DISQUS Home 9+ Inbox Discover Discuss

Community

256 yegor256.com

43 Comments • Created 6 months ago

How an Immutable Object Can Have State and Behavior?

I often hear this argument against immutable objects: "Yes, they are useful when the state doesn't change. However, in our case, we deal with frequently changing objects. We simply can't afford to create a new document every time we just ne…

(yegor256.com)

43 Comments

Recommend Share Sort by Newest

Join the discussion…

Yuriy • 2 months ago

Yegor, please check links in phrase "(by the way, I'm using@Immutable annotation from jcabi-aspects)" - they are broken.

Reply • Share ›

Yegor Bugayenko author → Yuriy • 2 months ago

Thanks! Fixed.

Reply • Share ›

Conformer • 3 months ago

While I like your purist mindset (being somewhat of a purist myself), I do not think that the Memory class makes much sense in Java to render objects "immutable". All you are really doing is creating a *workaround* to make your objects *appear* immutable, but, in reality, they are not. Java already gives us memory write/read access in a much more convenient way. They are called mutable properties. This is exactly what the Memory class would be providing, so it would not make much sense to have such a class, since you already have what you need.

Instead of forcing "immutability" in these situations, it would be better to just go along with using mutable properties. This is when their usage is perfectly justifiable.

I think Java made the right choice to disallow direct memory access, since it is unnecessary and only makes your code more error-prone.

1 • Reply • Share ›

Hans-Peter Störr • 5 months ago

I think with this approach you are giving up most of the benefits immutability can have, because you reject a very important concept that perfectly complements it: side-effect free methods. This means, when you call a method of an immutable object again with equal (immutable) arguments, you will get an equal result. (That's probably "constant objects" in your terminology.)

Together this gives the (for me) major drivers for immutability:
1. A major simplification: the objects cannot change behind your back, anymore.
2. Automatic thread safety and easier concurrency.
None of these are present in your design. More or less, you even discourage this. To have basically every method access or modify external state, IMHO defies the very concept of immutability. This is the most visible in the "mutable memory" design: you just move the mutable attributes into a separate memory area, rename the now external state to "behaviour" and call it immutable. But it just behaves like a plain old mutable object. So I think that's just what it is - an astonishingly complex mutable object, as are the objects directly accessing the database. They have just the same issues with concurrency. And you get back the problems with memory corruption and memory leaks and even performance problems, that garbage collected languages have done away with so nicely.

Going for immutability requires a major, but important, mindshift. You probably can't view the objects as representation of real entities anymore, but as snapshots of these - any relevant changes require replacing the object by a fresh one. I'm quite curious whether the book "object design" allows for this, or whether there is a conceptual conflict. I don't think there is a conflict of immutability with object orientation per se, but probably with the philosophy of identifying one object instance with the real thing.

5 • Reply • Share ›

Yegor Bugayenko author Hans-Peter Störr • 5 months ago

Working with just immutable data structures ("constant objects") is not possible. You need to abstract mutable real-life entities. Look at java.io.File, a perfect example of an immutable object that is an abstraction of a mutable real-life entity (file on disc). The key mindset shift issue here is our belief that an object is a piece of data in memory with methods attached to it. Which is not true.

• Reply • Share ›

Hans-Peter Störr Yegor Bugayenko • 5 months ago

Of course you can't make everything immutable and side-effect free - you need to have input and output as well. But these are relatively rare -

especially write operations. So it is a good idea to restrict them to special places, so that they interfere as little as possible with your program. And only in those relatively few places you need careful reasoning about changing state, side effects and concurrency. Thats a rather big gain!

In Haskell, for instance, where everything is immutable, input output operations get a special treatment as IO-Monads (which are immutable in your sense with side effects), since the operations with side effects are troublesome. Whereas you actually sugest to put them everywhere.

Unfortunately I have trouble googling up a good explanation of that perspective from a Java viewpoint. This is more prominent in functional languages, where this is also called "pure functions" - see e.g. http://www.braveclojure.com/fu... .

