43, you can't reuse the same object to cache a different article. But the article can still change, and then the cache needs to be updated with the new article text. But you can't update the ID (43), since that means it is no longer the same article. Everything up to this point which has used


this Cache object has expected it to mean Article 43. Now you've changed its ID to 50, and everything breaks.

In Java you could make bastard classes which has both immutable and mutable properties, with the mutable ones as a representation of memory access, since you can't actually do it properly. But the optimization and maintainability issues won't change from this.

2 ^ | v • Reply • Share ›


 **Martin** ➔ Gunter Hà Olsen • 5 months ago

More from [yegor256.com](#)

 **yegor256.com** recently published


**How to Protect a Business Idea While Outsourcing**

8 Comments  Recommend 

 **yegor256.com** recently published

**Software Quality Award - Yegor Bugayenko**

10 Comments  Recommend 

 **yegor256.com** recently published

**Three Things I Expect From a Software Architect**

23 Comments  Recommend 