Mutable vs immutable objects

Asked 16 years, 2 months ago Modified 2 years, 7 months ago

Viewed 111k times

205

I'm trying to get my head around mutable vs immutable objects. Using mutable objects gets a lot of bad press (e.g. returning an array of strings from a method) but I'm having trouble understanding what the negative impacts are of this. What are the best practices around using mutable objects? Should you avoid them whenever possible?

oop immutability mutable

Share

Improve this question

Follow

edited Nov 2, 2018 at 12:32

Machavity ♦

31.6k ● 27 ● 95 ● 105

asked Oct 18, 2008 at 7:28

Alex Angas

60k ● 41 ● 143 ● 212

`string` is immutable, at least in .NET, and I think in many other modern languages as well. – Domenic Oct 18, 2008 at 9:15

4 it depends on what a String really is in the language - erlang 'strings' are just an array of ints, and haskell "strings" are

arrays of chars. – Chii Oct 18, 2008 at 12:30

4 Ruby strings are a *mutable* array of bytes. Absolutely drives me nuts that you can change them in-place.
– Daniel Spiewak Oct 18, 2008 at 15:40

6 Daniel - why? Do you find yourself doing it by accident?
– Iraimbilanja Mar 8, 2009 at 9:43

5 @DanielSpiewak The solution for being driven nuts that you can change strings in place is simple: just don't do it. The solution for being driven nuts because you **can't** change strings in place is not that simple. – Kaz Jan 1, 2014 at 8:43

12 Answers

Sorted by: Highest score (default)

185

Well, there are a few aspects to this.

1. Mutable objects without reference-identity can cause bugs at odd times. For example, consider a `Person` bean with a value-based `equals` method:

   ```
   Map<Person, String> map = ...
   Person p = new Person();
   map.put(p, "Hey, there!");

   p.setName("Daniel");
   map.get(p);       // => null
   ```

   The `Person` instance gets "lost" in the map when used as a key because its `hashCode` and equality were based upon mutable values. Those values changed outside the map and all of the hashing became obsolete. Theorists like to harp on this point, but in practice I haven't found it to be too much of an issue.

2. Another aspect is the logical "reasonability" of your code. This is a hard term to define, encompassing everything from readability to flow. Generically, you should be able to look at a piece of code and easily understand what it does. But more important than that, you should be able to convince yourself that it does what it does *correctly*. When objects can change independently across different code "domains", it sometimes becomes difficult to keep track of what is where and why ("spooky action at a distance"). This is a more difficult concept to exemplify, but it's something that is often faced in larger, more complex architectures.
3. Finally, mutable objects are *killer* in concurrent situations. Whenever you access a mutable object from separate threads, you have to deal with locking. This reduces throughput and makes your code *dramatically* more difficult to maintain. A sufficiently complicated system blows this problem so far out of proportion that it becomes nearly impossible to maintain (even for concurrency experts).

Immutable objects (and more particularly, immutable collections) avoid all of these problems. Once you get your mind around how they work, your code will develop into something which is easier to read, easier to maintain and less likely to fail in odd and unpredictable ways. Immutable objects are even easier to test, due not only to their easy mockability, but also the code patterns they tend to enforce. In short, they're good practice all around!

With that said, I'm hardly a zealot in this matter. Some problems just don't model nicely when everything is immutable. But I do think that you should try to push as much of your code in that direction as possible, assuming of course that you're using a language which makes this a tenable opinion (C/C++ makes this very difficult, as does Java). In short: the advantages depend somewhat on your problem, but I would tend to prefer immutability.

Share Improve this answer Follow

edited Feb 15, 2020 at 21:48
jaco0646
17k ● 10 ● 67 ● 95

answered Oct 18, 2008 at 7:38

Daniel Spiewak
55.1k ● 14 ● 111 ● 120

12 Great response. One small question however: doesn't C++ have good support for immutability? Isn't the const-correctness feature sufficient? – Dimitri C. Sep 2, 2010 at 7:13

1 @DimitriC.: C++ actually has some more fundamental features, most notably a better distinction between storage locations that are supposed to encapsulate unshared state from those which encapsulate object identity. – supercat Sep 20, 2013 at 17:39

3 In case of Java programming Joshua Bloch brings good explanation on this topic in his famous book **Effective Java (Item 15)** – dMathieuD Sep 25, 2013 at 12:06

This is a nice example. Still, there's an easy "middle ground" practice that could be followed where the "key" properties of a class must be final and passed in the constructor, and the hash code computed only from those. This can often be done once in a common base class. – Randy Hudson Oct 22 at 13:22

33

Immutable Objects vs. Immutable Collections

One of the finer points in the debate over mutable vs. immutable objects is the possibility of extending the concept of immutability to collections. An immutable object is an object that often represents a single logical structure of data (for example an immutable string). When you have a reference to an immutable object, the contents of the object will not change.