1 • Reply • Share ›

Yegor Bugayenko author Hans-Peter Störr • 5 months ago

Of course it's better to work mostly with constant objects. No doubts about it. But sometimes you need mutability, when your object is abstracting a mutable real-life entity. IO-Monads from Haskell is a good example. I think we're on the same page here, just arguing about terminology :) This article doesn't suggest to make all objects abstracting memory pieces. Instead, this should be the last resort. In most cases it's better to have constant objects, which have constant methods.

• Reply • Share ›

Geoffrey • 5 months ago

This is a great way to look at objects, but I think it's a little too hard-line.
1. A variable is just a piece of data in memory, so how would it be any different to use a variable than to access memory directly?
2. Setting an array to final and then changing the array doesn't mean the object is immutable, it just means your variable is in an array instead of its own.
3. Are you suggesting to throw optimization out the window by writing data to persistence instead of using a variable for the sake of a conceptually (but not really) immutable object?

2 • Reply • Share ›

Yegor Bugayenko author Geoffrey • 5 months ago

An object property and a variable are two different things, conceptually speaking. In modern languages, like Java or C++, they are very similar or even identical, but they should not be. "*A variable is just a piece of data in memory*" - this is C-style type of thinking. You're thinking like a computer. You're thinking in terms of bytes and bits. But in OOP you should not think that way. You need "object thinking" in order to write good object-oriented software. You should forget about bytes and memory. Also, should forget about optimization (in terms of bytes and bits). I understand that it's hard, but please try :) If you manage to look at things from a different perspective, you will enjoy OOP!

Reply • Share ›

Afroradiohead • 5 months ago

First off, that was a great read. Thank you for writing that up Yegor.

Secondly, I've been messing around with the idea that "Objects are immutable, but can be changed based on it's own decisions". Of course this goes into a whole different style of coding where you put event listeners in the constructors of every object and they listens to the events they need to listen to. To give an quote I made up that might better explain why this may work: "Does a Component directly change the child SubComponent it contains? Or does the SubComponent decide to change based on the Component".

Anyway, I know it's a pretty abstract explanation and example but from your experience do you feel that this design pattern is plausible?

• Reply • Share ›

Yegor Bugayenko author Afroradiohead • 5 months ago

I think it totally depends on the reality you're trying to abstract. Each component is an abstraction (a representation) of a real-life object. If that object is passive and needs requests in order to be changed - model it this way in Java. For example, a file could be a passive object. Only when necessary you call `save(String)` and it changes the content. On the other hand, you may have the same file as an active object, which does something on its own and initiates interactions with other objects. For example, a file may have a method `whenChanged(Listener)`, which is called by the file itself, when its content is changed.

Thus, it totally depends on the reality you're abstracting :)

• Reply • Share ›

Afroradiohead Yegor Bugayenko • 5 months ago

Nice! Thanks Yegor, I totally get what your saying. I appreciate your reply.

Just something to toss out there, at this very moment I'm doodling some ideas of how to think of objects and storing them. Here's the summary of some random thoughts:

"Classes are a just proxy of a visual/physical object... the class name is just our unimportant label... properties are important... databases are a memory card of that object... users are looking at stored data from a different perspective, but it's just data"

This is my thinking process. From your perspective, are there thoughts that you immediately see is worth building upon or discarding?

• Reply • Share ›

Yegor Bugayenko author Afroradiohead • 5 months ago

You're on the right track, check these articles:
http://www.yegor256.com/2014/1... and
http://www.yegor256.com/2014/1.... Also this one will help:
http://www.yegor256.com/2014/0

http://www.yegor256.com/2014/0...

1 | • Reply • Share ›

Bruno Martins • 6 months ago

Software isn´t bad just because you have mutable objects. Immutable objects have its advantages , but so do Mutable!

I think that the reason why you think so strongly on immutable objects, might be related to the fact that developers don´t use them as much as they should, yes. But, there is no black or white here, there is pros & cons for both of them.

You then go by your way to explain the whole "A document's title should not be part of its state. Instead, the title should be its behavior" , "proxy of a real-life ". Which then, not only forces you to use an instance´s interface not depending on an object state at all, and combined with the "direct-memory access" thing you wish you could had in all high level languages .... breaks the very OOP concepts you are here trying to defend.

