The Importance of Immutability in Java

by Sam Atkinson MVB · Apr. 05, 16 · Java Zone · Opinion

Take 60 minutes to understand the Power of the Actor Model with "Designing Reactive Systems: The Role Of Actors In Distributed Architecture". Brought to you in partnership with Lightbend.

One of the consistent criticisms of Java is that it lacks a formal immutable type. We can (and should) make a good attempt at creating immutable Objects, even though they may be inherently flawed due to the nature of the JVM.

An immutable object is one whose state cannot and will not change after it's initial creation. Immutable objects are great, mostly because they are Thread safe (and threaded code should be avoided as much as possible). You can pass them around without fear they will be changed. I highly recommend you spend some time with a functional language like Scala to really appreciate the amazing power immutable Objects can have (but then come straight back, because Scala has a whole different bag of problems).

Interestingly, almost all of the new features in Java 8 (Date and Time, Optionals and Streams) have been implemented in an immutable fashion. This allows much of the performance benefits that can come from things such as parallel Streams. Immutable Objects allow us to create side-effect free functions as seen in Functional Programming languages which are the basis for creating fast, lock free code.

How to Create an Immutable Object in Java?

The main action is to mark all fields as final. This obviously means they cannot change after initial construction. Beware though, that you can still have a final field where the Object contained is mutable. In this case it is necessary to copy the Object when initially set in the constructor, and provide a copy of the Object when it is being accessed from outside the class.

This obviously adds complication to our code and design. Ideally you should follow the original advice from Effective Java:

> **"Classes should be immutable unless there's a very good reason to make them mutable....If a class cannot be made immutable, limit its mutability as much as possible."**

If you have an Object field in your class endeavour where possible to make it immutable too.

If you succeed in making all your fields immutable then you may choose to also make them public as I do- the fields cannot be changed post construction and are only used for reading. This makes the addition of a getter redundant. In my code bases if I'm accessing data using the fields then I know that class is Immutable. The main exception to this is when I'm writing libraries, as it's much harder to refactor if you need to introduce mutability later on. If you own all the code though, to move from field access to method access is one shortcut in IntelliJ.

You should also make your class final to defend from subclassing. It would be possible to create mutable subclasses and thus ruining your hard work.

It takes a concious effort to create immutable classes but it should be your target wherever possible. A common anti-pattern I've seen in developers is that after creating a constructor and set of fields they generate getters/setters for all of them. Do not do this! Firstly, code should only be written if it is needed; if you have no test or prod code that is accessing a field then it does not need a getter; if no other code is trying to change the field after creation then you don't need a setter. Create code on demand and do not optomize early.

One of the main criticisms of immutable objects is that it can lead to a proliferation of objects and as a result performance issues- have the potential for a significant amount of churn of new objects as you're having to create a new one for any state change. Unless you're in a crazy high performance environment (and you're almost certainly not) then it really isn't an issue. Objects are cheap. Even Oracle thinks so:

> **"The impact of object creation is often overestimated and can be offset by some of the efficiency associated with immutable objects. These include decreased overhead due to garbage collection, and the elimination of code needed to protect mutable objects from corruption."**

This is a classic case of eagerly optomising code. If you create immutable code and it turns out you're hitting massive performance roadblocks then refactor. For most cases you'll be fine. There are obvious exceptions; data structures tend to be much easier to implement and more performant if using mutability. As always, apply common sense to your approach, but default to immutability.

Like This Article? Read More From DZone

How Functional Programming Will (Finally) Do Away With the GoF

Observer Pattern Tutorial with Java Examples

Patterns

Singleton Pattern Tutorial with Java Examples

Free DZone Refcard
Getting Started With Kotlin

Topics: IMMUTABILITY , JAVA , DESIGN PATTERNS

Opinions expressed by DZone contributors are their own.

Java Partner Resources