An immutable collection is a collection that never changes.

When I perform an operation on a mutable collection, then I change the collection in place, and all entities that have references to the collection will see the change.

When I perform an operation on an immutable collection, a reference is returned to a new collection reflecting the change. All entities that have references to previous versions of the collection will not see the change.

Clever implementations do not necessarily need to copy (clone) the entire collection in order to provide that immutability. The simplest example is the stack implemented as a singly linked list and the push/pop operations. You can reuse all of the nodes from the previous collection in the new collection, adding only a single node for the push, and cloning no nodes for the pop. The push_tail operation on a singly linked list, on the other hand, is not so simple or efficient.

Immutable vs. Mutable variables/references

Some functional languages take the concept of immutability to object references themselves, allowing only a single reference assignment.

- In Erlang this is true for all "variables". I can only assign objects to a reference once. If I were to operate on a collection, I would not be able to reassign the new collection to the old reference (variable name).
- Scala also builds this into the language with all references being declared with **var** or **val**, vals only being single assignment and promoting a functional style, but vars allowing a more C-like or Java-like program structure.
- The var/val declaration is required, while many traditional languages use optional modifiers such as **final** in java and **const** in C.

Ease of Development vs. Performance

Almost always the reason to use an immutable object is to promote side effect free programming and simple reasoning about the code (especially in a highly concurrent/parallel environment). You don't have to worry about the underlying data being changed by another entity if the object is immutable.

The main drawback is performance. Here is a write-up on a simple test I did in Java comparing some immutable vs. mutable objects in a toy problem.

The performance issues are moot in many applications, but not all, which is why many large numerical packages, such as the Numpy Array class in Python, allow for In-Place updates of large arrays. This would be important for application areas that make use of large matrix and vector operations. This large data-parallel and computationally intensive problems achieve a great speed-up by operating in place.

Share Improve this answer Follow

edited Aug 22, 2019 at 1:47
Ihor Patsian
1,298 ● 2 ● 17 ● 27

answered Sep 18, 2013 at 4:04
Ben Jackson
341 ● 3 ● 3

14

Immutable objects are a very powerful concept. They take away a lot of the burden of trying to keep objects/variables consistent for all clients.

You can use them for low level, non-polymorphic objects - like a CPoint class - that are used mostly with value semantics.

Or you can use them for high level, polymorphic interfaces - like an IFunction representing a mathematical function - that is used exclusively with object semantics.

Greatest advantage: immutability + object semantics + smart pointers make object ownership a non-issue, all clients of the object have their own private copy by default. Implicitly this also means deterministic behavior in the presence of concurrency.

Disadvantage: when used with objects containing lots of data, memory consumption can become an issue. A solution to this could be to keep operations on an object symbolic and do a lazy evaluation. However, this can then lead to chains of symbolic calculations, that may negatively influence performance if the interface is not designed to accommodate symbolic operations. Something to definitely avoid in this case is returning huge chunks of memory from a method. In combination with chained symbolic operations, this could lead to massive memory consumption and performance degradation.

So immutable objects are definitely my primary way of thinking about object-oriented design, but they are not a dogma. They solve a lot of problems for clients of objects, but also create many, especially for the implementers.