4 | • Reply • Share ›

Yegor Bugayenko author Bruno Martins • 6 months ago

Can you list some of the advantages of mutable objects please?

| • Reply • Share ›

valenterry Yegor Bugayenko • 6 months ago

One advantage is, that mutable objects are sometimes more intuitive and more natural to the human way of thinking. The reason is that in real life we also have mutable objects right? Our brain is trained to work with them. It has something to do with identity.
A list of files for example can be converted (with map function) to a new list of strings. This feels natural in some kind at least after you got used to it. But there are other objects which do not naturally feel good as immutable, at least not for me and many others. E.g. in a video game when we have a player. The player has a lifepoints. If he gets hit, I don't create a new player, no. I change its lifepoints (the lifepoints however can be immutable though). It is just more natural. Or do you feel creating a new player is more natural? I would wonder.

Bruno is right when he says that you use an interface which is not depending at object state. That is really really bad. I expect my immutable document instance not to ever change at all and to behave always the same. As a user of this class, I am not expecting to get such an document object, use it and then - out of nowhere - it returns different values than before. You can, of course, document that behaviour. But you are just moving the state anywhere else. How is the mutable state in a database/filesystem better than in the program? It isn't at all. So instead, stay with your normal immutable solution, because that is the correct one. It may be slow, so what? Correct solutions are sometimes slow. Deal with it, take a better language, get faster hardware. If you can't do either, just use mutable objects and don't try to make them dependent from another mutable object somewhere else.

6 | • Reply • Share ›

Bruno Martins ➦ Yegor Bugayenko • 6 months ago

Try to restructure & re-balance an efficient binary tree search or graphs only with immutable objects.

Immutable objects have advantages and should be used way more than they are? Yes, absolutely ! But, anyone that says "always" or "never" should be distrusted.

3 ^ | ˅ • Reply • Share ›

Yegor Bugayenko author ➦ Bruno Martins • 6 months ago

Again, can you give at least one advantage of mutable objects? "Try to restructure..." is hardly an argument because in order to argue against it I have to spend a few hours of work, and it still won't be a strong counter-argument. So, please, give an advantage, just like I listed advantages of immutable objects here: http://www.yegor256.com/2014/0...

^ | ˅ • Reply • Share ›

Bruno Martins ➦ Yegor Bugayenko • 6 months ago

Wasn´t expecting you to actually implement anything. But, the problem is itself explanatory. When balancing or re-structuring trees; allocating and instantiating new objects for every tree, subtree and node, when the updating the references was all you needed.

Your background might be web development (guessing), but in general practice, design solutions for software engineering ... there is no black & white, right or wrong. All has a context and a scope.

1 ^ | ˅ • Reply • Share ›

Craig • 6 months ago

Where and how the document is stored is not part of the document state, identity or behavior. You should have document store that knows how to store documents and your document should know only about it's title and id. Your design will have some very bad practical implications as well. Dependency injection being the most obvious. Also what happens when the document moves from memory to S3? Is that a new document or is all that logic inside your domain? This also depends on whatever storage code you are proxying to being thread safe.

You are much better off making copies of the objects and leaving it the VM to manage the memory. If all your objects are immutable copies are not nearly as expensive as you would think. The new object will not contain new copies of all it's fields, but pointers to the existing ones.

^ | ˅ • Reply • Share ›

Leonard Brünings • 6 months ago

Conceptually speaking, this document is acting as a proxy of a real-life document that has a title stored somewhere — in a file, for example document that has a title stored somewhere — in a file, for example.

The problem here is that a real world document is *NOT* immutable, so how can the proxy be immutable?

A good object is immutable, and good software contains only immutable objects.
I don't agree with this sentence, mutability and immutability do not determine if an object is good.
And software is not bad just because it contains mutable objects.

In your previous post you listed these properties of immutable objects, and I wholly agree with them.

