Kamal Mahmassani

28/11/2017

OOP Assignment

# 1 Immutable Objects:

“An object is considered *immutable* if its state cannot change after it is constructed” (Java Tutorials Docs). A number of advantages comes from using immutable objects.

## 1.1 Advantages:

* **Thread Safety Because no methods can modify the state of immutable objects, they are thread safe no matter how many threads use them and do not require synchronization.**
* **Avoids Temporal Coupling and “Side Effects” Because state cannot be modified, we do not have to worry about the order our methods are called, as they will by definition not have any effect on our object. Also we do not have to worry about external methods modifying or tainting our objects that we pass to them, and can re use them as many times as needed.**
* **Reduces Needless Copying We can expose any immutable object as we don’t have to worry that they will be changed by the caller. This means we don’t have to defensively copy them when returning them causing unneeded object proliferation.**

## 1.2 Disadvantages:

* **Protecting Mutable Members** Extra overhead is required when returning internal mutable properties that our immutable structure may contain, they must be defensively copied before being returned.
* **Changes Require New Objects** In order to make a “change” a completely new object is created, often a copy is created with the change in place. This causes extra overhead and object proliferation, however it can be mitigated somewhat with methods such as *path copying*. The impact of this overhead is often over estimated, as Oracle says: “Programmers are often reluctant to employ immutable objects, because they worry about the cost of creating a new object as opposed to updating an object in place. The impact of object creation is often overestimated, and can be offset by some of the efficiencies associated with immutable objects. These include decreased overhead due to garbage collection, and the elimination of code needed to protect mutable objects from corruption.”

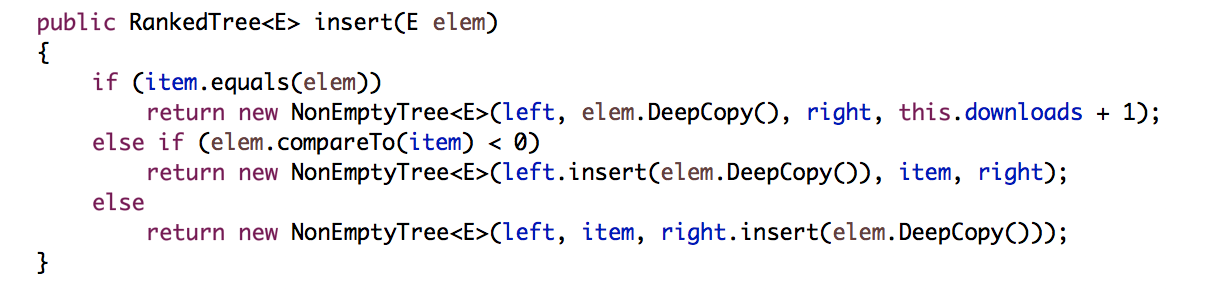


Figure Using Path Copying

## 1.3 Steps to Immutability:

* **Do not provide setter methods** We should not provide any methods that modify our fields or the objects referenced by our fields. If we fail to do this, then we allow state changes i.e. not immutable.
* **Make all fields final and private** Setting our fields as final allows them to be instantiated only once, and we will not be able to change them again, especially in the case of primitive types. However, the final keyword is only concerned with the reference, so in the case of complex types and objects, it would still allow the objects properties to be changed. We must make these and all our data members private, so that external classes cannot directly access them and their underlying objects and possible change their state.



Figure Private and Final Fields

* **Declare classes as final** This stops subclasses from extending your classes and overriding your methods and possible breaking immutability by, for example, providing references to your internal fields.



Figure Class set as final

* **Do not allow mutable fields to be changed** If our instance contains mutable objects, we must not allow these fields to be changed. We can do this by not providing methods that modify these fields, and by not sharing references to these fields. A good practice is to never store references to external mutable objects passed to our class, for example in a constructor. We should defensively and deeply copythese objects and store references to the copies. Likewise, when we need to return a mutable type, we should not return our own mutable objects, rather we should copy them and return the copies.

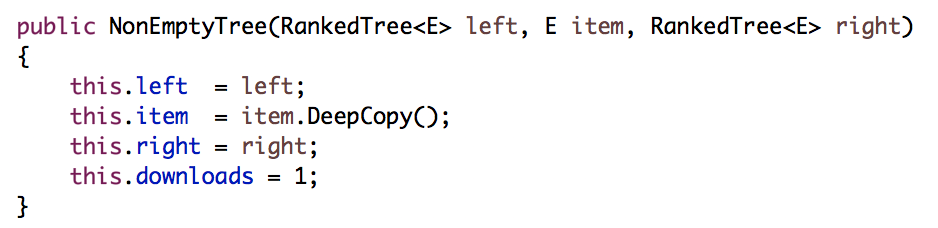


Figure Copying external mutable objects

## 1.4 Conclusion

“Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code” (Oracle Java Tutorial Docs).

The benefits of immutability in most cases far outweigh the costs. They are especially useful in many cases such as keys for maps and in sets. They make it easier to write and understand code, and can be easily shared and cached, as their values will not change. As Joshua Bloch said in 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.”