Share Improve this answer Follow

edited Aug 22, 2019 at 3:00
Ihor Patsian
1,298 ● 2 ● 17 ● 27

answered Oct 18, 2008 at 7:58
QBziZ
3,250 ● 26 ● 24

I think I misunderstood Section 4 for the greatest advantage: immutability + object semantics + smart pointers renders object ownership a "moot" point. So it's debatable? I think you're using "moot" incorrectly... Seeing as the next sentence is the object has implied "deterministic behavior" from its "moot" (debatable) behavior. – Benjamin Oct 8, 2014 at 21:21

You are right, I did use 'moot' incorrectly. Consider it changed :) – QBziZ Oct 10, 2014 at 9:02

12

Check this blog post: http://www.yegor256.com/2014/06/09/objects-should-be-immutable.html. It explains why immutable objects are better than mutable. In short:

- immutable objects are simpler to construct, test, and use
- truly immutable objects are always thread-safe
- they help to avoid temporal coupling

- their usage is side-effect free (no defensive copies)
- identity mutability problem is avoided
- they always have failure atomicity
- they are much easier to cache

Share Improve this answer Follow

answered Jun 10, 2014 at 6:22
yegor256
105k ● 130 ● 460 ● 620

5

You should specify what language you're talking about. For low-level languages like C or C++, I prefer to use mutable objects to conserve space and reduce memory churn. In higher-level languages, immutable objects make it easier to reason about the behavior of the code (especially multi-threaded code) because there's no "spooky action at a distance".

Share Improve this answer Follow

answered Oct 18, 2008 at 7:33
John Millikin
201k ● 41 ● 215 ● 227

You're suggesting that the threads are quantumly entangled? That's quite a stretch :) The threads are actually, if you think about it, pretty close to being entangled. A change that one thread makes affects the other. +1 – ndrewxie Dec 14, 2017 at 21:11

A mutable object is simply an object that can be modified after it's created/instantiated, vs an immutable object that

5

cannot be modified (see the Wikipedia page on the subject). An example of this in a programming language is Pythons lists and tuples. Lists can be modified (e.g., new items can be added after it's created) whereas tuples cannot.

I don't really think there's a clearcut answer as to which one is better for all situations. They both have their places.

Share Improve this answer Follow

answered Oct 18, 2008 at 7:39

willurd

12k ● 5 ● 30 ● 23

3

Shortly:

Mutable *instance is passed by reference.*

Immutable *instance is passed by value.*

Abstract example. Lets suppose that there exists a file named *txtfile* on my HDD. Now, when you are asking me to give you the *txtfile* file, I can do it in the following two modes:

1. I can create a shortcut to the *txtfile* and pass shortcut to you, or
2. I can do a full copy of the *txtfile* file and pass copied file to you.

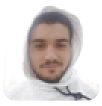
In the first mode, the returned file represents a mutable file, because any change into the shortcut file will be reflected into the original one as well, and vice versa.

In the second mode, the returned file represents an immutable file, because any change into the copied file will not be reflected into the original one, and vice versa.

Share Improve this answer Follow

edited May 13, 2022 at 23:19

answered Oct 31, 2015 at 18:27



Teodor
124 ● 8

2 This isn't strictly true. Immutable instances can indeed be passed by reference, and often are. Copying is mainly done when you need to update the object or data structure. – Jamon Holmgren Apr 20, 2016 at 2:40

@JamonHolmgren , can't disagree your point but example I'm referring to is about protecting the original file of any made change on its copies. But, by applying your case on my example would mean to return `a new read-only copy of the original file` instead (or `a read-only wrapper file on top of original one`). – Teodor May 13, 2022 at 23:18

2

Mutable collections are in general faster than their **immutable** counterparts when used for in-place operations.

However, **mutability** comes at a cost: you need to be much more careful sharing them between different parts of your program.

It is easy to create bugs where a shared **mutable** collection is updated unexpectedly, forcing you to hunt down which line in a large codebase is performing the unwanted update.

A common approach is to use **mutable** collections locally within a function or private to a class where there is a performance bottleneck, but to use **immutable** collections elsewhere where speed is less of a concern.

That gives you the high performance of **mutable** collections where it matters most, while not sacrificing the safety that **immutable** collections give you throughout the bulk of your application logic.

Share Improve this answer Follow

answered Aug 4, 2020 at 16:37

Chema
2,828 ● 2 ● 16 ● 26

1

If a class type is mutable, a variable of that class type can have a number of different meanings. For example, suppose an object `foo` has a field `int[] arr`, and it holds a reference to a `int[3]` holding the numbers {5, 7, 9}. Even though the type of the field is known, there are at least four different things it can represent:

- A potentially-shared reference, all of whose holders care only that it encapsulates the values 5, 7, and 9. If `foo` wants `arr` to encapsulate different values, it must replace it with a different array that contains the desired values. If one wants to make a copy of `foo`, one may give the copy either a reference to `arr` or a new array holding the values {1,2,3}, whichever is more convenient.