* Immutable objects are simpler to construct, test, and use.
* Truly immutable objects are always thread-safe.
* They help avoid temporal coupling.
* Their usage is side-effect free (no defensive copies).
* They always have failure atomicity.

see more

5 ^ | ∨ • Reply • Share ›

Hans-Peter Störr ➦ Leonard Brünings • 5 months ago

I like the listing of properties of immutable objects Leonard Brünings listed in his comment http://www.yegor256.com/2014/1... here.
Unfortunately, the design discussed in this article - especially the mutable memory idea - violates these very much. You will have the same concurrency issues as you have with plain old mutable objects, since the external state kept in the external memory can be trashed by other threads behind your back. Since document.title(text) modifies the title (now just stored in an separate memory area), you have temporal coupling between document.title(text) and document.title(), this call is not side-effect free and, if you don't take care, not even thread-safe.

^ | ∨ • Reply • Share ›

Yegor Bugayenko author ➦ Hans-Peter Störr • 5 months ago

Yes, it's all true, but there is a difference, I believe, between thread-safe objects and thread-safe methods. Objects of class java.io.File are immutable and thread-safe, but their methods are not thread-safe. This article explains this idea better, I hope: http://www.yegor256.com/2014/1...

^ | ∨ • Reply • Share ›

Hans-Peter Störr ➦ Yegor Bugayenko • 5 months ago

I'm even more confused. Just seen from the outside, i.e., the other objects that use such an "immutable" object: What would be the behavioural difference between an object based on your "mutable memory" design, and an object that has exactly the same public methods, but just stores the data in a private byte array instead, or even just in plain old mutable attributes? If there is none, why bother with such an complicated and probably brittle construction?

• Reply • Share ›

Yegor Bugayenko author ➦ Hans-Peter Störr • 5 months ago

I'm using `@Immutable` annotation in all my Java projects. This annotation controls every class it is attached to and prohibits any properties that are not `final`. This annotation ensures that objects are truly immutable. That's why, it's a technical limitation for me.

On the other hand, it is a question of language design. Java (and many other languages) mix together memory for data storage/manipulation and memory for objects. This creates a confusion which leads to mutable objects.

• Reply • Share ›

Hans-Peter Störr ➦ Yegor Bugayenko • 5 months ago

I've asked too many questions at once. I'm sorry to repeat myself, but I think that's a crucial question: Is there any difference in the behaviour (as visible from the outside) of your "mutable memory" objects and an object with the same public methods that's simply implemented with plain old mutable private fields?

• Reply • Share ›

Yegor Bugayenko author ➦ Hans-Peter Störr • 5 months ago

There is absolutely no difference, if we're talking about Java or C++.

• Reply • Share ›

jdon framework ➦ Leonard Brünings • 6 months ago

in DDD, Immutable object is Value Object that has no ID, but Mutable Object is a Entity, an aggregate root entity will animate all objects in its aggragate edge , jdon framework (http://en.jdon.com) uses in-memory cache as the Memory class that keep up an aggregate root entity

• Reply • Share ›

Yegor Bugayenko author ➦ Leonard Brünings • 6 months ago

Many thanks for this comment! You raised so many interesting questions that I have to write a new post for each of them :) And I'll do it in the next few weeks. For now, to answer something, I will say that you confuse immutable objects with **constant objects**. Both of them never change their state, but constant objects always behave the same way. This is an immutable object:

```
class ImmutableFile {
  private final File file;
  ImmutableFile(File f) {
    this.file = f;
  }
  public String read() {
    // read and return file content
  }
}
```

And this is a constant object:

```
class ConstantFile {
  private final String content;
  ConstantFile(File file) {
    this.content = // read file
  }
  public String read() {
    return this.content;
  }
}
```

See the difference?

• Reply • Share ›

Invisible Arrow Yegor Bugayenko • 4 months ago

Let's say that both the classes defined above implement a common interface 'File' which has a 'read' method.
When it comes to checking for equality (the 'equals' method), would the two implementations be considered equal, if they provide the same behavior?
A more general question is: Would we consider implementations to be 'equal' if they exhibit the same behavior?
Ex: In Java, a 'HashSet' containing the same elements is considered equal to a 'LinkedHashSet' containing the same set of elements.

1 • Reply • Share ›

Yegor Bugayenko author Invisible Arrow • 4 months ago

Yeah, it's a good question. I think, yes, if behavior (implemented interfaces) and state (encapsulated data) are identical - objects should be identical. I hit this issue very often in Java and haven't found a solution yet.

• Reply • Share ›

Leonard Brünings Yegor Bugayenko • 6 months ago

Well I think you have a different definition about immutable objects than most people then.
In fact immutable and constant object can mostly be used interchangeably, but they have
subtle differences.

In a language that supports true constants, a constant object is defined at compile time
and will never change during the runtime. So you cant create new instances of a constant
object. A constant object is of course also immutable.

An immutable object never changes after its creation, but this creation can occur at runtime.
So you can create new instances of this object, but one instance will never change in its state
or behaviour. Since its behaviour can only be determined by its internal state (see also https://disqus.com/home/discus....

see more

1 • Reply • Share ›

Yegor Bugayenko author → Leonard Brünings • 5 months ago

This is my answer: http://www.yegor256.com/2014/1...

• Reply • Share ›

valenterry → Leonard Brünings • 6 months ago

I agree, Yegors understanding and definition of immutability regarding software design / classes / objects seems to be different to how most understand it. Yegor, I think you should clarify this in your post as it will save many discussions and missunderstandings here.

1 • Reply • Share ›

Marcos Douglas Santos → valenterry • 6 months ago

IMHO he already clarified... but the most people still continues thinking at the same way as learned about objects and classes.

Yegor's point has some problems, especially about performance, but the basic idea is not new. And I think he is is right, whether we like it or not. Make sense for me.

1 • Reply • Share ›

Invisible Arrow • 6 months ago

Would it make sense to use a 'Map' (declared final, but mutable ex. 'HashMap') instead of an 'Array' to represent mutable memory in Java (as we currently don't have a 'Memory' class)?
Only reason for 'Map' being ease of usage compared to an 'Array'. But I realize it's not strictly the 'Memory' we're trying to represent.

Also in terms unit testing, how do we mock 'Memory' as it is being initialized in the constructor of the object.
Should 'Memory' be injected as a dependency (as part of the constructor)? That seems a bit odd :), but just thinking how we could isolate the state for unit testing.

• Reply • Share ›

Yegor Bugayenko author → Invisible Arrow • 6 months ago

Sure, it can be a perfectly mockable class/interface, injectable via constructor. And yes, you're right, in the mean time we can use an instance of HashMap or LinkedList or something similar. "Thanks" to Java, `final` doesn't mean that an object can't modify itself. But in my projects I'm using `@Immutable` annotation from jcabi-aspects, which doesn't allow any mutable objects inside an immutable one, take a look: http://aspects.jcabi.com/annot... And I like your way of thinking, we're on the same page! :)

• Reply • Share ›

Invisible Arrow Yegor Bugayenko • 6 months ago

Cool :) Will look into jcabi-aspects. Looks interesting :)
Was thinking on the lines of defining a simple 'Memory' interface that has save/retrieve behavior.
One simple implementation for in-memory storage would essentially contain a 'HashMap' or 'LinkedList' to store the data in memory.

• Reply • Share ›

Yegor Bugayenko author Invisible Arrow • 6 months ago

The problem is that jcabi-aspects won't allow you to use `HashMap` or `LinkedList` inside an immutable `Memory`. Try it yourself, you'll see. Such a `Memory` should be implemented internally by the JDK. So, it's a pure theoretical discussion :)

• Reply • Share ›

Invisible Arrow Yegor Bugayenko • 6 months ago

Ah I see now. I was playing around with jcabi-aspects and see that currently only array fields annotated with 'Immutable.Array' in an 'Immutable' object can technically still be mutated :) But I see from the docs that this limitation might be overcome in future versions.
Anyway, the 'Memory' concept is pretty fascinating and hope to see it implemented in some way in the JDK.
Until then, the other approaches would do the trick for now :)

• Reply • Share ›

Marcos Douglas Santos • 6 months ago

That `Memory` class you propose will contain the title of this document in only one position (pointer) globally for all application code? It should be, right?

More from yegor256.com

256 yegor256.com recently published

How to Protect a Business Idea While Outsourcing

8 Comments Recommend

256 yegor256.com recently published

My Favorite Software Books - Yegor Bugayenko

13 Comments Recommend

256 yegor256.com recently published

How to Implement an Iterating Adapter

13 Comments Recommend