- The only reference, anywhere in the universe, to an array which encapsulates the values 5, 7, and 9. set of three storage locations which at the moment hold the values 5, 7, and 9; if `foo` wants it to encapsulate the values 5, 8, and 9, it may either change the second item in that array or create a new array holding the values 5, 8, and 9 and abandon the old one. Note that if one wanted to make a copy of `foo`, one must in the copy replace `arr` with a reference to a new array in order for `foo.arr` to remain as the only reference to that array anywhere in the universe.

- A reference to an array which is owned by some *other* object that has exposed it to `foo` for some reason (e.g. perhaps it wants `foo` to store some data there). In this scenario, `arr` doesn't encapsulate the contents of the array, but rather its *identity*. Because replacing `arr` with a reference to a new array would totally change its meaning, a copy of `foo` should hold a reference to the same array.

- A reference to an array of which `foo` is the sole owner, but to which references are held by other object for some reason (e.g. it wants to have the other object to store data there--the flipside of the previous case). In this scenario, `arr` encapsulates both the identity of the array and its contents. Replacing `arr` with a reference to a new array would totally change its meaning, but having a clone's `arr` refer to `foo.arr` would violate the assumption that `foo` is the sole owner. There is thus no way to copy `foo`.

In theory, `int[]` should be a nice simple well-defined type, but it has four very different meanings. By contrast, a reference to an immutable object (e.g. `String`) generally only has one meaning. Much of the "power" of immutable objects stems from that fact.

Share Improve this answer

Follow

answered Sep 20, 2013 at 17:36

supercat

80.8k ● 9 ● 174 ● 220

0

If you return references of an array or string, then outside world can modify the content in that object, and hence make it as mutable (modifiable) object.

Share Improve this answer

Follow

answered Dec 17, 2013 at 7:32

user1923551

4,712 ● 36 ● 29

0

Immutable means can't be changed, and mutable means you can change.

Objects are different than primitives in Java. Primitives are built in types (boolean, int, etc) and objects (classes) are user created types.

Primitives and objects can be mutable or immutable when defined as member variables within the implementation of a class.

A lot of people people think primitives and object variables having a final modifier infront of them are immutable, however, this isn't exactly true. So final almost doesn't mean immutable for variables. See example here http://www.siteconsortium.com/h/D0000F.php.

Share Improve this answer Follow

edited Feb 25, 2014 at 2:56
Community Bot
1 ● 1

answered Jan 1, 2014 at 6:20
JTHouseCat
445 ● 4 ● 4

0

General Mutable vs Immutable

`Unmodifiable` - is a wrapper around modifiable. It guarantees that it can not be changed directly(but it is possibly using backing object)

`Immutable` - state of which can not be changed after creation. Object is immutable when all its fields are immutable. It is a next step of Unmodifiable object

Thread safe

The main advantage of Immutable object is that it is a naturally for concurrent environment. The biggest problem in concurrency is `shared resource` which can be changed any of thread. But if an object is immutable it is `read-only` which is thread safe operation. Any modification of an original immutable object return a copy

source of truth, side-effects free

As a developer you are completely sure that immutable object's state can not be changed from any place(on purpose or not). For example if a consumer uses immutable object he is able to use an original immutable object

compile optimisation

Improve performance

Disadvantage:

Copying of object is more heavy operation than changing a mutable object, that is why it has some performance footprint

To create an `immutable` object you should use:

1. Language level

Each language contains tools to help you with it. For example:

- Java has `final` and `primitives`
- Swift has `let` and `struct`[About].

Language defines a type of variable. For example:

- Java has `primitive` and `reference` type,
- Swift has `value` and `reference` type[About].

For `immutable` object more convenient is `primitives` and `value` type which make a copy by default. As for `reference` type it is more difficult(because you are able to change object's state out of it) but possible. For example you can use `clone` pattern on a developer level to make a `deep`(instead of `shallow`) copy.

2. Developer level

As a developer you should not provide an interface for changing state

[Swift] and [Java] immutable collection

Share Improve this answer Follow

edited May 12, 2021 at 8:31

answered Jul 16, 2020 at 12:58

yoAlex5

34k ● 10 ● 223 ● 239